Journal of Applied Computer Science Methods

Published by University of Social Sciences

ACSM

Volume 8 Number 2 2016

University of Social Sciences, IT Institute

Społeczna Akademia Nauk
Łódź

ISSN 1689-9636

International Journal of Applied Computer Science Methods

Journal of Applied Computer Science Methods

Published by University of Social Sciences

ACSM

Volume 8 Number 2 2016

University of Social Sciences, IT Institute

Społeczna Akademia Nauk
Łódź

Instytut Technologii Informatycznych iti

INTERNATIONAL JOURNAL OF APPLIED COMPUTER SCIENCE METHODS (JACSM)
is a semi-annual periodical published by the University of Social Sciences (SAN)
in Lodz, Poland.

PUBLISHING AND EDITORIAL OFFICE:
University of Social Sciences (SAN)
Information Technology Institute (ITI)
Sienkiewicza 9
90-113 Lodz
Tel.: +48 42 6646654
Fax.: +48 42 6366251
E-mail: acsm@spoleczna.pl
URL: https://www.degruyter.com/view/j/jacsm

Print: Mazowieckie Centrum Poligrafii, ul. Słoneczna 3C, 05-270 Marki, www.c-p.com.pl, biuro@c-p.com.pl

AIMS AND SCOPE:

The **International Journal of Applied Computer Science Methods is** a semi-annual, refereed periodical, publishes articles describing recent contributions in theory, practice and applications of computer science. The broad scope of the journal includes, but is not limited to, the following subject areas:
Knowledge Engineering and Information Management: *Knowledge Processing, Knowledge Representation, Data Mining, Machine Learning, Knowledge-based Systems, Knowledge Elicitation, Knowledge Acquisition, E-learning, Web-intelligence, Collective Intelligence, Language Processing, Approximate Reasoning, Information Archive and Processing, Distributed Information Systems.*
Intelligent Systems: *Intelligent Database Systems, Expert Systems, Decision Support Systems, Intelligent Agent Systems, Artificial Neural Networks, Fuzzy Sets and Systems, Evolutionary Methods and Systems, Hybrid Intelligent Systems, Cognitive Systems, Intelligent Systems and Internet, Complex Adaptive Systems.*
Image Understanding and Processing: *Computer Vision, Image Processing, Computer Graphics, Pattern Recognition, Virtual Reality, Multimedia Systems.*
Computer Modeling, Simulation and Soft Computing: *Applied Computer Modeling and Simulation, Intelligent Computing and Applications, Soft Computing Methods, Intelligent Data Analysis, Parallel Computing, Engineering Algorithms.*
Applied Computer Methods and Computer Technology: *Programming Technology, Database Systems, Computer Networks Technology, Human-computer Interface, Computer Hardware Engineering, Internet Technology, Biocybernetics.*

DISTRIBUTION:

Apart from the standard way of distribution (in the conventional paper format), on-line dissemination of the JACSM is possible for interested readers.

Contents

CONVERGENCE ANALYSIS OF INVERSE ITERATIVE NEURAL NETWORKS WITH L_2 PENALTY

Yanqing Wen[1], Jian Wang[1], Bingjia Huang[1] and Jacek M. Zurada[2,3]

[1]College of Science, China University of Petroleum, Qingdao 266580, China
wangjiannl@upc.edu.cn; hbjia@upc.edu.cn

[2]Department of Electrical and Computer Engineering, University of Louisville, Louisville, KY, 40292, USA)

[3]Information Technology Institute, University of Social Sciences, Łodz 90-113, Poland
jacek.zurada@louisville.edu

Abstract

The iterative inversion of neural networks has been used in solving problems of adaptive control due to its good performance of information processing. In this paper an iterative inversion neural network with L_2 penalty term has been presented trained by using the classical gradient descent method. We mainly focus on the theoretical analysis of this proposed algorithm such as monotonicity of error function, boundedness of input sequences and weak (strong) convergence behavior. For bounded property of inputs, we rigorously proved that the feasible solutions of input are restricted in a measurable field. The weak convergence means that the gradient of error function with respect to input tends to zero as the iterations go to infinity while the strong convergence stands for the iterative sequence of input vectors convergence to a fixed optimal point.

Keywords: neural networks; gradient descent; inverse iterative; monotonicity; regularization; convergence

1 Introduction

Artificial neural networks have been widely used in cognitive science, computational intelligence and intelligent information processing [1, 2]. Feedforward neural networks are some of the most popular networks whose learning modes and theoretical properties are studied in numerous reports [3-5]. Backpropagation (BP) algorithm is the most broadly applied technique to train the feedforward neural networks. For BP networks, there are one or more hidden layers, in which the adjacent layer are fully connected with weights. Gradient descent methods are often employed to find the optimal solutions by charging

weights in the descent direction of objective function. Generally speaking, there are three main drawbacks of this classical BP networks: slow convergence, poor generalization and local optimal solution.To overcome these obstacles, many training improvements for BP networks have been suggested such as adding penalty terms (regularization), adaptive adjustment of learning rate and introducing momentum terms [6-10]. Actually, it is a common strategy to improve the generalization and prune more redundant weights through regularization method.

Inverse problem is one of the most important mathematical problems which tells us about parameters that cannot be directly observed [11, 12]. It is the inverse of a forward problem which deals with the results and then compute the input. Contrary to the feedforword neural networks which correspond to the forward problem, the inverse problem results in iterative inversion of neural networks.

For BP algorithm, the output error is propagated backward through the network and the error is computed by the weights. Conversely, an iterative inversion algorithm has been proposed in [13], where the weights learning is replaced by inputs learning. In this approach, errors in the network output are described with the network inputs. In addition, this iterative inversion algorithm trains by the gradient descent method. In order to solve the optimization problem of electromagnetic mechanism, a novel inverse network has been designed which effectively avoids the local minimum problem [14]. Similar to the Bonhoeffer-Van der Pol (BVP) model, an inverse function delayed network is presented by the use of anti-delay function model. It demonstrates that this proposed network can quickly converge to the optimal solution of combinatorial optimization problems. In [18], a real-time inversion of neural network has been described by combining the particle swarm method. The reconfigurable implementation of network inversion effectively reduced the computation time to near real-time levels.

For trained neural networks, over-fitting is a common problem which leads to poor generalization. To overcome this problem, a typical technique is to employ the regularization method, that is, introduce the penalty term [12-17]. We note that L_2 norm of the parameters is one of the most often used penalty terms. There are many researches on L_2 regularization which demonstrate the it can produce smooth solution and effectively control the magnitude of the parameters [14, 15, 16, 18].

As we know, the iterative inversion of neural networks has been widely used in real applications. However, it is necessary to pay attention to its theoretical analysis. In [19], an iterative inversion algorithm of neural networks with momentum has been designed and its convergence results are proved in detail. However, the boundedness of inputs can not be guaranteed which may lead to a very large solution.

In this paper, we focus on the iterative inversion algorithm of neural networks with L_2 penalty term. The monotonicity of error function has been proved which shows that the objective functions of input are decreasing along with the iterations. More importantly, the boundedness of the inputs are rigorously proved through introducing the L_2 penalty term. Furthermore, both the week and strong convergence results are obtained, that is, the gradient of error function with respect to input vector approaches zero and the iterative input sequence converges to a fixed optimal point as the iterations go to infinity.

The rest of the paper is organized as follows: in Section 2, the iterative version algorithm with L_2 penalty is presented. In Section 3, the proofs of the theoretical results are demonstrated in detail. Finally, we conclude the paper with some useful remarks in Section 4.

2 Inverse iterative algorithms with L_2 penalty

Let us begin with an introduction of an inverse iterative algorithms for neural network with three layers. The numbers of neurons for the input, hidden and output layers are p, n and 1, respectively, suppose that the input sample and the corresponding ideal output sample are $\mathbf{x} \in \mathbf{R}^p$ and $O \in \mathbf{R}$.

Let $\mathbf{V} = \left(v_{ij}\right)_{n\times p}$ be the weight matrix connecting the input and the hidden layers, and write $\mathbf{v}_i = \left(v_{i1}, \cdots, v_{ip}\right)^T \in \mathbf{R}^p \ \ i = 1, 2, \cdots, n$. The weight vector connecting the hidden and the output layers is denoted by $\mathbf{w} = (w_1, w_2, \cdots, w_n)^T \in \mathbf{R}^n$. Let $g : \mathbf{R} \to \mathbf{R}$ be given activation functions for the hidden and output layers. For convenience, we introduce the following vector valued function

$$G\left(\mathbf{u}\right) = \left(g(u_1), g(u_2), \cdots, g(u_n)\right)^T \tag{1}$$

For any given input $\mathbf{x} \in R^p$, the output of the hidden neurons is $G(\mathbf{Vx})$, and the final actual output is

$$y = g\left(\mathbf{w} \cdot G(\mathbf{Vx})\right) \tag{2}$$

The error function with L_2 regularization penalty term is

$$E\left(\mathbf{x}\right) = \frac{1}{2}\left(O - g\left(\mathbf{w} \cdot G\left(\mathbf{Vx}\right)\right)\right)^2 + \lambda \left\|\mathbf{x}\right\|_2^2 \tag{3}$$

where $\|\mathbf{x}\|_2^2 = \sum_{j=1}^{p} |x_j|^2$.

The purpose of inverse iterative algorithms is for the given output $O \in \mathbf{R}$, input $\mathbf{x}$ makes error function $E(\mathbf{x})$ to achieve its minimal value. To simplify the writing, we do the following transformation

$$\tilde{g}(t) = \frac{1}{2}(O - g(t))^2, \quad t \in R \tag{4}$$

The gradient of the error function with respect to $\mathbf{x}$ is given by

$$E_{\mathbf{x}}(\mathbf{x}) = \tilde{g}'(\mathbf{w} \cdot G(\mathbf{V}\mathbf{x})) \sum_{i=1}^{n} w_i g'(\mathbf{v}_i \cdot \mathbf{x}) \mathbf{v}_i + \lambda \nabla \left(\|\mathbf{x}\|_2^2 \right) \tag{5}$$

where $\nabla \left(\|\mathbf{x}\|_2^2 \right) = (2x_1, 2x_2, \cdots, 2x_p)^T$.

Given an initial input vector $\mathbf{x}^0 \in R^p$, inverse iterative algorithms with L_2 penalty updates the inputs iteratively by the formula

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \eta E_{\mathbf{x}}(\mathbf{x}^k) \tag{6}$$

$$= \mathbf{x}^k - \eta [\tilde{g}'(\mathbf{w} \cdot G(\mathbf{V}\mathbf{x})) \sum_{i=1}^{n} w_i g'(\mathbf{v}_i \cdot \mathbf{x}) \mathbf{v}_i + \lambda \nabla \left(\|\mathbf{x}^k\|_2^2 \right)].$$

where $\eta > 0$ is the learning rate.

For convenience, we introduce the following notations:

$$\Delta \mathbf{x}^k = \mathbf{x}^{k+1} - \mathbf{x}^k = -\eta E_{\mathbf{x}}(\mathbf{x}^k) \tag{7}$$

$$G^k = G(\mathrm{v}\mathrm{x}^k) \tag{8}$$

$$\psi^k = G^{k+1} - G^k \tag{9}$$

3 Main results and proofs

For any $\mathbf{x} \in \mathbf{R}^p$, we write $\| x\| = \sqrt{\sum_{j=1}^{p} (x_j)^2}$, where $\|\cdot\|$ stands for the Euclidean norm in $\mathbf{R}^p$. Let $\Omega_0 = \{\mathbf{x} \in \Omega : E_{\mathbf{x}}(\mathbf{x}) = 0\}$ be the stationary point

set of the error function $E(\mathbf{x})$, where $\Omega \subset \mathbf{R}^p$ is a bounded open set. Let $\Omega_{0,s} \subset \mathbf{R}$ be the projection of Ω_0 onto the s th coordinate axis, that

$$\Omega_{0,s} = \left\{ x_s \in R : \mathbf{x} = (x_1, \cdots, x_s, \cdots, x_p)^T \in \Omega_0 \right\} \tag{10}$$

for $s = 1, 2, \cdots, p$. To analyze the convergence of the algorithm, we need the following assumptions:

- $(A1)$ The activation and function g continuously differentiable, $g'(t)$ is uniformly bounded and *Lipschitz* continuous on $\mathbf{R}$;
- $(A2)$ The weight sequence $\left\{ \mathbf{w}^k, \mathbf{V}^k \right\}_{k=0}^{\infty}$ is uniformly bounded;
- $(A3)$ The initial input vector of inverse iterative algorithms with L_2 penalty $\mathbf{x}^0$ is uniformly bounded;
- $(A4)$ $\Omega_{0,s}$ does not contain any interior point for every $s = 1, 2, \cdots, p$.

We first present two useful lemmas for the convergence analysis.

Lemma 1. Let $q(x)$ be a function defined on a bounded closed interval $[a,b]$ such that $q'(x)$ is *Lipschitz* continuous with *Lipschitz* constant $K > 0$. Then, $q'(x)$ is differentiable almost everywhere in $[a,b]$ and

$$\left| q''(x) \right| \le K, \quad a.e.[a,b] \tag{11}$$

Moreover, there exists a constant $C > 0$ such that

$$q(x) \le q(x_0) + q'(x_0)(x - x_0) + C(x - x_0)^2, x_0, x \in [a,b]. \tag{12}$$

Proof. Since $q'(x)$ is *Lipschitz* continuous on $[a,b]$, $q'(x)$ is absolutely continuous and the derivative $q''(x)$ exists almost everywhere on $[a,b]$. Hence let x is a derivative point of $q'(x)$ on $[a,b]$,

$$\begin{aligned} \left| q''(x) \right| &= \left| \lim_{h \to 0} \frac{q'(x+h) - q'(x)}{h} \right| \\ &= \lim_{h \to 0} \left| \frac{q'(x+h) - q'(x)}{h} \right| \le K \end{aligned} \tag{13}$$

$$\left| q''(x) \right| \le K, \quad a.e.[a,b] \tag{14}$$

Using the integral *Taylor* expansion, we deduce that

$$\begin{aligned} q(x) &= q(x_0) + q'(x_0)(x-x_0) + (x-x_0)^2 \int_0^1 (1-t) q''(x_0 + t(x-x_0)) dt \\ &\le q(x_0) + q'(x_0)(x-x_0) + (x-x_0)^2 \int_0^1 K(1-t) dt \\ &= q(x_0) + q'(x_0)(x-x_0) + C(x-x_0)^2. \\ &\left(C = \frac{K}{2},\ x_0, x \in [a,b] \right) \end{aligned} \tag{15}$$

Lemma 2. Let $\{b_m\}$ be a bounded sequence satisfying $\lim_{m\to\infty}(b_{m+1} - b_m) = 0$. Write $\gamma_1 = \liminf_{n\to\infty} \inf_{m>n} b_m$, $\gamma_2 = \limsup_{n\to\infty} \sup_{m>n} b_m$: There exists a subsequence $\{b_{i_k}\}$ of $\{b_m\}$ such that $S = \{a \in R : b_{i_k} \to a(k \to \infty)\}$. Then we have

$$S = [\gamma_1, \gamma_2] \tag{16}$$

Proof. It is obvious that $\gamma_1 \le \gamma_2$ and $S \subseteq [\gamma_1, \gamma_2]$. If $\gamma_1 = \gamma_2$, then $\lim_{m\to\infty} b_m = \gamma_1 = \gamma_2$, simply proof $S = [\gamma_1, \gamma_2]$. Let us consider the case $\gamma_1 < \gamma_2$ and proceed to prove that $S \supseteq [\gamma_1, \gamma_2]$.

For any $a \in (\gamma_1, \gamma_2)$, there exists $\varepsilon > 0$, such that $(a-\varepsilon, a+\varepsilon) \subset (\gamma_1, \gamma_2)$. Noting that $\lim_{m\to\infty}(b_{m+1} - b_m) = 0$, we observe that b_m travels to and from between γ_1 and γ_2 with very small pace for all large enough m. Hence, there must be infinite number of points of the sequence $\{b_m\}$ falling into $(a-\varepsilon, \mathrm{a}+\varepsilon)$. This implies $a \in S$ and thus $(\gamma_1, \gamma_2) \subseteq S$ Furthermore $[\gamma_1, \gamma_2] \subseteq S$ immediately leads to $S = [\gamma_1, \gamma_2]$. This completes the proof.

Now, we are ready to prove the monotonicity theorem and convergence theorem.

Theorem 1. (**Monotonicity**) Suppose the conditions $(A1)$, $(A2)$, $A(3)$ are valid, and the learning rate satisfies (29), for any given initial input vector $\mathbf{x}^0 \in \square^p$, the error function holds that

$$E\left(\mathbf{x}^{k+1}\right) \le E\left(\mathbf{x}^k\right), \quad k = 0,1,2,\cdots. \tag{17}$$

then, there exists $E^* \ge 0$ such that

$$\lim_{k\to\infty} E\left(\mathbf{x}^k\right) = E^*. \tag{18}$$

Proof. By assumption $(A1)$ and **Lemma 1**, let

$$|g(t)|,|g'(t)|,|g''(t)|<\tilde{C}. \tag{19}$$

where $t\in R,\tilde{C}$ is constant. By assumption $(A2)$, let

$$\|\mathbf{w}^k\|\le C_1,\|\mathbf{v}_i{}^k\|\le C_2. \tag{20}$$

$i=1,2,\cdots,n,C_1,C_2$ is constant.
there holds

$$\begin{aligned}\|\psi^k\|&=\|G^{k+1}-G^k\|\\&=\sqrt{\sum_{i=1}^{n}(g(\mathbf{v}_i\cdot\mathbf{x}^{k+1})-g(\mathbf{v}_i\cdot\mathbf{x}^k))^2}\\&=\sqrt{\sum_{i=1}^{n}(g'(t_i))^2\|\mathbf{v}_i\|^2\|\Delta\mathbf{x}^k\|^2}\\&\le\sqrt{n}\max_{1\le i\le n}g'(t_i)\|\mathbf{v}_i\|\ \|\Delta\mathbf{x}^k\|\\&\le\sqrt{n}\tilde{C}C_2\|\Delta\mathbf{x}^k\|\end{aligned} \tag{21}$$

where $t_i=\mathbf{v}_i\cdot\Delta\mathbf{x}^{k+1}+\theta_i\mathbf{v}_i\cdot\Delta\mathbf{x}^k,\theta_i\in(0,1).$
Using the integral *Taylor* expansion, we deduce that

$$\begin{aligned}&E\left(\mathbf{x}^{k+1}\right)-E\left(\mathbf{x}^k\right)\\&=\left[\tilde{g}\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^{k+1}\right)\right)-\tilde{g}\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^k\right)\right)\right]\\&\quad+\lambda\left[\left\|\mathbf{x}^{k+1}\right\|_2^2-\left\|\mathbf{x}^k\right\|_2^2\right]\\&=\tilde{g}'\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^k\right)\right)\mathbf{w}\cdot\psi^k\\&\quad+\left(\mathbf{w}\cdot\psi^k\right)^2\int_0^1(1-t)\tilde{g}''\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^k\right)+t\psi^k\right)dt\\&\quad+\lambda\sum_{j=1}^{p}[(x_j^{k+1})^2-(x_j^k)^2]\\&=\delta_1+\delta_2+\delta_3.\end{aligned} \tag{22}$$

where,

$$\begin{aligned}
\delta_1 &= \tilde{g}'\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^k\right)\right)\mathbf{w}\cdot\psi^k \\
&= \tilde{g}'\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^k\right)\right)\sum_{i=1}^{n} w_i(\mathrm{g}(\mathbf{v}_i\cdot\mathbf{x}^{k+1})-\mathrm{g}(\mathbf{v}_i\cdot\mathbf{x}^k)) \\
&= \sum_{i=1}^{n}\tilde{g}'\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^k\right)\right)w_i\,\mathrm{g}'(\mathbf{v}_i\cdot\mathbf{x}^k)\mathbf{v}_i\cdot\Delta\mathbf{x}^k \\
&+\sum_{i=1}^{n}\tilde{g}'\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^k\right)\right)w_i\left(\int_0^1(1-t)g''\left(\mathbf{v}_i\cdot\mathbf{x}^k+t\mathbf{v}_i\cdot\Delta\mathbf{x}^k\right)dt\right)\left(\mathbf{v}_i\cdot\Delta\mathbf{x}^k\right)^2
\end{aligned} \tag{23}$$

According to (6), (7), we can deduce that

$$\begin{aligned}
\delta_1 &= -\frac{1}{\eta}\Delta\mathbf{x}^k\cdot\Delta\mathbf{x}^k-\lambda\nabla\left(\left\|f(\mathbf{x})\right\|_2^2\right)\cdot\Delta\mathbf{x}^k \\
&= -\frac{1}{\eta}\left\|\Delta\mathbf{x}^k\right\|^2-2\lambda\sum_{j=1}^{p}x_j^k\Delta\mathrm{x}_j^k
\end{aligned} \tag{24}$$

It follows from (19) and (20) that

$$\delta_1 \le -\frac{1}{\eta}\left\|\Delta\mathbf{x}^k\right\|^2-2\lambda\sum_{j=1}^{p}x_j^k\Delta\mathrm{x}_j^k+A_1\|\,\Delta\mathbf{x}^k\|^2 \tag{25}$$

where $A_1=\frac{1}{2}\tilde{C}^2C_1C_2^2.$

$$\begin{aligned}
\delta_2 &= \left(\mathbf{w}\cdot\psi^k\right)^2\int_0^1(1-t)\tilde{g}''\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^k\right)+t\psi^k\right)dt \\
&\le \|\,\mathbf{w}\|^2\|\,\psi^k\|^2A_2' \\
&\le A_2''\|\,\psi^k\|^2
\end{aligned} \tag{26}$$

where $A_2'=\frac{1}{2}\tilde{C},A_2''=A_2'C_1^2.$

Employing (21), we deduce that

$$\delta_2 \le A_2\|\,\Delta\mathbf{x}^k\|^2 \tag{27}$$

where $A_2=n\tilde{C}^2C_2^2A_2''.$

Indeed

$$\begin{aligned}\delta_3 &= \lambda\sum_{j=1}^{p}\left(\left(x_j^{k+1}\right)^2-\left(x_j^k\right)^2\right)\\ &= 2\lambda\sum_{j=1}^{p}x_j^k\Delta x_j^k+2\lambda\sum_{j=1}^{p}(\Delta x_j^k)^2\\ &= 2\lambda\sum_{j=1}^{p}x_j^k\Delta x_j^k+2\lambda\left\|\Delta \mathrm{x}^k\right\|^2\end{aligned} \tag{28}$$

Let

$$\eta<\frac{1}{A_1+A_2+2\lambda} \tag{29}$$

we have that,

$$\begin{aligned}&E\left(\mathbf{x}^{k+1}\right)-E\left(\mathbf{x}^k\right)=\delta_1+\delta_2+\delta_3\\ &\le-\frac{1}{\eta}\left\|\Delta\mathbf{x}^k\right\|^2-2\lambda\sum_{j=1}^{p}x_j^k\Delta x_j^k+A_1\left\|\Delta\mathbf{x}^k\right\|^2+\\ &A_2\left\|\Delta\mathbf{x}^k\right\|^2+2\lambda\sum_{j=1}^{p}x_j^k\Delta x_j^k+2\lambda\left\|\Delta\mathbf{x}^k\right\|^2\\ &=(-\frac{1}{\eta}+A_1+A_2+2\lambda)\left\|\Delta\mathbf{x}^k\right\|^2<0.\end{aligned} \tag{30}$$

The proof of the monotonicity theorem is completed.

Theorem 2. (**Boundedness**) The iterative sequence of input vectors $\left\{X^k\right\}_{k=0}^{\infty}$ of inverse iterative algorithms with L_2 penalty is uniformly bounded.

Proof. By assumption $(A3)$, let

$$\left\|\mathbf{x}^0\right\|\le M_0 \tag{31}$$

According to (6); (7), we can deduce that

$$\mathbf{x}^1=\mathbf{x}^0-\eta E_{\mathbf{x}^0}\left(\mathbf{x}^0\right) \tag{32}$$

$$\mathbf{x}^1=\mathbf{x}^0-\eta[\tilde{g}'\left(\mathbf{w}\cdot G\left(\mathbf{V}\mathbf{x}^0\right)\right)\sum_{i=1}^{n}w_i g'\left(\mathbf{v}_i\cdot\mathbf{x}^0\right)\mathbf{v}_i+\lambda\nabla\left(\left\|\mathbf{x}\right\|_2^2\right) \tag{33}$$

By (29), let $\eta < \frac{1}{A_1 + A_2 + 2\lambda} < \frac{1}{2\lambda}$, then we have

$$\left\|\mathbf{x}^1\right\| \le \left\|\mathbf{x}^0\right\| + \frac{1}{2\lambda}[\mathrm{C}_4 + 2\lambda M_0] \equiv M_1 \tag{34}$$

where $C_4 = \sup \tilde{g}'\left(\mathbf{w} \cdot G\left(\mathbf{V}\mathbf{x}^0\right)\right) \sup_{t \in R} g'(t) \sup\left\|\mathrm{w}^0\right\| \sup\left\|\mathrm{v}_i^0\right\|$.

In the same way, we can deduce that

$$\left\|\mathbf{x}^2\right\| \le M_1 + (\mathrm{C}_4 + 2\lambda M_1) \equiv M_2 \tag{35}$$

Repeating this procedure, There is constant $M_j (3 \le j \le k)$, such that

$$\left\|\mathbf{x}^j\right\| \le M_j \tag{36}$$

Let. $M = \max\{M_0, M_1, \cdots, M_K\}$, then $\left\|\mathbf{x}^j\right\| \le M$.

Hence,

$$\left\|\mathbf{x}^k\right\| \le M, k = 0, 1, \cdots \tag{37}$$

The iterative sequence of input vectors $E(\mathbf{x}^0)$ is uniformly bounded.

Theorem 3. (**Weak convergence**) Assume that conditions $(A1)$, $(A2)$, $(A3)$ are valid, the error function defined by (3), and the learning rate satisfies (30), for any arbitrary initial value $\mathbf{x}^0 \in \square^p$, $\mathbf{x}^k$ defined by (6), then

$$\lim_{k \to \infty} \left\|E_{\mathbf{x}}\left(\mathbf{x}^k\right)\right\| = 0 \tag{38}$$

Proof. By the results of **Theorem 2**. Let $\alpha = \frac{1}{\eta} - A_4 - \lambda A_3$,

$$\begin{aligned} E\left(\mathbf{x}^{k+1}\right) &\le E\left(\mathbf{x}^k\right) - \alpha\left\|\Delta\mathbf{x}^k\right\|^2 \\ &\le E\left(\mathbf{x}^{k-1}\right) - \alpha\left(\left\|\Delta\mathbf{x}^k\right\|^2 + \left\|\Delta\mathbf{x}^{k-1}\right\|^2\right) \\ &\le \cdots \le E\left(\mathbf{x}^0\right) - \alpha\sum_{l=0}^{k}\left\|\Delta\mathbf{x}^k\right\|^2. \end{aligned} \tag{39}$$

For $E(\mathbf{x}^k) \geq 0, k \in \square$, where $K \in \square^+$, then

$$\alpha \sum_{k=0}^{K} \left\| \Delta \mathbf{x}^k \right\|^2 \leq E\left(\mathbf{x}^0\right). \tag{40}$$

Let $K \to \infty$, we can deduce that

$$\sum_{k=0}^{\infty} \left\| \Delta \mathbf{x}^k \right\|^2 \leq \frac{1}{\alpha} E\left(\mathbf{x}^0\right) < \infty \tag{41}$$

This immediately gives

$$\lim_{k \to \infty} \| \Delta \mathbf{x}^k \| = 0. \tag{42}$$

Combining (6) and (8) results in:

$$\lim_{k \to \infty} \left\| E_{\mathbf{x}}\left(\mathbf{x}^k\right) \right\| = 0. \tag{43}$$

This completes the proof of the weak convergence.

Theorem 4. (**Strong convergence**) If assumptions $(A1)-(A4)$ are valid, there holds the strong convergence: There exists $\mathbf{x}^* \in \Omega_0$ such that

$$\lim_{k \to \infty} \mathbf{x}^k = \mathbf{x}^*. \tag{44}$$

Proof. According to $(A3)$, the sequence $\left\{\mathbf{x}^k\right\}\left(k \in \square\right)$ has a subsequence that is convergent to, say, $\mathbf{x}^*$. Then $\mathbf{x}^{k_i} \to \mathbf{x}^* (k_i \to \infty)$; k_i is a subsequence of k. It follows from (43) and the continuity of $E_{\mathbf{x}}\left(\mathbf{x}\right)$ that

$$\left\| E_{\mathbf{x}}\left(\mathbf{x}^*\right) \right\| = \lim_{i \to \infty} \left\| E_{\mathbf{x}}\left(\mathbf{x}^{k_i}\right) \right\| = \lim_{m \to \infty} \left\| E_{\mathbf{x}}\left(\mathbf{x}^k\right) \right\| = 0. \tag{45}$$

This implies that $\mathbf{x}^*$ is a stationary point of $E\left(\mathbf{x}\right)$. Hence, $\left\{\mathbf{x}^k\right\}$ has at least one accumulation point and every accumulation point must be a stationary point of $E\left(\mathbf{x}\right)$.

Next, by reduction to absurdity, we prove that $\left\{\mathbf{x}^k\right\}$ has precisely one accumulation point. Let us assume to the contrary that $\left\{\mathbf{x}^k\right\}$ has at least two accumulation points, without loss of generality, we assume that the first components of x andex do not equal to each other, that is, $\overline{x}_1 \neq \tilde{x}_1$. For

$\forall \lambda \in (0,1)$, let $x_1' = \lambda \bar{x}_1 + (1-\lambda)\tilde{x}_1$. By **Lemma 2**, there exists a subsequence $\left\{x_1^{k_{i_1}}\right\} \subset \{x_1^k\}$ such that $x_1^{k_{i_1}} \to x_1', (i_1 \to \infty)$, here fki1g is a subsequence of fkg. Due to the boundedness of $\left\{\mathbf{x}^{k_{i_1}}\right\}$, there is a convergent subsequence $\left\{k_{i_2}\right\} \subset \left\{k_{i_1}\right\}$, we define $\mathbf{x}^{k_{i2}} \to \mathbf{x}_2'(i_2 \to \infty)$. Repeating this procedure, we end up with decreasing subsequences $x_t^{k_{i_t}} \to x_t', (i_t \to \infty), t = 1,2,\cdots,p$. Write $x' = \{x_1', x_2', \cdots, x_p'\}$. Then, we see that x' is an accumulation point of $\{\mathbf{x}_k\}$ for any $\lambda \in (0,1)$. But this means that 0 has interior points, which contradicts with the assumption $(A4)$. Thus, $\mathbf{x}^*$ must be a unique accumulation point of $\mathbf{x}^k$. This completes the proof of the strong convergence.

4 Conclusions

An inverse iterative algorithm for neural networks with L_2 penalty has been proposed in this paper. The main contributions of this paper are focused on the theoretical analyses. The monotonicity of the error function and boundedness of inputs have been proved under mild conditions. The gradient sequence of the error function with respect to the inputs tends to zero as the iterations go to infinity, this results in the weak convergence. The strong convergence (the input sequence approaches a fixed point) is then obtained by an additional assumption.

Acknowledgments

The authors wish to thank the anonymous reviewers for careful error proofing of the manuscript and many insightful comments and suggestions which greatly improved this work. This project was supported in part by the National Natural Science Foundation of China (No. 61305075), the China Postdoctoral Science Foundation (No. 2012M520624), Natural Science Foundation of Shandong Province (No. ZR2013FQ004), the Specialized Research Fund for the Doctoral Program of Higher Education of China (No. 20130133120014) and the Fundamental Research Funds for the Central Universities (No. 15CX05053A, 15CX08011A).

References

1. J. M. Zurada, *Introduction to artificial neural systems: West Publishing Company,* St. Paul, 1992.
2. S. S. Haykin, *Neural networks and learning machines: Pearson Education*, Upper Saddle River, 2009.
3. P.Werbos, *Beyond regression: New tools for prediction and analysis in the behavioral sciences,* 1974.
4. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by backpropagating errors,* Cognitive modeling, 1988.
5. G. E. Hinton, *Connectionist learning procedures,* Artificial intelligence, vol. 40, no. 1, pp. 185-234, 1989.
6. R. Reed, *Pruning algorithms-a survey,* Neural Networks, IEEE Transactions on, vol. 4, no. 5, pp. 740-747, 1993.
7. M. Ishikawa, *Structural learning with forgetting,* Neural Networks, vol. 9, no. 3, pp. 509-521, 1996.
8. R. Setiono, *A penalty-function approach for pruning feedforward neural networks,* Neural computation,vol. 9, no. 1, pp. 185-204, 1997.
9. H. M. Shao,W.Wei., and L.-j. Liu, "Convergence of Online Gradient Method with Penalty for BP Neural Networks," Communications in Mathematical Research vol. 26, no. 1, pp. 67-75, 2010.
10. J.Wang, J. Yang, andW.Wu, *Convergence of cyclic and almost-cyclic learning with momentum for feedforward neural networks,* Neural Networks, IEEE Transactions on, vol. 22, no. 8, pp. 1297-1306, 2011.
11. G. Uhlmann, *Inside out: inverse problems and applications*: Cambridge University Press, 2003.
12. M. Zamparo, S. Stramaglia, J. Banavar, and A. Maritan, *Inverse problem for multivariate time series using dynamical latent variables,* Physica A: Statistical Mechanics and its Applications, vol. 391, no. 11, pp. 3159-3169, 2012.
13. J. Kindermann, and A. Linden, *Inversion of neural networks by gradient descent,* Parallel computing,vol. 14, no. 3, pp. 277-286, 1990.
14. A. Fanni, and A. Montisci, *A neural inverse problem approach for optimal design,* Magnetics, IEEE Transactions on, vol. 39, no. 3, pp. 1305-1308, 2003.
15. Y. Hayakawa, and K. Nakajima, *Design of the inverse function delayed neural network for solving combinatorial optimization problems,* Neural Networks, IEEE Transactions on, vol. 21, no. 2, pp. 224-237, 2010.
16. D. Cherubini, A. Fanni, A. Montisci, and P. Testoni, *Inversion of MLP neural networks for direct solution of inverse problems,* Magnetics, IEEE Transactions on, vol. 41, no. 5, pp. 1784-1787, 2005.
17. E. W. Saad, and D. C. Wunsch II, *Neural network explanation using inversion,* Neural Networks,vol. 20, no. 1, pp. 78-93, 2007.

18. R. W. Duren, R. J. Marks, P. D. Reynolds, and M. L. Trumbo, *Real-time neural network inversion on the SRC-6e reconfigurable computer,* Neural Networks, IEEE Transactions on, vol. 18, no. 3, pp. 889-901, 2007.
19. S.-q. Meng, *Convergence of an inverse iteration algorithm for neural networks,* Dalian University of Technology, 2007.
20. Z.-b. Xu, H. Zhang, Y. Wang, X.-y. Chang, and Y. Liang, *L 1/2 regularization,* Science China-Information Sciences, vol. 53, no. 6, pp. 1159-1169, 2010.
21. W.Wu, Q. Fan, J. M. Zurada, J.Wang, D. Yang, and Y. Liu, *Batch gradient method with smoothing L1/2 regularization for training of feedforward neural networks,* Neural Networks, vol. 50, pp. 72-78, 2014.

SOFTWARE SYSTEMS CLUSTERING USING ESTIMATION OF DISTRIBUTION APPROACH

Mahjoubeh Tajgardan, Habib Izadkhah, Shahriar Lotfi

Department of Computer Science, Faculty of Mathematical Science, University of Tabriz, Tabriz, Iran
mahjoubeh_tajgardan@yahoo.com; izadkhah@tabrizu.ac.ir; shahriar_lotfi@tabrizu.ac.ir

Abstract

Software clustering is usually used for program understanding. Since the software clustering is a NP-complete problem, a number of Genetic Algorithms (GAs) are proposed for solving this problem. In literature, there are two well-known GAs for software clustering, namely, Bunch and DAGC, that use the genetic operators such as crossover and mutation to better search the solution space and generating better solutions during genetic algorithm evolutionary process. The major drawbacks of these operators are (1) the difficulty of defining operators, (2) the difficulty of determining the probability rate of these operators, and (3) do not guarantee to maintain building blocks. Estimation of Distribution (EDA) based approaches, by removing crossover and mutation operators and maintaining building blocks, can be used to solve the problems of genetic algorithms. This approach creates the probabilistic models from individuals to generate new population during evolutionary process, aiming to achieve more success in solving the problems. The aim of this paper is to recast EDA for software clustering problems, which can overcome the existing genetic operators' limitations. For achieving this aim, we propose a new distribution probability function and a new EDA based algorithm for software clustering. To the best knowledge of the authors, EDA has not been investigated to solve the software clustering problem. The proposed EDA has been compared with two well-known genetic algorithms on twelve benchmarks. Experimental results show that the proposed approach provides more accurate results, improves the speed of convergence and provides better stability when compared against existing genetic algorithms such as Bunch and DAGC.

Key words: Software System, Clustering, Genetic Algorithm, Estimation of Distribution Algorithm (EDA), Probability Model

1 Introduction

In large software systems, program comprehension is an important factor for its development and maintenance [1]. Clustering is presented as a key activity in reverse engineering to extract software architecture (structure) to improve

understanding of the program [2]. The aim of the software clustering methods is to automatically group the similar artifacts of a software system together into clusters and discover the software structure based on relationships between artifacts in a software system, in which the relationships between the artifacts of different clusters are minimized, and the relationships between the artifacts of the same cluster are maximized (maximum cohesion and minimum coupling) [4]. In general, low coupling and high cohesion are characteristics for well-designed software systems [3]. The first stage in the software clustering is to extract a Call Dependency Graph (CDG) from the program to improve the comprehensibility of complex software systems [4]. CDG is usually used in search-based clustering algorithms for modeling the calls between artifacts. Figure 1 shows a sample of the clustered call dependency graph of a program. In this sample the relationship between artifacts in clusters is high and the coupling between clusters is low (well-designed).

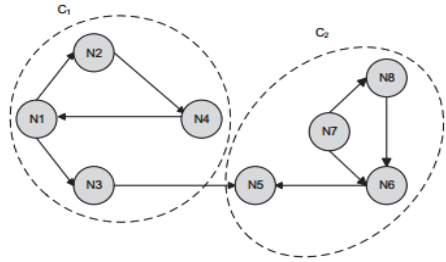

Figure 1. Clustered call dependency graph

Considering huge search space, the problem of finding the best clustering for a software system is a non-deterministic polynomial complete (NP-Complete) problem, hence, the necessity of the use of evolutionary algorithms to achieve a proper clustering is known [4]. Some genetic algorithms are provided in the context of software clustering in which communication and information exchange between individuals is done through the selection and recombination of the individual in a generation. This information movement causes partial solutions to combine with each other, and then higher quality solutions are obtained possibly. With all positive features that the standard genetic algorithm has, the major drawback of this algorithm is that the behavior of genetic algorithm depends on parameters like how to define the crossover and mutation operators and their probabilities, etc. [5]. Therefore, researcher requires experiments in order to choose the suitable values for these parameters [5]. Crossover is a process of taking the pairs of selected parents and producing new offspring from them. The aim of mutation operator is to avoid 'getting stuck' at local optimum points, maintain genetic diversity and discover new areas of the search space. These operators are executed serially. Crossover and mutation operators have a fixed rate of happening (i.e., the operators are applied with a fixed probability) that varies across problems. In

problems that require certain crossover and mutation operators, defining these operators is very difficult and complex, because genetic operators must be defined in a way that can produce valid individuals. For example, in a permutation-based encoding, we need operators that can be maintained as well as the permutation property of individuals [6]. Moreover, in some problems in which the genetic operators do not guarantee the building blocks protection, GA shows a poor performance [5].

In order to overcome the disadvantages of the genetic algorithm, a class of evolutionary algorithms called estimation of distribution algorithms (EDAs) is provided that has advantages in comparison with genetic algorithm. These advantages include [5]: (1) generating new individuals using the probability distribution of all virtualization solutions of previous generation, instead of using the genetic operators; (2) maintaining building blocks in successive generations by giving more chances to partial solutions; and (3) improving the speed of progress towards optimal solution by maintaining building blocks.

In this paper, we recast the estimation of distribution algorithm for software systems clustering, aiming to overcome the genetic algorithm problems and achieving the better clustering by keeping the building blocks during evolutionary process. We propose a probability distribution function to generate a new population using the features of clustering problem, which can solve the problem using genetic operators such as crossover and mutation. The results of our experiments showed that our estimation of distribution algorithm can provide acceptable clustering from the perspective of a domain expert, and as a result contribute to the understanding of software system.

The structure of the rest of this paper is as follows: Section 2 provides some background about EDA and addresses the limitations of the existing works. Section 3 explains the proposed algorithm for software clustering using EDA. Section 4 gives the results of applying our clustering algorithm and some known evolutionary algorithms and discusses on results. Finally, Section 5 concludes the paper.

2 Background and Related Works

In this section, we provide the basic information required for software systems clustering using the estimation of distribution algorithms and some related works in the field of software systems clustering.

2.1 Estimation of distribution algorithms

EDAs are population-based search algorithms based on probabilistic modeling of promising solutions [5]. In EDAs the new population is generated using a probability distribution estimated from the selected individuals of the previous

generation [5]. Figure 2 illustrates the EDA approaches in the optimization process. The EDA follow the following steps in the optimization process:

1. Firstly, the initial individuals of size μ as initial population are generated. The generation of these μ individuals is usually carried out by assuming a uniform distribution on each variable. Then, each individual using fitness function is evaluated.
2. Secondly, λ individuals (where $\lambda \leq \mu$) based on specified criteria are selected for calculating the n–dimensional probabilistic model that better represents the interdependencies. The aim is to calculate joint probability distribution of selected individuals. This step is also known as the learning procedure, and it is the most crucial one, since representing appropriately the dependencies between the variables is essential for a proper evolution towards fitter individuals.
3. New population is generated according to calculated probability distribution.
4. Finally, new population is replaced into previous population.

Steps 2, 3 and 4 are repeated until a stopping condition is verified. Examples of stopping conditions are: achieving a fixed number of populations or a fixed number of different evaluated individuals, uniformity in the generated population, and the fact of not obtaining an individual with a better fitness value after a certain number of generations.

2.2 Software clustering algorithms

Generally, in literature, software clustering algorithms can be generally categorized into the following groups:

1. Clustering algorithms based on concept analysis [7]: In such algorithms, the goal is to classify procedures and variables into clusters. Clustering algorithms of this group are merely used for extracting software architecture form the respective procedural codes and are not conclusive for large systems, as quoted by the author [7].
2. Hierarchical clustering algorithms [8-11]: In these algorithms, each entity is initially considered in a separate cluster, and then; these clusters are gradually combined with each other creating larger clusters. These algorithms provide hierarchical structure from system architecture [8]. The pitfall of hierarchical methods is their failure to benefit from software engineering criteria for determining clusters or code clusters. The hierarchical approaches seem to be useful for program understanding and knowledge discovery, because in general they allow the original problem to be studied at different levels of detail by navigating up and down the hierarchy. However, it is a difficult problem to find the appropriate

height at which to prune a hierarchy of clusters to obtain optimal partitioning.

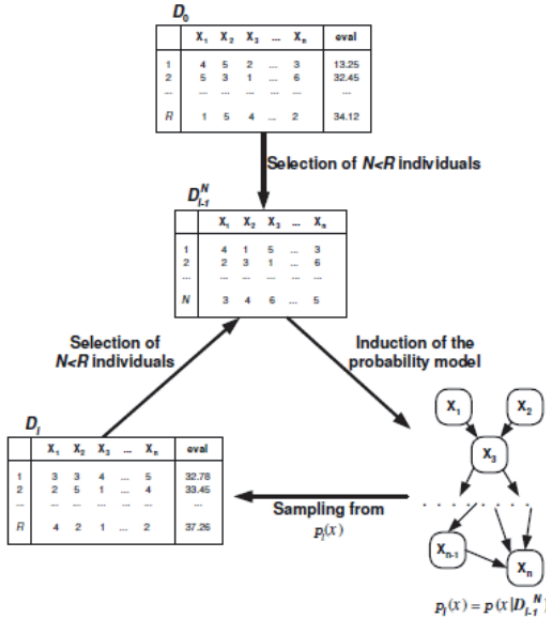

Figure 2. Illustration of EDA approaches in the optimization process

3. Search-based clustering algorithms: clustering problem is treated as a search task in these algorithms. Since searching the complete state space turns the situation into a NP-Complete problem [11], heuristic search techniques such as genetic algorithm are deployed for finding the optimal or near optimal answer during a reasonable time. Search operation is carried out using criteria of maximal cohesion and minimal coupling of clusters. These criteria are particularly suitable in object-based systems for identifying sub-systems or clusters. These methods are divided into two categories: global search (Such as Bunch [12, 13] and DAGC [6]), local search (Such as SAHC [13] and NAHC [13]) and combining local and global search (Such as HC+Bunch [14]) methods. The main drawback of local search methods is that they have the risk of getting stuck in local maximum values, but, global search methods are able to escape from these local maximums [12]. Search-based algorithms have been able to achieve better results than the hierarchical techniques.

Genetic algorithms are widely and effectively used for NP-Hard optimization problems. They can produce acceptably near-optimal answers in reasonable time [6]. Genetic clustering algorithms are very subjective [15]; well-known tools such as DAGC, Bunch use genetic algorithm for clustering soft-

ware systems. In the Bunch [8], each individual is an array that the number of its genes is equal to the number of nodes in the call dependency graph (CDG) and the content of each gene identifies a cluster that contains the corresponding node. In the DAGC [6], each array (individuals) is a permutation of the nodes of N integer. An individual can be decoded into a clustering by the following process: m^{th} cell of the individual represents the node number 'm' of the CDG. Its content includes number of another node of graph like 'p' ($1 \le p \le N$) and if 'p' is equal or greater than 'm', then 'm' is placed in a new cluster otherwise 'm' belongs to the same cluster as 'p'.

Objective function used in Bunch and DAGC and our algorithm is TurboMQ [11, 12]. If the internal edges of cluster and edges between two clusters are respectively represented by μ_i and $\varepsilon_{i,j}$, TurboMQ value will be then computed as follows:

$$CF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum_{j=1}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & otherwise \end{cases} \quad (1)$$

$$TurboMQ = \sum_{i=1}^{k} CF_i \quad (2)$$

3 The proposed algorithm

Search-based software clustering methods such as the Bunch and DAGC are clearly superior to the hierarchical methods. However, they have a particular drawback that it was explained before. We try to address it by introducing a new algorithm in this section. This new algorithm is based on probabilistic model and does not use genetic operators such as crossover and mutation, instead keep the building blocks. In other words, we present a probabilistic model to generate a new population. This section explains our proposed probabilistic model (subsection 3.1) and software clustering using EDA (subsection 3.2).

3.1 Probabilistic model

To obtain the probability model, let M be an n×n square matrix (where n is the number of software artifacts), and initially the values of all elements except the main diagonal are 1/n. The value of M[i, j] represents the probability that two artifacts will be placed in the same cluster on individuals in the next generation. For example, if we have software system containing 5 artifacts, initially, the probability will be as shown in Table (1).

Table 1. The initial probability matrix

	F1	F2	F3	F4	F5
F1	0	0.2	0.2	0.2	0.2
F2	0.2	0	0.2	0.2	0.2
F3	0.2	0.2	0	0.2	0.2
F4	0.2	0.2	0.2	0	0.2
F5	0.2	0.2	0.2	0.2	0

$$M[i,j] = \frac{1}{n} \qquad i,j = 1,2,\dots,n \tag{3}$$

Then, we find the best and worst individuals of the population in each generation and change the values of probability matrix as follows:

1. If two artifacts i and j are placed in the same cluster in the best individual but are not placed in the same cluster in the worst individual (suppose t is the number of iterations for the evolutionary algorithm):

$$\text{if } M[i,j] < 1 - \frac{5}{t} \qquad M[i,j] = M[i,j] + \frac{5}{t} \tag{4}$$

2. If two artifacts i and j are placed in the same cluster in the worst individual but are not placed in the same cluster in the best individual:

$$\text{if } M[i,j] > \frac{5}{t} \qquad M[i,j] = M[i,j] - \frac{5}{t} \tag{5}$$

3. If two artifacts i and j are placed in the same cluster in the best and the worst individuals, the value of probability in the probability matrix does not change.

After changing the probability model, the new population is produced using the new possibilities. For example, Suppose in the Table (2), (a) and (b) are the best and the worst individuals respectively, then new probabilities assuming t=100 are given in Table (3).

It is obvious that in this model the probability of any two artifacts is not equal to 0 and 1. We consider this condition for maintaining the diversity of our population and preventing premature convergence.

Table 2. The best and worst individual

	F1	F2	F3	F4	F5
(a)	3	3	1	1	2
(b)	3	1	2	2	3

Table 3. The new probability matrix

	F1	F2	F3	F4	F5
F1	0	0.25	0.2	0.2	0.2
F2	0.25	0	0.2	0.2	0.2
F3	0.2	0.2	0	0.2	0.2
F4	0.2	0.2	0.2	0	0.2
F5	0.2	0.2	0.2	0.2	0

3.2 Clustering using EDA

In our proposed algorithm each solution is shown as an individual. To represent individuals, we use Bunch algorithm's chromosome representation [11, 12], but with limited number of clusters. In the Bunch, each individual is an array that the number of its genes is equal to the number of nodes in the call dependency graph (CDG) and the content of each gene identifies a cluster that contains the corresponding node and its numeric value is between one to N that N is the number of nodes in the CDG. Formally, an encoding on a string S is defined as:

$$S = s1\ s2\ s3\ s4\ \ldots\ sN \tag{6}$$

Where, N is the number of nodes in the CDG and s_i identifiers the cluster that contains the i^{th} node of the graph. For example, the graph in Figure 3 is encoded as the following string S:

S = 2 2 3 3 1 1 1

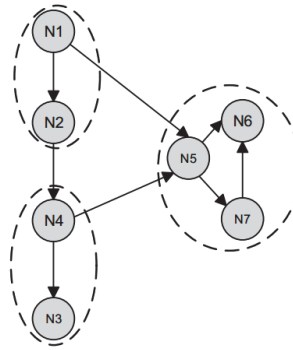

Figure 3. A sample clustering

In contrast with existing genetic based algorithm for software clustering, in our method, we use the probability model instead of the using crossover and mutation. In the proposed algorithm, first an initial population of individuals is generated randomly and the individuals are evaluated using TurboMQ fit-

ness function. Then until the termination condition is established, individuals are selected using the selection operator and then offspring is generated according to calculated probability distribution. The previous population is replaced by the new population. Figure 4 shows two first generation of our proposed algorithm. In this figure initial population indicate the number of individuals and the corresponding fitness.

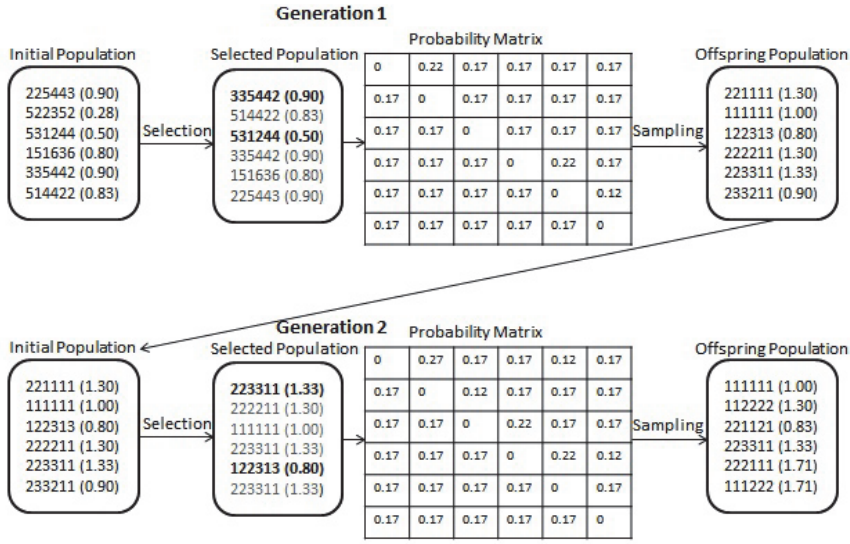

Figure 4. Two first generation of our proposed algorithm

We use the top of triangle probability matrix for the generation of each individual in new population as Algorithm 1. In any iteration, the old population is replaced by new produced population. Briefly, the EDAs process is as Algorithm 2.

Algorithm 1: New population generation algorithm

- Until for each artifact, the cluster is not determined, the following steps are repeated:
  1. Select a row from the probability matrix, randomly.
  2. For each two artifacts generate a random number from [0, 1]. If the value of the random number is smaller or equal to their probability value, two artifacts are placed in the same cluster.

Algorithm 2: EDA based software clustering algorithm

BEGIN
Generate initial population of size μ, randomly.
Evaluate each individual using TurboMQ (Eq. 2)

While (a fixed number of generation is not achieved)

1. Select λ=2 promising and worse individuals (where $\lambda \leq \mu$);
2. Calculate the probability distribution matrix using selected individuals (Section 3.1);
3. Generate new population according to calculated probability distribution;
4. Evaluate each new offspring using TurboMQ;
5. Replace offspring into main population;

END

4 Experimental Results

In this section, we compare the results obtained by proposed EDA and five well-known algorithms such as Bunch, DAGC, NAHC (Next Ascent Hill Climbing), SAHC (Steepest Ascent Hill Climbing), HC+Bunch. For evaluating the obtained clustering, internal and external metrics are used. External one is used to compare results of obtained clustering algorithm by the clustering provided by a domain expert. In fact, the external metrics are used for assessing the reliability of an algorithm. Mojo [16], edgeMojo [17], Precision/Recall [11], and F_m as harmonic mean of Precision/Recall are of external metrics. Mtunis is an academic operating system and since the clustering of this operating system is available so we've used it to evaluate the reliability of proposed algorithm. When the clustering produced by the expert is not available, internal metrics can be used. Table 4 shows the comparison of the proposed algorithm with some existing clustering algorithms. What is clear in this table is that proposed algorithm is able to provide clustering similar to clustering of an expert (When amount of Mojo, edgeMojo is lower, it represents more similarity between clustering produced algorithm with the one produced by an expert, while the larger F_m indicates more similarity).

In Table 5, the proposed algorithm is compared with known evolutionary algorithms on twelve benchmarks in terms of TurboMQ and the average value in twenty runs. In all these cases, it's obvious that the proposed algorithm was able to separate the clusters equal or better than Bunch and DAGC. The results of the EDA are equal to Bunch in seven and two cases and better than Bunch in five and nine cases in terms of TurboMQ and average, respectively. These results are also equal to DAGC in two cases, better than DAGC in ten cases in terms of TurboMQ and better than DAGC in terms of average in all cases.

Table 4. Comparisons of proposed algorithm with two well-known GA

	Mojo	Edge Mojo	F_m
Bunch	**5**	**7.47**	**0.57**
DAGC	**7**	**10.33**	**0.48**
HC+Bunch	**9**	**11.14**	**0.25**
NAHC	**5**	**13.14**	**0.53**
SAHC	**5**	**10.81**	**0.55**
EDA	**5**	**7.47**	**0.57**

Table 5. Comparisons of proposed algorithm with two well-known GA

	BUNCH			DAGC			EDA		
Software systems	**# of clusters**	**TurboMQ**	**Average**	**# of clusters**	**TurboMQ**	**Average**	**# of clusters**	**TurboMQ**	**Average**
compiler	**4**	**1.506**	**1.506**	**4**	**1.506**	**1.455**	**4**	**1.506**	**1.506**
boxer	**7**	**3.101**	**3.101**	**7**	**3.101**	**2.910**	**7**	**3.101**	**3.091**
mtunis	**5**	**2.314**	**2.286**	**6**	**2.241**	**2.048**	**5**	**2.314**	**2.314**
ispell	**7**	**2.177**	**2.140**	**8**	**1.997**	**1.872**	**6**	**2. 190**	**2.180**
bison	**13**	**2.606**	**2.539**	**15**	**1.763**	**1.679**	**12**	**2.664**	**2.633**
cia	**14**	**2.706**	**2.627**	**19**	**1.833**	**1.691**	**12**	**2.787**	**2.740**
ciald	**8**	**2.851**	**2.834**	**12**	**2.463**	**2.275**	**8**	**2.851**	**2.849**
modulizer	**7**	**2.648**	**2.608**	**9**	**2.112**	**1.915**	**7**	**2.648**	**2.628**
nos	**5**	**1.636**	**1.625**	**5**	**1.606**	**1.508**	**5**	**1.636**	**1.635**
rcs	**9**	**2.175**	**2.115**	**11**	**1.894**	**1.766**	**8**	**2.194**	**2.161**
spdb	**6**	**5.741**	**5.741**	**8**	**5.314**	**5.076**	**6**	**5.741**	**5.741**
star	**10**	**3.809**	**3.673**	**16**	**2.831**	**2.524**	**9**	**3. 832**	**3.766**

In Table 6, the speed of convergence in proposed algorithm and Bunch is compared. We run the algorithms ten times and considered 1000 for number of iterations in each run. In cases that the obtained TurboMQ by our algorithm is equal to Bunch, the advantage of our method is that the speed of convergence to the solution is more and algorithm finds the solution in lower reps; for illustration Figure 5 and Figure 6 show convergence diagram of compiler benchmark for Bunch and EOD respectively. Obviously, EOD is converged in lower number of iterations.

Table 6. Comparisons of proposed algorithm with Bunch in terms of the average of iterations for the convergence to the solution

	compiler	**spdb**	**boxer**	**mtunis**	**ciald**	**modulizer**	**nos**
BUNCH	258	325	270	396	577	505	345
EOD	70	98	114	141	314	329	113

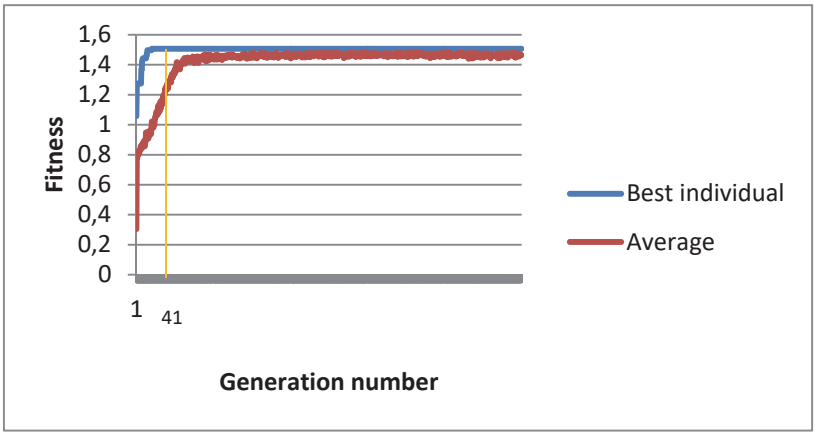

Figure 5. Convergence diagram (compiler) for EDA

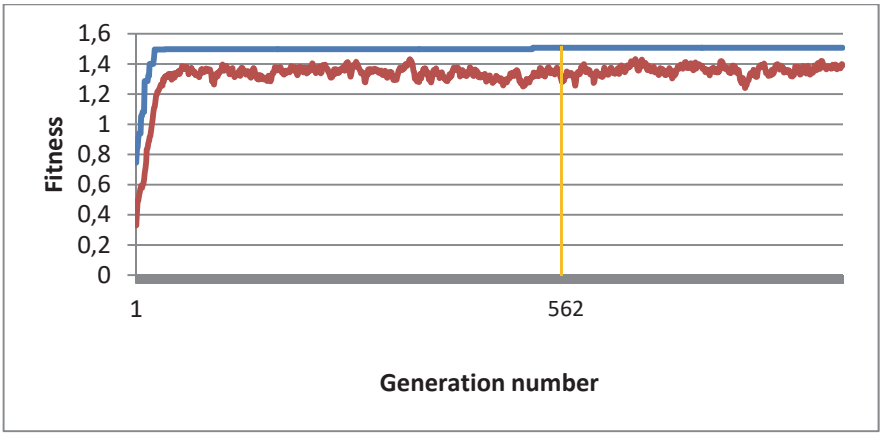

Figure 6. Convergence diagram (compiler) for Bunch

In Table 7, our algorithm is compared with Bunch and DAGC in terms of the standard deviation of the results of 20 runs (the lower Std. Deviation indicates more stability). The results of this table show that stability of proposed algorithm is higher than Bunch and DAGC in most and all cases, respectively. So, we can say our algorithm has higher stability. For example, stability diagram of proposed algorithm for one of the benchmarks (compiler) is presented in Figure 7.

Table 7. Comparisons of proposed algorithm with two well-known GA in terms of Std. Deviation

	Std. Deviation											
Software systems	compiler	boxer	mtunis	ispell	bison	cia	modulizer	nos	rcs	spdb	star	ciald
EDA	0.002	0.018	0	0.009	0.024	0.034	0.038	0.002	0.028	0	0.075	0.003
BUNCH	0	0	0.028	0.023	0.047	0.053	0.039	0.015	0.030	0	0.080	0.010
DAGC	0.042	0.102	0.096	0.073	0.076	0.064	0.104	0.058	0.109	0.187	0.149	0.106

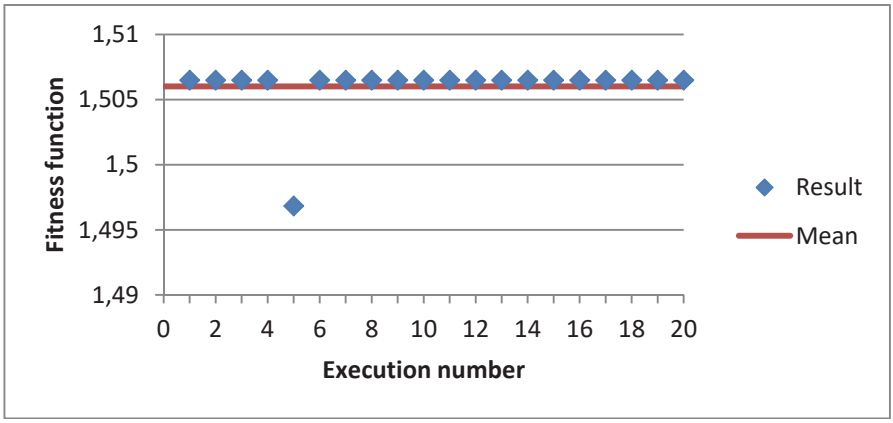

Figure 7. Stability diagram

5 Conclusion

In this paper, we have used estimation of distribution algorithm for software clustering problem. A probability model was presented using features of clustering problem. Results of initial tests showed that the proposed algorithm is very promising. For future work we are planning to do the following work:

1. In future work, we will try to test our algorithm on many software systems.
2. One of the important issues related to Bunch encoded is largeness of search space due to presence of some repetitive answers, i.e., although some generated encodes have apparently different representations, but in reality, they represent the same clustering. For example, though two chromosomes S_1= 2 2 4 4 1 and S_2=1 1 5 5 3 have different appearances but they are actually representative of the same clustering. Because, in both, there are three clusters so that nodes of C_1 and C_2 are in same cluster, nodes of C_3 and C_4 are in same cluster and node node C_5, located in distinct cluster. Search space in Bunch algorithm is n^n; this large search space decelerates speed of this algorithm to find appropriate structure. The state space of n^n is the worst state for a problem and search in this space is impossible in a rational time. Such state space would cause doubt in finding a

good structure for software by Bunch. We believe that we can reduce it using limited number of clusters. It is well known fact that the number of clusters are much less than the number of classes in a program. Considering the number of classes as *n*, if we limit the number of clusters to maximum n/3 of classes (it is usually much less than n/3.); therefore, the state space of Bunch can be reduced to $(\frac{n}{3})^n$. The upper bound of this state space is O(n!). This significant reduction may have a significant effect on improvement of the quality of achieved structure.

References

1. Zhang Q., Qiu D., Tian Q., Sun L., 2010, *Object-oriented software architecture recovery using a new hybrid clustering algorithm*. Fuzzy Systems and Knowledge Discovery (FSKD), Seventh International Conference on. Vol. 6. IEEE, 2010.
2. Bittencourt R. A., and Dalton D. G., 2009, *Comparison of graph clustering algorithms for recovering software architecture module views*. Software Maintenance and Reengineering, CSMR'09. 13th European Conference on. IEEE, 2009.
3. Poshyvanyk D., Marcus A., Ferenc R., *Using information retrieval based coupling measures for impact analysis*. Empirical software engineering 14.1: 5-32, 2009
4. Izadkhah H., Elgedawy I., and Isazadeh A., *E-CDGM: An Evolutionary Call-Dependency Graph Modularization Approach for Software Systems*. Cybernetics and Information Technologies 16.3: 70-90, 2016.
5. Larranaga P., and Lozano J., *Estimation of distribution algorithms: A new tool for evolutionary computation*. Vol. 2. Springer Science & Business Media, 2002.
6. Parsa S., and Bushehrian O., *A new encoding scheme and a framework to investigate genetic clustering algorithms*. Journal of Research and Practice in Information Technology 37.1: 127, 2005.
7. Lindig C., and Snelting G., *Assessing Modular Structure of Legacy Code based on Mathematical Concept Analysis*. Proceedings of the International Conference on Software Engineering, 1997.
8. Lindig C., and Snelting G., *Assessing Modular Structure of Legacy Code based on Mathematical Concept Analysis*. Proceedings of the International Conference on Software Engineering, 1997.
9. Cui J. F., and Chae H. S., *Applying Agglomerative Hierarchical Clustering Algorithms to Component Identification for Legacy Systems*. Information and Software Technology, Volume 53, Issue 6, Pages 601-614, 2011.

10. Maqbool O., and Babri H., *Hierarchical clustering for software architecture recovery*. IEEE Transactions on Software Engineering 33.11: 759-780, 2007.
11. Andritsos P., and Tzerpos V., *Information-theoretic software clustering*. IEEE Transactions on Software Engineering 31.2: 150-165, 2005.
12. Mitchell Brian S., *A heuristic search approach to solving the software clustering problem*. Diss. Drexel University, 2002.
13. Mitchell, Brian S., and Mancoridis S., *On the automatic modularization of software systems using the bunch tool*. IEEE Transactions on Software Engineering 32.3: 193-208, 2006.
14. Mahdavi K., Harman M., and Hierons R. M., *A multiple hill climbing approach to software module clustering*. *Software Maintenance, ICSM 2003. Proceedings. International Conference on*. IEEE, 2003.
15. Praditwong K., Harman M., and Yao X., *Software module clustering as a multi-objective search problem*. *IEEE Transactions on Software Engineering* 37.2: 264-282, 2011.
16. Tzerpos V., and Holt R. C., MoJo: *A distance metric for software clusterings*. Reverse Engineering, Proceedings. Sixth Working Conference on. IEEE, 1999.
17. Wen Z., and Tzerpos V., An *effectiveness measure for software clustering algorithms*. Program Comprehension, Proceedings. 12th IEEE International Workshop on. IEEE, 2004.

Low-cost Dynamic Constraint Checking for the JVM

Konrad Grzanek

IT Institute, University of Social Sciences
9 Sienkiewicza St., 90-113 Lodz, Poland
kgrzanek@spoleczna.pl

Abstract

Using formal methods for software verification slowly becomes a standard in the industry. Overall it is a good idea to integrate as many checks as possible with the programming language. This is a major cause of the apparent success of strong typing in software, either performed on the compile time or dynamically, on run-time. Unfortunately, only some of the properties of software may be expressed in the type system of event the most sophisticated programming languages. Many of them must be performed dynamically. This paper presents a flexible library for the dynamically typed, functional programming language running in the JVM environment. This library offers its users a close to zero run-time overhead and strong mathematical background in category theory.

Keywords: Formal software verification, software quality, dynamic type-checking, functional programming, category theory, Clojure

1 Introduction

Despite an apparent progress in programming languages theory and practice, the IT industry still experiences problems achieving a desired level of software quality and reliability. M. Thomas [2] mentions that there are from 4 up to 50 bugs (on average) in every 1 thousand lines of production code. This is why the computers are still problematic to rely on in the (not only) safety and mission critical areas of life [1], and the non-critical software is usually hard to use, has got a lowered level of security due to hidden bugs that may even not exhibit themselves on a regular usage basis, but may be exploited by the ones who search for vulnerabilities with an intention to steal information or to introduce other kinds of costly confusion.

This paper is a result of some real-life, production-related experiences and following considerations regarding when and how to perform constraints checks and other kinds

of formal software verification in a dynamically-typed programming language for the *Java Virtual Machine* (*JVM*) environment. We argue that it is a reasonable decision to imply lots of these checks on the time of the program's execution. We also provide a library called *ch* that allows to define and perform some run-time constraint checks. This article may be treated as a good introduction to how this library is implemented and how it can (and should) be used.

1.1 Reasoning About Software Correctness

Scale of contemporary software systems together with the fact that it is intended to run in a multi-tasking environment makes the formal verification methods a strong requirement not an option now. Growing popularity of tools like TLA+ (L. Lamport [5]), temporal logic, and even the hand proofs in software creation process confirm the growing need to become more and more dependent on the beauty of mathematical verification of software systems.

It is a commonly accepted truth that type systems in programming languages are a very strong point in the formal verification of software. Advances in type theory and practice [3, 4] have led to development of programming languages that are particularly effective in catching a variety of common bugs. Let us mention Haskell [9] or Ada here. Lamport [6] says:

„Types have become ubiquitous in computer science. [...] Types do more good than harm in a programming language: they let the compiler catch errors that would otherwise be found only after hours of debugging."

Strong typing means that the expressions in a language contain no implicit data conversions (coercions) that could lead to an unintentional loss of information. *Static type system* is the one in which the compiler (more generally speaking - a static verification tool, the one that reads and analyzes source code) is responsible for making a proof (or dis-proof) that a computer program does not violate any of the type-related invariants that are amenable to pre-run analysis. On the other side, the *dynamically typed languages* are these that leave at least some of the verifications of invariants for the run-time. We should also be aware that there are type-based invariants that may be verified neither statically, nor dynamically. For example, it is impossible to prove the correctness of a famous quick-sort algorithm using a type checker of any kind (for a discussion see [22]). In such cases using a formal method like the TLA+ ahead of the implementation phase alone to prove correctness, or even providing a hand proof is priceless. In any other scenario relying on the type system is a good idea. Typing the specification languages is a completely different problem - for more on that subject, please see [6].

1.2 Discussion on Static and Dynamic Type-Checking

Using a programming language with a static type checker seems a robust method of eliminating bugs in software. When considering only the type-related aspects of a language this statement leads to an immediate conclusion that the dynamic type checking should be avoided at all cost. But the language is much more that that, and there are

multiple reasons why the dynamically typed programming languages have made such a great success in the industry. Among the others, the dynamic languages:

- Allow to perform immediate tests of the functionality being implemented by using *REPL* (*Read-Eval-Print Loop*).
- In the case of languages from the Lisp family, like Clojure [7, 8] it is even possible to compile new functionality without stopping the running program. The REPL execution takes part in the same environment in which the production software runs, we have no debug-release cycle here, and that feature alone is a great productivity booster.
- A lack of static type system allows to define programming constructs that are intended to be run in contexts that would be very hard or even impossible to describe using a set of static type-related constraints. Some Lisp *macros* are good example here.

Even when we decide to rely on a static type checker we should be aware that the extent to which we will be able to verify programs using this tool is limited to what can be expressed in the rules of the type system, and this extent has bounds. Other checks/verifications/proofs must be performed anyway either on the run-time or by hand proving. Static type systems generally verify that the elements of a system fit together as a structure, and only sometimes can prove or disprove their homeostasis and well-functioning over a period of time.

1.3 Functional Programming

Another means of establishing high level of software quality and reliability is using functional programming languages like Haskell, Clojure, Erlang. The impact of the immutability of data structures on software correctness, its predictability and a relative ease of searching for bugs may be at least as big as using the type checker to find the mistakes statically. Out of the mentioned three languages two are dynamically typed (Erlang, Clojure) and they are highly successful even in mission critical domains.

1.4 Existing Solutions for Clojure

Clojure programming language [7, 8] is one of the most interesting contemporary JVM-targeted languages. It belongs to Lisp family, and - as its elders - is dynamically typed. This section presents current approaches that exist in this language, and that are related to the problems formulated above.

One attempt to employ some kind of static type checking is *clojure/core.typed* library [20]. This solution uses type annotations and a static type checker. Unfortunately, the realization has severe performance problems as described in the discussion [21]. Moreover, due to the reasons described in the paragraphs above a full static type checker does not make a perfect fit with respect to the pragmatics of using Lisps and to the overall idea of implementing software quickly.

Another similar library that suffers similar problems is *prismatic/schema* [19]. In the case of this solution, we find it better in terms of the overall usability and performance, but the type annotations a kind of get in the way with the normal ways of using Lisp, that is - writing as much as possible using *s-expressions*.

Finally, the most promising project in this domain is Cognitect's *clojure.spec* [17, 18]. It may become a de-facto standard specification tool, but its goals are slightly different than the ones we are looking at. Its development process is in its early stages, as the adoption in the industry.

These considerations led us to the idea of creating a custom solution - Clojure library meeting the requirements of dynamic specification/type/invariants validator, with the following assumptions:

- being deeply rooted in functional programming [9] and using notions from the category theory [10],
- consistency with the Lisp nature of Clojure programming language, by using s-expressions only (Lisp as a „big ball of mud"),
- relying on fast dynamic type-checking routines of the JVM - the *instanceof* operator,

Our solution is called *kongra/ch* [12, 13], shortly *ch*, and it has been successfully used in *kongra/prelude* [14, 15] and *aptell* [16] projects. The following sections of the paper are detailed description of its implementation and possible use.

2 Essentials of the ch Library

Every predicate check uses the following procedure to generate a message describing the value, together with its type, that violates the check:

```
(defn chmsg
  [x]
  (with-out-str
    (print "Illegal value ")    (pr x)
    (print " of type ") (pr (class x))))
```

When executed in the REPL the procedure works as follows:

```
user> (chmsg 123)
"Illegal value 123 of type java.lang.Long"
```

or, for nil values:

```
user> (chmsg nil)
"Illegal value nil of type nil"
```

The most essential syntactic structure in the *ch* library is a $(ch...)$ form. Due to the implementation issues the form is intended to be used both as an assertion and as a boolean-valued function. The definition begins with a supporting function *pred-call-form*:

```
(defn- pred-call-form
  ([form x]
   (let [form (if (symbol? form) (vector form) form)]
     (seq (conj (vec form) x))))

  ([form _ x]
   (let [form (if (symbol? form) (vector form) form)]
     (concat form (list nil x)))))
```

and the actual *ch* macro that uses the function to generate a target s-expression, either an assertion or a predicate-like call:

```
(defmacro ch {:style/indent 1}
  ([pred x]
   (let [x'    (gensym "x__")
         form (pred-call-form pred x')]
     `(let [~x' ~x] (assert ~form (chmsg ~x')) ~x')))

  ([pred #_ be-pred _ x]
   (let [form (pred-call-form pred x)]
     `(boolean ~form))))
```

To see, what actually happens when a compiler encounters the (*ch...*) form, please take a look at the expression *(ch nil? 123)*. This expression evaluates the function *nil?* belonging to the Clojure standard library against the argument, integral (Long) value 123. The target form is

```
(let [x__11622 123]
  (assert (nil? x__11622) (chmsg x__11622))
  x__11622)
```

The compiler introduces an additional local variable x__11622 that holds the value of an evaluated 123 input value (expression from compiler's point of view) and executes the (*assert...*) on it using the passed *nil?* function an assert's predicate. After a successful evaluation the evaluated value of x__11622 is returned resulting in the desired behavior. This time it's failure, because 123 is not a nil value:

```
user> (ch nil? 123)
AssertionError Assert failed: Illegal value 123 of type
    java.lang.Long
(nil? x__10994)  kongra.ch/eval10995
    (form-init3881948826253525319.clj:17)
```

If we decide to use (*ch...*) using its predicate "nature", like *(ch nil? :as-pred 123)*, we get:

```
(boolean (nil? 123))
```

and the evaluation of the form ends with *false* value being returned.

Now, let's talk about the performance. All the following performance benchmarks were taken in the commodity hardware environment: Intel i7-5500U, 16 GB of RAM, Ubuntu 14.04 64-bit using the awesome *criterium*[1] library for benchmarking Clojure codes. Additionally we have: CIDER 0.14.0 (Berlin), nREPL 0.2.12, Clojure 1.8.0, Java 1.8.0_121. At first a simple expression:

```
user> (quick-bench (nil? 123))
Evaluation count : 52064214 in 6 samples of 8677369 calls.
             Execution time mean : 1,113992 ns
    Execution time std-deviation : 0,041270 ns
   Execution time lower quantile : 1,058363 ns ( 2,5%)
```

[1] https://github.com/hugoduncan/criterium

```
   Execution time upper quantile : 1,160112 ns (97,5%)
                   Overhead used : 10,475943 ns
```

and the corresponding predicate form of (*ch...*):

```
user> (quick-bench (ch nil? :as-pred 123))
Evaluation count : 52235052 in 6 samples of 8705842 calls.
             Execution time mean : 1,078551 ns
    Execution time std-deviation : 0,115450 ns
   Execution time lower quantile : 0,942867 ns ( 2,5%)
   Execution time upper quantile : 1,215710 ns (97,5%)
                   Overhead used : 10,475943 ns
```

Also for the assertion:

```
user> (quick-bench (ch nil? nil))
Evaluation count : 52466976 in 6 samples of 8744496 calls.
             Execution time mean : 1,081879 ns
    Execution time std-deviation : 0,135884 ns
   Execution time lower quantile : 0,977794 ns ( 2,5%)
   Execution time upper quantile : 1,262656 ns (97,5%)
                   Overhead used : 10,475943 ns
```

We can clearly see that there is no apparent overhead of the call. This is only an introductory example, so it is impossible to reason now about the target performance loss when applying this approach to a production software. This will be discussed in further parts of the paper.

3 Generator of Ch(eck)s

The library would be far from being useful, if the user would be forced to use raw (*ch...*) forms everywhere. Instead, we introduce the *(defch . . .)* macro that allows the programmer to define his own named checks. The macro code goes as follows:

```
(defmacro defch {:style/indent 1}
  ([chname form]
   (let [x     (gensym "x__")
         form+ (append-arg form x) ]
     `(defch ~chname [~x] ~form+)))

  ([chname args form]
   (assert (vector? args))
   (let [args+ (insert-noparam args)
         form+ (insert-noarg   form)]
     `(defmacro ~chname {:style/indent 1}
        (~args  ~form)
        (~args+ ~form+)))))
```

To do its job, the macro performs some manipulations with the arguments and the shape of the target form, which is a subsequent macro in this case. So we may say that *defch* is a macro-writing macro.

The arguments must be enhanced do support a non-assertion (predicate) use, because - as in the case of raw *ch* - we tend to operate both in assertion and in predicate mode. The arguments of the target macro are prepared using the following *insert-noparam*:

```
(defn- insert-noparam
  [params]
  (vec (concat (butlast params)
               (list '_)
               (when (seq params) (list (last params))))))
```

while the two following procedures prepare the predicate target form:

```
(defn- insert-noarg
  [form]
  (let [;; lein eastwood passes a wrapper (sequence <form>),
        ;; let's strip it down:
        form (if (= (first form) 'sequence) (second form) form)
        [ccseq [cconcat & cclists]] form]
    (assert (=  ccseq        'seq) (str "Illegal ccseq "
       ccseq " in " form))
    (assert (=  cconcat   'concat) (str "Illegal cconcat "
       cconcat " in " form))
    (assert (>= (count cclists) 2) (str "Illegal cclists "
       cclists " in " form))
    (let [lsts   (butlast  cclists)
          lst    (last     cclists)
          noarg '(list        'nil)]
      '(seq (concat ~@lsts ~noarg ~lst)))))

(defn- append-arg
  [form x]
  (let [;; lein eastwood passes a wrapper (sequence <form>),
        ;; let's strip it down:
        form (if (= (first form) 'sequence) (second form) form)
        [ccseq [cconcat & cclists]] form]
    (assert (=  ccseq        'seq) (str "Illegal ccseq "
       ccseq " in " form))
    (assert (=  cconcat   'concat) (str "Illegal cconcat "
       cconcat " in " form))
    (assert (>= (count cclists) 2) (str "Illegal cclists "
       cclists " in " form))
    (let [arg '(list ~x)]
      '(seq (concat ~@cclists ~arg)))))
```

Finally we may take a look at how this entirety works together.

4 Unit Type Ch(eck)

Unit type is a very useful type in many programming languages. Unit type has exactly one value. It is so called *terminal object* in category of types and typed functions. In

some programming languages (e.g. Haskell) the unit type is expressed as (), while in others (like C/C++/Java) a *void* keyword is used to express something related, namely the fact that the procedure does not return any value. The latter approach may be somewhat informally referred to as a means to express a lack of information at the output of a procedure, and this follows the original nature of the unit type in category theory - a type with only one object carries on no information.

In Clojure, as in Java, we traditionally use nil as a representation of unit type. The check for nil-ness is defined as follows:

```
;; UNIT (NIL)
(defch chUnit [x] `(ch nil? ~x))
```

This form introduces a macro named *chUnit* that my be used in the following two ways:

```
user> (chUnit 1)
AssertionError Assert failed: Illegal value 1 of type
    java.lang.Long
(clojure.core/nil? x__11646)  kongra.ch/eval11647
    (form-init3881948826253525319.clj:83)
```

or

```
user> (chUnit :as-pred 1)
false
```

Additionally we introduce complementary ch(eck)s for non-nil values:

```
;; NON-UNIT (NOT-NIL)
(defn not-nil? {:inline (fn [x] `(if (nil? ~x) false true))}
                            [x]  (if (nil?  x) false true))
(defch chSome [x] `(ch not-nil? ~x))
```

Their cost is as abysmal as for nil? check, as presented in one of the previous sections. With the following definitions of test procedures:

```
(defn foo [x] (chUnit x))
(defn goo [x] (chSome x))
```

we have:

```
user> (quick-bench (foo nil))
Evaluation count : 35042178 in 6 samples of 5840363 calls.
             Execution time mean : 6,895234 ns
    Execution time std-deviation : 0,280141 ns
   Execution time lower quantile : 6,617001 ns ( 2,5%)
   Execution time upper quantile : 7,299458 ns (97,5%)
                   Overhead used : 10,475943 ns
```

and:

```
user> (quick-bench (goo 123))
Evaluation count : 34250304 in 6 samples of 5708384 calls.
             Execution time mean : 7,283686 ns
    Execution time std-deviation : 0,130028 ns
   Execution time lower quantile : 7,138009 ns ( 2,5%)
```

```
  Execution time upper quantile : 7,444272 ns (97,5%)
                  Overhead used : 10,475943 ns
```

To look more deeply in what happens under the hood in these test procedures, let's use *no.disassemble*[2] library to view the resulting byte-code for *foo*:

```
// Method descriptor #11 (Ljava/lang/Object;)Ljava/lang/Object;
 // Stack: 8, Locals: 2
 public static java.lang.Object invokeStatic(java.lang.Object
     x);
    0  aload_0 [x]
    1  aconst_null
    2  astore_0 [x]
    3  astore_1 [x__16079]
    4  aload_1 [x__16079]
    5  aconst_null
    6  invokestatic
        clojure.lang.Util.identical(java.lang.Object,
        java.lang.Object) : boolean [17]
    9  ifeq 18
   12  aconst_null
   13  pop
   14  goto 79
   17  pop
   18  new java.lang.AssertionError [19]
   21  dup
   22  getstatic kongra.ch$foo.const__1 : clojure.lang.Var [23]
   25  invokevirtual clojure.lang.Var.getRawRoot() :
       java.lang.Object [29]
   28  checkcast clojure.lang.IFn [31]
   31  ldc <String "Assert failed: "> [33]
   33  getstatic kongra.ch$foo.const__2 : clojure.lang.Var [36]
   36  invokevirtual clojure.lang.Var.getRawRoot() :
       java.lang.Object [29]
   39  checkcast clojure.lang.IFn [31]
   42  aload_1 [x__16079]
   43  invokeinterface
       clojure.lang.IFn.invoke(java.lang.Object) :
       java.lang.Object [39] [nargs: 2]
   48  ldc <String "\n"> [41]
   50  getstatic kongra.ch$foo.const__3 : clojure.lang.Var [44]
   53  invokevirtual clojure.lang.Var.getRawRoot() :
       java.lang.Object [29]
   56  checkcast clojure.lang.IFn [31]
   59  getstatic kongra.ch$foo.const__4 : java.lang.Object [48]
   62  invokeinterface
       clojure.lang.IFn.invoke(java.lang.Object) :
       java.lang.Object [39] [nargs: 2]
```

[2]https://github.com/gtrak/no.disassemble

```
 67  invokeinterface
     clojure.lang.IFn.invoke(java.lang.Object,
     java.lang.Object, java.lang.Object, java.lang.Object) :
     java.lang.Object [51] [nargs: 5]
 72  invokespecial
     java.lang.AssertionError(java.lang.Object) [54]
 75  checkcast java.lang.Throwable [56]
 78  athrow
 79  aload_1 [x__16079]
 80  aconst_null
 81  astore_1
 82  areturn
```

When Clojure direct linking is enabled, we have an even more optimized code like:

```
public static java.lang.Object invokeStatic(java.lang.Object
    x);
  0  aload_0 [x]
  1  aconst_null
  2  astore_0 [x]
  3  astore_1 [x__12541]
  4  aload_1 [x__12541]
  5  aconst_null
  6  invokestatic
      clojure.lang.Util.identical(java.lang.Object,
      java.lang.Object) : boolean [17]
  9  ifeq 19
 12  getstatic java.lang.Boolean.FALSE : java.lang.Boolean
     [23]
 15  goto 22
 18  pop
 19  getstatic java.lang.Boolean.TRUE : java.lang.Boolean
     [26]
 22  dup
 23  ifnull 37
 26  getstatic java.lang.Boolean.FALSE : java.lang.Boolean
     [23]
 29  if_acmpeq 38
 32  aconst_null
 33  pop
 34  goto 92
 37  pop
 38  new java.lang.AssertionError [28]
 41  dup
 42  ldc <String "Assert failed: "> [30]
 44  iconst_3
 45  anewarray java.lang.Object [32]
 48  dup
 49  iconst_0
 50  aload_1 [x__12541]
 51  invokestatic
```

```
    kongra.ch$chmsg.invokeStatic(java.lang.Object) :
    java.lang.Object [36]
54  aastore
55  dup
56  iconst_1
57  ldc <String "\n"> [38]
59  aastore
60  dup
61  iconst_2
62  iconst_1
63  anewarray java.lang.Object [32]
66  dup
67  iconst_0
68  getstatic kongra.ch$foo.const__5 : java.lang.Object [42]
71  aastore
72  invokestatic
    clojure.lang.ArraySeq.create(java.lang.Object[]) :
    clojure.lang.ArraySeq [48]
75  invokestatic
    clojure.core$pr_str.invokeStatic(clojure.lang.ISeq) :
    java.lang.Object [53]
78  aastore
79  invokestatic
    clojure.lang.ArraySeq.create(java.lang.Object[]) :
    clojure.lang.ArraySeq [48]
82  invokestatic
    clojure.core$str.invokeStatic(java.lang.Object,
    clojure.lang.ISeq) : java.lang.Object [58]
85  invokespecial
    java.lang.AssertionError(java.lang.Object) [61]
88  checkcast java.lang.Throwable [63]
91  athrow
92  aload_1 [x__12541]
93  aconst_null
94  astore_1
95  areturn
```

The code is actually a call to the *(assert ...)* form as defined in the original *(ch ...)* mechanism. With this in mind we may be certain to get a very efficient check-instrumenting code in all the places we use the *ch* library, and the actual cost of every check depends solely on the nature (and cost) of the predicates he uses. It is the responsibility of the programmer to keep them as cheap as possible. From what is well known the *instanceof* predicates in the JVM are particularly fast and cheap both on the CPU and the memory side. The following section discusses a way the *ch* library approaches the type-checks.

5 Class Membership Ch(eck)s

To come up with a proper, fast, run-time type checking, we first provide the following macro that makes the *instance?* call:

```
;; CLASS MEMBERSHIP
(defch chC [c x] `(ch (instance? ~c) ~x))
```

Then we are ready to define another utility macro-writing macro *defchC* that allows the programmer to define his own type checks (besides the ones defined in the *ch* library, see Appendix A):

```
(defmacro defchC
  [chname c]
  (let [x (gensym "x__")]
    `(defch ~chname [~x] `(chC ~~c ~~x))))
```

Among the others the check for *java.lang.Long* type is defined like: *(defchC chLong Long)*. Now we can turn the procedure *(defn foo [x] x)* into *(defn foo [x] (chLong x))*. The resulting procedure has the performance profile as in the following benchmarking result:

```
user> (quick-bench (foo 1))
Evaluation count : 198219084 in 6 samples of 33036514 calls.
             Execution time mean : 1,033696 ns
    Execution time std-deviation : 0,010647 ns
   Execution time lower quantile : 1,012954 ns ( 2,5%)
   Execution time upper quantile : 1,040669 ns (97,5%)
                   Overhead used : 2,032505 ns
```

Similarly for *java.lang.String* we have:

```
user> (quick-bench (chString ""))
Evaluation count : 88947072 in 6 samples of 14824512 calls.
             Execution time mean : 4,806284 ns
    Execution time std-deviation : 0,094706 ns
   Execution time lower quantile : 4,724241 ns ( 2,5%)
   Execution time upper quantile : 4,942071 ns (97,5%)
                   Overhead used : 2,032505 ns
```

These benchmarks shows two things:

- The modern JVM has perfect ways to optimize type checks up to a level that is almost hard to notice from a point of a programmer who writes common code, even in the performance-critical parts.
- Our approach makes no overhead when calling these mechanisms and making the JVM actually do its job.

6 Object Type Equality Ch(eck)s

In a common programmers' practice we often have to ensure two objects have exactly the same type. Check *chLike*, as defined below, serves exactly that:

```
;; OBJECT TYPE EQUALITY
(defmacro chLike* [y x] `(identical? (class ~y) (class ~x)))
(defch    chLike  [y x] `(ch (chLike* ~y) ~x))
```

Let's take look at its performance benchmark, here for two Strings:

```
user> (quick-bench (chLike "aaa" "bbb"))
Evaluation count : 88657260 in 6 samples of 14776210 calls.
             Execution time mean : 4,793546 ns
    Execution time std-deviation : 0,079737 ns
   Execution time lower quantile : 4,724969 ns ( 2,5%)
   Execution time upper quantile : 4,927724 ns (97,5%)
                   Overhead used : 2,032505 ns
```

7 Product (Pair/Tuple) and Co-Product (Discriminated Union Type) Ch(eck)s

To apply more compound checks that use logical operators we define the following generator of predicate checks:

```
(defmacro ch*
  [op chs x]
  (assert (vector? chs) "Must be a chs vector in (ch| ...)")
  (assert (seq     chs) "(ch| ...) must contain some chs"  )
  `(~op ~@(map #(pred-call-form % nil x) chs)))
```

The generator is used to define checks for *tuple* types and for *discriminated union* types:

```
;; PRODUCT (TUPLE)
(defch ch& [chs x] `(ch (ch* and ~chs) ~x))

;; CO-PRODUCT (DISCRIMINATED UNION TYPE)
(defch ch| [chs x] `(ch (ch* or  ~chs) ~x))
```

A special case here is a co-product of exactly two types, known as *Either a b* type constructor in some languages (e.g. Haskell), and its variant - the *Maybe a* type constructor, that can be defined as $typeMaybe\,a = Either\,()\,a$. The *ch* library specifies them as follows:

```
(defch chEither [chl chr x] `(ch| [~chl  ~chr]    ~x))
(defch chMaybe  [ch      x] `(chEither chUnit ~ch ~x))
```

And an example benchmark:

```
user> (quick-bench (chEither chString chLong 1))
Evaluation count : 91910928 in 6 samples of 15318488 calls.
             Execution time mean : 4,879827 ns
    Execution time std-deviation : 0,660203 ns
   Execution time lower quantile : 4,433577 ns ( 2,5%)
   Execution time upper quantile : 5,924470 ns (97,5%)
                   Overhead used : 2,032505 ns
```

Again, we cannot see any significant impact on the performance, other than few nanoseconds.

8 Registry of Ch(eck)s

Additionally the *ch* library provides a registry of checks, that helps the programmer to understand, what kind of checks an object or a collection of objects fulfill. The registry is actually a mapping from string (a name) into a check:

```
(def ^:private CHS (atom {}))
```

To register a new check we use the following macro:

```
(defmacro regch
  [ch]
  (assert (symbol? ch))
  (let [x (gensym "x__")]
    `(regch* ~(str ch) (fn [~x] ~(pred-call-form ch nil x)))))
```

together with its back-end:

```
(defn regch*
  [chname ch]
  (chUnit
   (do
     (assert (string? chname))
     (assert (fn?         ch))
     (swap! CHS
            (fn [m]
              (when (m chname)
                (println "WARNING: chname already in use:"
                    chname))
              (assoc m chname ch))) nil)))
```

Now we may use *chs* function:

```
(defn chs
  ([]
   (chSet (apply sorted-set (sort (keys @CHS)))))

  ([x]
   (chSet (->> @CHS
               (filter (fn [[_ pred]] (pred x)))
               (map first)
               (apply sorted-set))))
  ([x & xs]
    (chSet (->> (cons x xs) (map chs) (apply
        cset/intersection)))))
```

to reach for the information, e.g.:

```
user> (chs 1)
#{"chInteger" "chLong" "chNumber" "chRational"}
```

```
user> (chs "a")
#{"chString"}
user> (chs [1 2 3])
#{"chAssoc" "chColl" "chCounted" "chIfn" "chIndexed"
    "chJavaColl" "chJavaList" "chLookup" "chReversible"
    "chSeqable" "chSequential" "chVec"}
user> (chs inc)
#{"chFn" "chIfn"}
```

There is also a possibility to ask for checks common for a set of objects:

```
user> (chs 1)
#{"chInteger" "chLong" "chNumber" "chRational"}
user> (chs 1.23)
#{"chDouble" "chFloat" "chNumber"}
user> (chs 3/4)
#{"chNumber" "chRatio" "chRational"}
user> (chs 1 1.23 3/4)
#{"chNumber"}
```

Using the information provided the programmer can make a decision on what checks to use in a particular situation.

9 Example Use in Production Setting

The *ch* library is used extensively in production. Among the others it was used to tag some of the elements of *kongra/prelude* package. With the following namespace declaration:

```
(ns kongra.prelude.search
  (:require [kongra.ch      :refer :all]
            [kongra.prelude :refer :all]))
```

we define tree-search routines in the *kongra.prelude.search* namespace. In the first place we define a combiner function that controls the tree search process, by performing order-wise concatenation of search space elements. The concatenation operates on sequences and returns a sequence, thus the use of *chSeq* in the following code:

```
;; COMBINERS
(deftype Comb [f]
  clojure.lang.IFn
  (invoke [_ nodes new-nodes]
    (chSeq (f (chSeq nodes) (chSeq new-nodes)))))
```

For the combiner we define a *chComb* ch(eck) and a proper constructor (*consComb*):

```
(defchC chComb Comb)
(defn consComb [f] (Comb. (chIfn f)))
```

The combiners for breath-first and depth-first search strategies are defined as follows:

```
(def breadth-first-combiner      (consComb    concat))
(def lazy-breadth-first-combiner (consComb lazy-cat'))
```

```
(def depth-first-combiner        (consComb #(concat   %2 %1)))
(def lazy-depth-first-combiner   (consComb #(lazy-cat %2 %1)))
```

Using exactly the same pattern we define an abstraction for goal functions

```
;; GOAL
(deftype Goal [f]
  clojure.lang.IFn
  (invoke [_ x] (boolean (f x))))

(defchC chGoal 'Goal)
(defn consGoal [f] (Goal. (chIfn f)))
```

and for adjacency generators for the tree structure:

```
;; ADJACENCY
(deftype Adjs [f]
  clojure.lang.IFn
  (invoke [_ x] (chSeq (f x))))

(defchC chAdjs Adjs)
(defn consAdjs [f] (Adjs. (chIfn f)))
```

Finally the general *tree-search* procedure uses all the checks defined earlier as presented in the following listing:

```
(defn tree-search
  [start goal? adjs comb]
  (chGoal goal?) (chAdjs adjs) (chComb comb)
  (chMaybe chSome
    (loop [nodes (list start)]
      (when (seq nodes)
        (let [obj (first nodes)]
          (if (goal? obj)
            obj
            (recur (comb (rest nodes) (adjs obj)))))))))
```

Both major search strategies have the implementations like:

```
(defn breadth-first-search
  [start goal? adjs]
  (chMaybe chSome
    (tree-search start
                 (chGoal          goal?)
                 (chAdjs           adjs)
                breadth-first-combiner)))
```

and:

```
(defn depth-first-search
  [start goal? adjs]
  (chMaybe chSome
    (tree-search start
```

```
                (chGoal        goal?)
                (chAdjs         adjs)
                depth-first-combiner)))
```

A very useful procedure *breadth-first-tree-levels* that returns consecutive depth levels of a tree also uses the mechanisms.

```
(defn breadth-first-tree-levels
  [start adjs]
  (chAdjs adjs)
  (chSeq (->> (list              start)
              (iterate #(mapcat adjs %))
              (map              chSeq')
              (take-while          seq))))
```

And finally the following traversal mechanism returns a lazily evaluated sequence of all tree nodes visited according to a breadth-first strategy:

```
(defn breadth-first-tree-seq
  ([start adjs]
   (chAdjs adjs)
   (chSeq (apply concat (breadth-first-tree-levels start
       adjs))))

  ([start adjs depth]
   (chAdjs     adjs)
   (chPosLong depth)
   (chSeq (->> (breadth-first-tree-levels start adjs)
               (take  depth)
               (apply concat)))))
```

10 Plans for the Future Development

Performing the dynamic (run-time) checks for various constraints, including the verification of types is not enough to ensure proper integrity of software projects. Many factors come to mind here, including:

- Multiple versions of libraries that software projects depend upon, together with the information on the actual use of these dependencies, circularity of dependencies, and general lack of reliable sources of coherent packages (libraries) bundled together.
- Lack of the ability to effectively model highly complex systems, like the Java 8 Language Specification, being turned on into a working compiler or at least a static analyzer working according to the rules present in the specification. The estimated amount of work for creating Java compiler is hundreds of man-years, while it would be great to be able to perform activities like that in time an order of magnitude shorter. Problems of this kind were already discussed by us in [11].

These issues and the possibility to solve them in an uniform way will be subjects of our further research activities. We tend to make *ch* library a part of the solution.

References

1. Thomas M., 2016, *How Can Software Be So Hard?*, Gresham College Lecture, Feb. 2016, https://www.youtube.com/watch?v=VfRVz1iqgKU
2. Thomas M., 2015, *Should We Trust Computers*, Gresham College Lecture, Oct. 2015, https://www.youtube.com/watch?v=8SZfjvlbpMw
3. Pierce B.C., 2002, *Types and Programming Languages, 1st Edition*, MIT Press, ISBN-10: 0262162091, ISBN-13: 978-0262162098
4. Pierce B.C., 2004, *Advanced Topics in Types and Programming Languages*, MIT Press, ISBN-10: 0262162288, ISBN-13: 978-0262162289
5. Lamport L., 2002, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*, Addison Wesley, ISBN: 0-321-14306-X
6. Lamport L., Paulson L.L., 1999, *Should Your Specification Language Be Typed?*, ACM Transactions on Programming Languages and Systems, Vol. 21, No. 3, pp. 502-526
7. Emerick Ch., Carper B., Grand Ch., 2012, *Clojure Programming*, O'Reilly Media Inc., ISBN: 978-1-449-39470-7
8. Fogus M., Houser Ch., 2014, *The Joy of Clojure*, Manning Publications; 2 edition, ISBN-10: 1617291412, ISBN-13: 978-1617291418
9. Bird R., Wadler R., 1988, *Introduction to Functional Programming*. Series in Computer Science (Editor: C.A.R. Hoare), Prentice Hall International (UK) Ltd
10. Awodey S., 2010, *Category Theory, Second Edition*, Oxford University Press
11. Grzanek K., 2012, *Prerequisites for Effective Requirements Management*, Journal of Applied Computer Science Methods, Vol. 4, No 1, pp. 21-28
12. Grzanek K., 2017, *ch GitHub Repository*, https://github.com/kongra/ch
13. Grzanek K., 2017, *ch Clojars Page*, https://clojars.org/kongra/ch
14. Grzanek K., 2017, *prelude GitHub Repository*, https://github.com/kongra/prelude
15. Grzanek K., 2017, *prelude Clojars Page*, https://clojars.org/kongra/prelude
16. Grzanek K., 2017, *aptell GitHub Repository*, https://github.com/kongra/aptell
17. Clojure Team, 2017, *clojure.spec Rationale*, https://clojure.org/about/spec
18. Clojure Team, 2017, *clojure.spec Guide*, https://clojure.org/guides/spec
19. Plumatic, 2017, *prismatic/schema GitHub Repository*, https://github.com/plumatic/schema

20. Clojure Team, 2017, *clojure/core.typed GitHub Repository*, https://github.com/clojure/core.typed

21. CircleCI, 2015, *Why we're no longer using core.typed*, https://circleci.com/blog/why-were-no-longer-using-core-typed/

22. Hacker News, 2015, *Why were no longer using core.typed - discussion*, https://news.ycombinator.com/item?id=10271149

A Appendix: Selected Pre-defined Checks

In the beginning we define the following type checks for basic types (classes) belonging to the standard Clojure library and/or forming the Clojure run-time environment:

```
;; COMMON CHS
(defchC chAgent            clojure.lang.Agent)
(defchC chAtom              clojure.lang.Atom)
(defchC chASeq              clojure.lang.ASeq)
(defchC chBoolean                     Boolean)
(defchC chDeref           clojure.lang.IDeref)
(defchC chDouble                       Double)
(defchC chIndexed        clojure.lang.Indexed)
(defchC chLazy           clojure.lang.LazySeq)
(defchC chLong                           Long)
(defchC chLookup         clojure.lang.ILookup)
(defchC chRef                clojure.lang.Ref)
(defchC chSeqable        clojure.lang.Seqable)
(defchC chSequential clojure.lang.Sequential)
```

Another family of the basic checks is the checks defined upon some essential predicates belonging to the standard library of the language:

```
(defch  chAssoc            '(ch associative?))
(defch  chChar                    '(ch char?))
(defch  chClass                  '(ch class?))
(defch  chColl                    '(ch coll?))
(defch  chCounted               '(ch counted?))
(defch  chDecimal               '(ch decimal?))
(defch  chDelay                   '(ch delay?))
(defch  chFloat                   '(ch float?))
(defch  chFn                         '(ch fn?))
(defch  chFuture                '(ch future?))
(defch  chIfn                       '(ch ifn?))
(defch  chInteger               '(ch integer?))
(defch  chKeyword               '(ch keyword?))
(defch  chList                     '(ch list?))
(defch  chMap                      '(ch map?))
(defch  chNumber                '(ch number?))
(defch  chRatio                   '(ch ratio?))
(defch  chRational            '(ch rational?))
```

```
(defch  chRecord               '(ch record?))
(defch  chReduced             '(ch reduced?))
(defch  chReversible       '(ch reversible?))
(defch  chSeq                     '(ch seq?))
(defch  chSorted               '(ch sorted?))
(defch  chString               '(ch string?))
(defch  chSymbol               '(ch symbol?))
(defch  chVar                     '(ch var?))
(defch  chVec                  '(ch vector?))
```

Finally we introduce type checks for few basic Java collection interfaces. It is worth mentioning, that all Clojure collections implement one of these interfaces:

```
(defchC chJavaColl      java.util.Collection)
(defchC chJavaList            java.util.List)
(defchC chJavaMap              java.util.Map)
(defchC chJavaSet              java.util.Set)
```

B Appendix: Selected Tests

The *ch* library is covered with unit tests. Here we present few of them to give the reader another opportunity to get familiar with syntax and behavior of the library. In the following codes we use the namespace definition as below:

```
(ns kongra.ch-test
  (:require [clojure.test :refer :all]
            [kongra.ch :refer :all]))
```

First of all let's define few types with their accompanying type checks:

```
(deftype X []) (defchC chX X)
(deftype Y []) (defchC chY Y)
(deftype Z []) (defchC chZ Z)
```

We also define the following simple checks of various kinds:

```
(defch chMaybeX       '(chMaybe  chX                      ))
(defch chEitherXUnit  '(chEither chX chUnit               ))
(defch chEitherXY     '(chEither chX    chY               ))
(defch chXYZ          '(ch| [chX chY chZ]                 ))
(defch chMaybeLike1   '(chMaybe (chLike 1                )))
(defch chEitherLC     '(chEither (chC Long) (chC Character)))
(defch chEitherLC'
  '(chEither (ch (instance? Long)) (ch (instance? Character))))
```

as well as the compound one:

```
(defch chCompound1
  '(chEither
       (chMaybe  (chLike "aaa"))
       (chEither (chMaybe (ch (instance? Long)))
                 (chMaybe (ch (instance? Character))))))
```

Here the test cases follow:

```
(testing "(ch ...)"
    (is (thrown? AssertionError (ch (nil?)       1)))
    (is (nil?                   (ch (nil?)     nil)))
    (is (false?                 (ch (nil?) nil   1)))
    (is (true?                  (ch (nil?) nil nil))))

(testing "(ch ...) with symbolic preds"
    (is (thrown? AssertionError (ch  nil?        1)))
    (is (nil?                   (ch  nil?      nil)))
    (is (false?                 (ch  nil?  nil   1)))
    (is (true?                  (ch  nil?  nil nil))))

(testing "(chC ...)"
    (is (= ""                   (chC String      "")))
    (is (thrown? AssertionError (chC String       1)))
    (is (thrown? AssertionError (chC String     nil)))
    (is (true?                  (chC String nil  "")))
    (is (false?                 (chC String nil   1)))
    (is (false?                 (chC String nil nil))))

(testing "(defchC ...)"
    (is                         (chX      (X.)))
    (is (thrown? AssertionError (chX        1)))
    (is (thrown? AssertionError (chX      nil)))
    (is (true?                  (chX nil (X.))))
    (is (false?                 (chX nil    1)))
    (is (false?                 (chX nil nil))))

(testing "(chLike ...)"
    (is                         (chLike     1          2))
    (is (thrown? AssertionError (chLike     1     "aaa")))
    (is (thrown? AssertionError (chLike "aaa"         2)))
    (is (thrown? AssertionError (chLike     1       nil)))

    (is (true?                  (chLike   2/3 nil   3/4)))
    (is (false?                 (chLike     1 nil "aaa")))
    (is (false?                 (chLike "aaa" nil     2)))
    (is (false?                 (chLike     1 nil   nil))))

(testing "(chUnit ...)"
    (is (nil?                   (chUnit     nil)))
    (is (thrown? AssertionError (chUnit       1)))

    (is (true?                  (chUnit nil nil)))
    (is (false?                 (chUnit nil ""))))

(testing "(chSome ...)"
    (is                         (chSome        1))
```

```
    (is (thrown? AssertionError (chSome      nil)))

    (is (true?                  (chSome  nil  "")))
    (is (false?                 (chSome  nil nil))))

(testing "(chMaybe ...)"
    (is (nil?                   (chMaybe chX      nil)))
    (is                         (chMaybe chX      (X.)))
    (is (thrown? AssertionError (chMaybe chX      (Y.))))
    (is (true?                  (chMaybe chX nil  nil)))
    (is (true?                  (chMaybe chX nil  (X.))))
    (is (false?                 (chMaybe chX nil  (Y.))))

    (is (nil?                   (chMaybe chUnit nil)))
    (is (thrown? AssertionError (chMaybe chUnit (X.))))
    (is (thrown? AssertionError (chMaybe chUnit (Y.)))))

(testing "(chEither ...)"
    (is (nil?                   (chEither chX chUnit  nil)))
    (is                         (chEither chX chUnit  (X.)))
    (is (thrown? AssertionError (chEither chX chUnit  (Y.))))
    (is                         (chEither chX chY     (X.)))
    (is                         (chEither chX chY     (Y.)))
    (is (thrown? AssertionError (chEither chX chY    (Z.))))
    (is (thrown? AssertionError (chEither chX chY     nil)))

    (is (true?   (chEither chX chUnit nil  nil)))
    (is (true?   (chEither chX chUnit nil (X.))))
    (is (false?  (chEither chX chUnit nil (Y.))))
    (is (true?   (chEither chX chY    nil (X.))))
    (is (true?   (chEither chX chY    nil (Y.))))
    (is (false?  (chEither chX chY    nil (Z.))))
    (is (false?  (chEither chX chY    nil nil))))

(testing "(chCompound1 ...)"
    (is                         (chCompound1 (+ 1 2 3 4)))
    (is                         (chCompound1          \c))
    (is                         (chCompound1       "xyz"))
    (is (nil?                   (chCompound1         nil)))
    (is (thrown? AssertionError (chCompound1         3/4)))

    (is (true?                  (chCompound1 nil (+ 1 2 3 4))))
    (is (true?                  (chCompound1 nil          \c)))
    (is (true?                  (chCompound1 nil       "xyz")))
    (is (true?                  (chCompound1 nil         nil)))
    (is (false?                 (chCompound1 nil         3/4))))
```

Information for Authors

Authors are invited to submit original papers, within the scope of the JACSM, with the understanding that their contents are unpublished and are not being actively under consideration for publication elsewhere.

For any previously published and copyrighted material, a special permission from the copyright owner is required. This concerns, for instance, figures or tables for which copyright exists. In such a case, it is necessary to mention by the author(s), in the paper, that this material is reprinted with the permission.

Manuscripts, in English, with an abstract and the key words, are to be submitted to the Editorial Office in electronic version (both: source and pdf formats) via e-mail. A decision to accept/revise/reject the manuscript will be sent to the Author along with the recommendations made by at least two referees. Suitability for publications will be assessed on the basis of the relevance of the paper contents, its originality, technical quality, accuracy and language correctness.

Upon acceptance, Authors will transfer copyright of the paper to the publisher, i.e. the Academy of Management, in Lodz, Poland, by sending the paper to the JACSM Editorial Office.

The papers accepted for publication in the JACSM must be typeset by their Authors, according to the requirements concerning final version of the manuscripts. However, minor corrections may have been carried out by the publisher, if necessary.

The printing area is 122 mm × 193 mm. The text should be justified to occupy the full line width, so that the right margin is not ragged, with words hyphenated as appropriate. Please fill pages so that the length of the text is no less than 180 mm.

The first page of each paper should contain its title, the name of the author(s) with the affiliation(s) and e-mail address(es), and then the abstract and keywords, as show above. Capital letters must be applied in the title of a paper, and 14 pt. boldface font should be used, as in the example at the first page. Use: 11-point type for the name(s) of the author(s), 10-point type for the address(es) and the title of the abstract, 9-points type for the abstract and the key words.

For the main text, please use 11-point type and single-line spacing. We recommend using Times New Roman (TNR) fonts in its normal format. Italic type may be used to emphasize words in running text. Bold type and underlining should be avoided. With these sizes, the interline distance should be set so that some 43 lines occur on a full-text page.

The papers should show no printed page numbers; these are allocated by the Editorial Office.

Authors have the right to post their Academy of Management-copyrighted material from JACSM on their own servers without permission, provided that the server displays a prominent notice alerting readers to their obligations with respect to copyrighted material. Posted work has to be the final version as printed, include the copyright notice and all necessary citation details (vol, pp, no, …).

An example of an acceptable notice is:
“This material is presented to ensure timely dissemination of scholarly work, and its personal or educational use is permitted. Copyright and all rights therein are retained by the Copyright holder. Reprinting or re-using it in any form requires permission of the Copyright holder.

The submission format for the final version of the manuscript
and the MS Word template are available on our web site at:
http://acsm.spoleczna.pl

Journal of Applied Computer Science Methods

Contents