Communications
in Computer and Information Science 275

Leszek A. Maciaszek   Kang Zhang (Eds.)

Evaluation of Novel Approaches to Software Engineering

6th International Conference, ENASE 2011
Beijing, China, June 8-11, 2011
Revised Selected Papers

Springer

Volume Editors

Leszek A. Maciaszek
Wrocław University of Economics
Institute of Business Informatics
53-345 Wrocław, Poland
and
Macquarie University
Department of Computing
Sydney, NSW 2109, Australia
email: leszek.maciaszek@mq.edu.au

Kang Zhang
University of Texas at Dallas
Erik Jonsson School of Engineering
and Computer Science
800 W. Campbell Road
Richardson, TX 75080-3021, USA
E-mail: kzhang@utdallas.edu

Preface

The mission of the ENASE (Evaluation of Novel Approaches to Software Engineering) conferences is to be a prime international forum for discussing and publishing research findings and IT industry experiences with relation to evaluation of novel approaches to software engineering. By comparing novel approaches with established traditional practices and by evaluating them against software quality criteria, the ENASE conferences advance knowledge and research in software engineering, identify the most hopeful trends, and propose new directions for consideration by researchers and practitioners involved in large-scale software development and integration.

This CCIS volume contains papers of the 6th edition of ENASE held in Beijing, China. The previous conferences took place in Erfurt, Germany (2006), Barcelona, Spain (2007), Madeira, Portugal (2008), Milan, Italy (2009), and Athens, Greece (2010). There is a growing research community around ENASE, and it is increasingly recognized as an important international conference for researchers and practitioners to review and evaluate emerging as well as established SE methods, practices, architectures, technologies and tools. The ENASE conferences host also keynotes, workshops, and panels.

For the 6th ENASE in Beijing we received 75 papers from 31 countries, of which 55 were regular papers and 20 were short or position papers. The reviewing process was carried out by about 80 members of the ENASE 2011 Program Committee. The final decision of acceptance/rejection was taken based on the reviews received by the PC Co-chairs Leszek Maciaszek and Kang Zhang. Borderline papers were subjected to extra considerations and discussions before decisions were reached.

For ENASE 2011, we finally accepted 18 full papers (with scores 4 and above; max. 6) and 10 short papers. The relevant acceptance statistics for full papers are: 32.7% (based on 55 submissions) or 24% (based on 75 submissions)—clearly, the former percentage is more truthful. The acceptance rate confirms the desire of ENASE conferences to ensure a high quality of presented papers and associated events. All six ENASE conferences had the acceptance rate for full papers at around or below 30%.

Papers accepted for ENASE 2011 were presented in nine categories:

1. Software Quality and Testing
2. Requirements Engineering
3. Programming
4. Software Processes and Methods

5. Software Tools and Environments
6. Business Process and Services Modeling
7. Software Components
8. Software Effort and Processes
9. Socio-Technical Aspects of Software Development

November 2011   Leszek Maciaszek
                                                          Kang Zhang

Organization

Conference Chair

Joaquim Filipe   Polytechnic Institute of Setúbal / INSTICC, Portugal

Program Co-chairs

Leszek Maciaszek   Macquarie University, Australia / University of Economics, Poland
Kang Zhang   The University of Texas at Dallas, USA

Organizing Committee

Sérgio Brissos	INSTICC, Portugal
Patrícia Alves	INSTICC, Portugal
Helder Coelhas	INSTICC, Portugal
Vera Coelho	INSTICC, Portugal
Andreia Costa	INSTICC, Portugal
Patrícia Duarte	INSTICC, Portugal
Bruno Encarnação	INSTICC, Portugal
Liliana Medina	INSTICC, Portugal
Carla Mota	INSTICC, Portugal
Raquel Pedrosa	INSTICC, Portugal
Vitor Pedrosa	INSTICC, Portugal
Daniel Pereira	INSTICC, Portugal
Cláudia Pinto	INSTICC, Portugal
José Varela	INSTICC, Portugal
Pedro Varela	INSTICC, Portugal

Program Committee

Colin Atkinson, Germany	Rebeca Cortazar, Spain
Farokh B. Bastani, USA	Massimo Cossentino, Italy
Giuseppe Berio, France	Philippe Dugerdil, Switzerland
Ghassan Beydoun, Australia	Angelina Espinoza, Spain
Maria Bielikova, Slovak Republic	Joerg Evermann, Canada
Dumitru Burdescu, Romania	Maria João Ferreira, Portugal
Wojciech Cellary, Poland	Agata Filipowska, Poland
Panagiotis Chountas, UK	Juan Garbajosa, Spain

Janusz Getta, Australia

Cesar Gonzalez-Perez, Spain

Ian Gorton, USA

Jeff Gray, USA

Hans-Gerhard Gross, The Netherlands

Brian Henderson-Sellers, Australia

Rene Hexel, Australia

Charlotte Hug, France

Bernhard G. Humm, Germany

Zbigniew Huzar, Poland

Akira Imada, Belarus

Warwick Irwin, New Zealand

Stefan Jablonski, Germany

Slinger Jansen, The Netherlands

Monika Kaczmarek, Poland

Wan Kadir, Malaysia

Robert S. Laramee, UK

Xabier Larrucea, Spain

George Lepouras, Greece

Pericles Loucopoulos, UK

Graham Low, Australia

Jian Lu, China

André Ludwig, Germany

Leszek Maciaszek, Australia

Cristiano Maciel, Brazil

Lech Madeyski, Poland

SaschaMueller-Feuerstein, Germany

Johannes Müller, Germany

Anne Hee Hiong Ngu, USA

Andrzej Niesler, Poland

Janis Osis, Latvia

Mieczyslaw Owoc, Poland

Marcin Paprzycki, Poland

Jeffrey Parsons, Canada

Oscar Pastor, Spain

Naveen Prakash, India

Lutz Prechelt, Germany

Elke Pulvermueller, Germany

Rick Rabiser, Austria

Gil Regev, Switzerland

Artur Rot, Poland

Francisco Ruiz, Spain

Krzysztof Sacha, Poland

Motoshi Saeki, Japan

Heiko Schuldt, Switzerland

Manuel Serrano, Spain

Jerzy Surma, Poland

Stephanie Teufel, Switzerland

Rainer Unland, Germany

Olegas Vasilecas, Lithuania

Igor Wojnicki, Poland

Kang Zhang, USA

Auxiliary Reviewers

Saquib Anwar, Canada

Roman Lukyanenko, Canada

Giovanni Pilato, Italy

Luca Sabatucci, Italy

Valeria Seidita, Italy

Invited Speakers

Harold Krikke	Tilburg University, The Netherlands
Xuewei Li	Beijing Jiaotong University, China
Kecheng Liu	University of Reading, UK
Leszek Maciaszek	Macquarie University / University of Economics, Australia / Poland
Yannis A. Phillis	Technical University of Crete, Greece
Shoubo Xu	Chinese Academy of Engineering / Beijing Jiaotong University, China
Yulin Zheng	UFIDA, China
Lida Xu	Old Dominion University, USA

Table of Contents

Papers

A Study on Software Effort Prediction Using Machine Learning Techniques

Wen Zhang, Ye Yang, and Qing Wang

Laboratory for Internet Software Technologies, Institute of Software
Chinese Academy of Sciences, Beijing 100190, P.R.China
{zhangwen,ye,wq}@itechs.iscas.ac.cn

Abstract. This paper conducts a study on of software effort prediction using machine learning techniques. Both supervised and unsupervised learning techniques are employed to predict software effort using historical dataset. The unsupervised learning as k-medoids clustering equipped with different similarity measures is used to cluster projects in historical dataset. The supervised learning as J48 decision tree, back propagation neural network (BPNN) and naïve Bayes is used to classify the software projects into different effort classes. We also impute the missing values in the historical datasets and then machine learning techniques are adopted to predict software effort. Experiments on ISBSG and CSBSG datasets demonstrate that unsupervised learning as k-medoids clustering produced a poor performance. Kulzinsky coefficient has the best performance in measuring the similarities of projects. Supervised learning techniques produced superior performances than unsupervised learning techniques in software effort prediction. BPNN produced the best performance among the three supervised learning techniques. Missing data imputation improved the performances of both unsupervised and supervised learning techniques in software effort prediction.

Keywords: Effort prediction, Machine learning, k-medoids, BPNN, Missing imputation.

1 Introduction

The task of software effort prediction is to estimate the needed effort to develop a software artifact [17]. Overestimate of software effort may lead to tight schedule of development and faults may leave in the system after delivery, whereas underestimate of effort may lead to delay of deliver of system and complains from customers. The importance of software development effort prediction has motivated the construction of prediction models to estimate the needed effort as accurate as possible.

Current software effort prediction techniques can be categorized into four types: empirical, regression, theory-based, and machine learning techniques [2]. Machine Learning (ML) techniques learn patterns (knowledge) from historical project data and use these patterns for effort prediction, such as artificial neural network (ANN), decision tree, and naive Bayes. Recent studies [2] [3] provide detailed reviews of different studies on predicting software development effort.

The primary concern of this paper is on using machine learning techniques to predict software effort. Despite that COCOMO has provided a viable solution to effort estimation by building analytic model, machine learning techniques such as naïve Bayes and artificial neural network have come up with alternative approaches by making use of knowledge learned from historical projects. Although machine learning techniques though may not be the best solution for effort estimation, we believe they can be used at least by project managers to complement other models. Especially in intensely competitive software market, accurate estimation of software development effort has a decisive effect on success of a software project. Consequently, effort estimation using different techniques, and further risk assessment of budget overrun are of necessity for a trustworthy software project [5].

The basic idea of using machine learning techniques for effort prediction is that, historical data set contains many historical projects which are described by features with their values to characterize those projects and, similar values of the features of projects may induce almost the similar project efforts. The task of machine learning methods is to learn the inherent patterns of feature values and their relations with project efforts, which can be used for predicting the effort of new projects.

The rest of this paper was organized as follows. Section 2 introduce machine learning techniques to software effort prediction. Both unsupervised and supervised learning techniques are introduced. Section 3 conducts experiments to examine the effectiveness of machine learning techniques on software prediction. The datasets we used in the experiments, and the performance measures for unsupervised and supervised learning techniques are also introduced. The experimental results are illustrated with explanations. Section 4 presents the threats to validity to this research. Section 5 reviews related work of this paper. Section 6 concluds this paper.

2   Effort Prediction Using Machine Learning

2.1   Effort Prediction with Unsupervised Learning

Generally, researchers in software engineering hold the assumption that projects with similar characteristics, such as the number of function points, application domain and programming language, are expected to have approximately equivalent efforts (at least they should be in the same effort class). In the standpoint of machine learning, clustering software projects on the basis of a random subset can capture information on the unobserved attributes [8]. If we regarded effort as an attribute that also characterize software projects in the data set, then software effort can be deduced by clustering projects using other attributes.

To validate this assumption, k-medoids [9] is adopted for clustering the projects and three similarity measures are used to measure the similarities of boolean vectors that represent the software projects. k-medoids is actually evolved from k-means [9] and their difference lies in that k-medoids assigns existing element in a cluster as cluster center but k-means assigns mean vector of elements in a cluster as the cluster center. We adopt k-medoids other than k-means because the mean vector of boolean vectors lacks explainable meaning in practice nevertheless their medoid denotes a real project. The typical k-medoids clustering is implemented by partitioning around medoids (PAM)

Algorithm 1. The k-medoids clustering implemented by PAM algorithm.

Input:

k, the number of clusters

m, Boolean vectors

Output:

k clusters partitioned from the m Boolean vectors.

Procedure:

1. Initialize: randomly select k of the m Boolean vectors as the mediods.
2. Associate each Boolean vector to the closest medoid under predefined similarity measure.
3. For each mediod d
4.   For each non-medoid Boolean vector b
5.   Swap d and b and compute the total cost of the configuration
6.   End for
7. End for
8. Select the configuration with the lowest cost.
9. Repeat steps 2 to 5 until there is no change in the medoid.

algorithm as depicted in Algorithm 1. The computation complexity and the convergence of PAM algorithm refers to [9].

The three adopted similarity measures are Dice coefficient, Jaccard coefficient and Kulzinsky coefficient for binary vectors [10]. Assuming that D_i and D_j are two projects represented by two n-dimensional Boolean vectors and $s_{pq}(D_i, D_j)$ is the number of entries in D_i and D_j whose values are p and q respectively, we define A, B, C and D in Equation 1.

$$A = s_{11}(D_i, D_j), B = s_{01}(D_i, D_j),$$
$$C = s_{10}(D_i, D_j), E = s_{00}(D_i, D_j) \tag{1}$$

The similarity measures of Dice, Jaccard and Kulzinsky coefficients are listed in Table 1. We regard that E, which means that the characteristic does not exist in both D_i and D_j, might not be an important factor when measuring similarity of two projects because, the proportion of zero in values of variables is very large in both ISBSG and CSBSG data set.

Table 1. Three similarity measure used in k-medoids clustering

Measure	Similarity	Range
Dice	$\frac{A}{2A+B+C}$	$[0, \frac{1}{2}]$
Jaccard	$\frac{A}{A+B+C}$	$[0,1]$
Kulzinsky	$\frac{A}{B+C}$	∞

2.2 Effort Prediction with Supervised Learning

The employed supervised learning techniques are those usually used in effort prediction, including J48 decision tree, BPNN and naive Bayes. The J48 decision tree classifier follows the following simple algorithm. In order to classify the effort of a software project, it firstly creates a decision tree based on the values of variables in the training

data set. Whenever it encounters a set of boolean vectors (training set) it identifies the variable that has the largest information gain [14]. Among the possible values of this variable, if there is any value for which there is no ambiguity, that is, for which the projects falling within this value having the same label of effort, then we terminate that branch and assign to the terminal node the label of effort.

The back propagation neural network (BPNN) [15] is used to classify the software projects in both ISBSG and CSBSG data sets as well. BPNN defines two sweeps of the network: first a forward sweep from the input layer to the output layer and second a backward sweep from the output layer to the input layer. The back ward sweep is similar to the forward sweep except that error values are propagated back through the network to determine how the weights of neurons are to be changed during training. The objective of training is to find a set of network weights of neurons that construct a model for prediction with minimum error.

A three-layer fully connected feed-forward network which consists of an input layer, a hidden layer and an output layer is adopted in the experiments. The "tansigmod" function is used in the hidden layer with 5 nodes and "purelinear" function for the output layer with 3 nodes [17]. The network of BPNN is designed as shown in Figure 1.
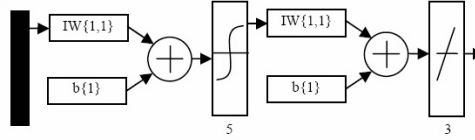
Fig. 1. BPNN with 5 nodes in hidden layer and 3 nodes in output layer

Naive Bayes [16] is a well known probabilistic classifier in machine learning. It is based on the Bay's theorem of posteriori probability and assumes that the effect of an attribute value on a given class is independent of the value of the other attributes. This class conditional independence assumption simplifies computation involved in building the classifier so we called the produced classifier "naive". Compared to other traditional prediction models, naive Bayes provides tools for risk estimation and allows decision-makers to combine historical data with subjective expert estimates [2].

The J48 decision tree and naive Bayes are implemented using Weka (Waikato Environment for Knowledge Analysis) (`http://www.cs.waikato.ac.nz/ml/weka/`) and, BPNN is implemented using Matlab simulink tool box (`http://www.mathworks.com/products/neural-net/`). Also, MINI algorithm [12] is used to impute the missing values of Boolean vectors if necessary.

3   Experiments

3.1   The Data Sets

We employed two data sets to investigate the predictability of software effort using machine learning techniques. The one is ISBSG (International Software Benchmarking Standard Group) data set (`http://www.isbsg.org`) and the other one is CSBSG (Chinese Software Benchmarking Standard Group) data set [7].

ISBSG Data Set. ISBSG data set contains 1238 projects from insurance, government, etc., of 20 different countries and each project was described with 70 attributes. To make the data set suitable for the experiments, we conduct three kinds of preprocessing: data pruning, data discretization and adding dummy variables.

We pruned ISBSG data set into 249 projects with 22 attributes using the criterion that each project must have at least 2/3 attributes whose values are observed and, for each attribute, its values must be observed on at least 2/3 of total projects. We adopt the criterion for data selection in that too many missing values will deteriorate the performances of most machine techniques thus a convincing evaluation of software effort prediction is impossible. Among the 22 attributes, 18 of them are nominal attributes and 4 of them are continuous attributes. Table 2 lists the attributes used in the ISBSG data set.

Data discretization is utilized to transfer the continuous attributes into discrete variables. The values of each continuous attribute are preprocessed into 3 unique partitions. Too many partitions of values of an attribute will cause data redundancy nevertheless too few partitions may not capture the distinction of values of continuous attributes.

For each nominal attribute, dummy variables are added according to its unique values to make all variables having binary values [24]. As a result, all the projects are described using 99 boolean variables with 0-1 and missing values. Only some of machine learning techniques can handle mixed data of nominal and continuous values but, most machine learning techniques can be used to handle Boolean values. In preprocessing, missing values are denoted as "-1" and kept for all projects on corresponding variables. Table 3 shows the value distribution of variables of ISBSG projects after preprocessing. Most values of the variables are zeros due to the transferring from discrete attributes to binary variables.

Finally, software effort of those selected 249 projects in the ISBSG data set was categorized into 3 classes. The projects with "normalized work effort" more than 6,000 person hours were categorized into the class with effort label as "high", projects with "normalized work effort" between 2,000 and 6,000 person hours as "medium" and projects with "normalized work effort" less than 2,000 person hours as "low". Table 4 lists the effort distribution of the selected projects in the ISBSG data set.

CSBSG Data Set. CSBSG data set contains 1103 projects from Chinese software industry. It was created in 2006 with its mission to promote Chinese standards of software productivity. CSBSG projects were collected from 140 organizations and 15 regions across China by Chinese association of software industry. Each CSBSG project is described with 179 attributes. The same data preprocessing as those used in ISBSG data set is used on CSBSG data set. In data pruning, 104 projects and 32 attributes (15 nominal attributes and 17 continuous attributes) are extracted from CSBSG data set. Table 5 lists the attributes used in the CSBSG data set.

In data discretization, the values of each continuous attribute are partitioned into 3 unique classes. Dummy variables are added to transfer nominal attributes into Boolean variables. As a result, 261 Boolean variables are produced to describe the 104 projects with missing values denoted as "-1". The value distribution of variables of CSBSG projects is shown in Table 6 and we can see that CSBSG data set has more missing values than ISBSG data set.

Table 2. The used attributes from ISBSG data set

Branch	Description	Type
Sizing technique	Count Approach	Nominal
	Adjusted Functional Points	Continuous
Schedule	Project Activity Scope	Nominal
Quality	Minor Defects	Continuous
	Major Defects	Continuous
	Extreme Defects	Continuous
Grouping Attributes	Development Type	Nominal
	Organization Type	Nominal
	Business Area Type	Nominal
	Application Type	Nominal
Architecture	Architecture	Nominal
Documents Techniques	Development Techniques	Nominal
Project Attributes	Development Platform	Nominal
	Language Type	Nominal
	Primary Programming language	Nominal
	1st Hardware	Nominal
	1st Operating System	Nominal
	1st Data Base System	Nominal
	CASE Tool Used	Nominal
	Used Methodology	Nominal
Product Attributes	Intended Market	Nominal

Table 3. The value distribution of variables of projects in ISBSG data set

Value	Proportion
1	20% $\sim$ 50%
0	20% $\sim$ 60%
-1	5% $\sim$ 33%

Table 4. The effort classes categorized in ISBSG data set

Class No	Number of projects	Label
1	64	Low
2	85	Medium
3	100	High

Finally, the projects in CSBSG data set were categorized into 3 classes according to their real efforts. The projects with "normalized work effort" more than 5,000 person hours were categorized into the class with effort label as "high", projects with "normalized work effort" between 2,000 and 5,000 person hours as "medium" and projects with "normalized work effort" less than 2,000 person hours as "low". Table 7 lists the effort distribution of the selected projects in the CSBSG data set.

With supervised learning, our experiments are carried out with 10-flod cross-validation technique. For each experiment, we divide the whole data set (ISBSG or CSBSG data set) into 10 subsets. 9 of 10 subsets are used for training and the remaining 1

Table 5. The used attributes from CSBSG data set

Branch	Description	Type
Basic information of projects	Count Approach	Nominal
	City of development	Nominal
	Business area	Nominal
	Development type	Nominal
	Application type	Nominal
	Development Platform	Nominal
	IDE	Nominal
	Programming Language	Nominal
	Operation System	Nominal
	Database	Nominal
	Target Market	Nominal
	Architecture	Nominal
	Maximum Number of Concurrent Users	Nominal
	Life-cycle model	Nominal
	CASE Tool	Nominal
Size	Added lines of code	Continuous
	Revised lines of code	Continuous
	Reused lines of code	Continuous
	Number of team members in inception phase	Continuous
	Number of team members in requirement phase	Continuous
	Number of team members in design phase	Continuous
	Number of team members in coding phase	Continuous
	Number of team members in testing phase	Continuous
Schedule	Time limit in planning	Continuous
Quality	Predicted number of Defects in requirements phase	Continuous
	Predicted number of Defects in design phase	Continuous
	Predicted number of Defects in testing phase	Continuous
	Number of defects within one month after deliver	Continuous
Other	Number of requirement changes in requirement phase	Continuous
	Number of requirement changes in design phase	Continuous
	Number of requirement changes in coding phase	Continuous
	Number of requirement changes in testing phase	Continuous

Table 6. The value distribution of variables for describing projects in CSBSG data set

Value	Proportion
1	15% ∼ 40%
0	20% ∼ 60%
-1	10% ∼ 33%

Table 7. The effort classes categorized in CSBSG data set

Class No	Number of projects	Label
1	27	Low
2	31	Medium
3	46	High

subset was used for testing. We repeat the experiment 10 times and, the performance of the prediction model is measured by the average of 10 accuracies of the 10 repetitions.

3.2 Evaluation Measures

In software engineering, the deviation of predicted effort to real effort is used to measure the accuracy of effort estimators, such as MMRE (Magnitude of Relative Error), PRED(x) (Prediction within x) and AR (Absolute Residual) [6]. In machine learning, the performance of classification is often evaluated by accuracy and, F-measure [13] is usually used to evaluate the performance of unsupervised learning. Essentially, the evaluation measures of effort predictors in software engineering and those in machine learning do not conflict. In this paper, we adopted the measures from machine learning to evaluate the performances of effort predictors.

Accuracy. Assuming that $D = (D_1, ..., D_i, ..., D_m)$ is a collection of software projects, where D_i is a historical project and it is denoted by n attributes $X_i (1 \leqslant i \leqslant n)$. That is, $D_i = (x_{i1}, ..., x_{ij}, ..., x_{in})^T$. h_i denotes the label of effort for project D_i.

x_{ij} is the value of attribute $X_j (1 \leqslant j \leqslant n)$ on D_j. To evaluate the performance of a classifier in effort prediction, the whole data set was divided into two subsets: one is used for training the classifier and the other one is used for testing. That is, $D = (D_{train} \mid D_{test}) = (D_1, ..., D_k \mid D_{k+1}, ..., D_m)^T$, where k is the predefined number of projects in training set and m is the total number of projects in D. For instance, in 10-fold-cross validation, k should be predefined as $0.9m$ and the remaining $0.1m$ projects are used for testing the trained model. h_i is known for training set but remains unknown for testing set. By machine learning on the training set, a classifier denoted as M is produced. If we define a Boolean function F as Equation 2, then the performance of M is evaluated by accuracy as Equation 3.

$$F(M(D_j), h_j) = \begin{cases} 1, \text{ if } M(D_j) = h_j; \\ 0, \text{ otherwise.} \end{cases} \tag{2}$$

$$accuracy = \frac{1}{m-k} \sum_{k < j \leq m} F(M(D_j), h_j) \tag{3}$$

F-measure. In classification, Y was partitioned into l clusters and l is a predefined number of clusters in the data set. That is, $Y = c_1, ..., c_l$, $c_i = \{D_{i,1}, ..., D_{i,|c_i|}\}$ $(1 \leq i \leq l)$ and $c_i \cap c_j = \phi$. F-measure [14] is employed to evaluate the performance of unsupervised learning (clustering). The formula of F-measure is depicted as Equations 7 with the supports of Equations 4, 5, and 6.

$$P(i, j) = \frac{n_{i,j}}{n_j} \tag{4}$$

$$R(i, j) = \frac{n_{i,j}}{n_i} \tag{5}$$

$$F(i,j) = \frac{2 \times P(i,j) \times R(i,j)}{P(i,j) + R(i,j)} \qquad (6)$$

$$F - measure = \sum_i \frac{n_i}{n} max_j F(i,j) \qquad (7)$$

Here, n_i is the number of software projects with effort label h_i, n_j is the cardinality of cluster c_j, and $n_{i,j}$ is the number of software projects with effort label h_i in cluster c_j. n is the total number of software projects in Y. $P(i,j)$ is the proportion of projects in cluster c_j with effort label h_i; $R_{i,j}$ is the proportion of projects with effort label h_i in cluster c_j; $F(i,j)$ is the F-measure of cluster c_j with respect to projects with effort label h_i. In general, the larger the F-measure is, the better is the clustering result is.

3.3   Experimental Results

Results from Unsupervised Learning. PAM algorithm is used to cluster the software projects in ISBSG and CSBSG data sets. The number of clusters is predefined as the number of classes. That is, the parameter k in PAM algorithm for both ISBSG and CSBSG data sets was set as 3. Without imputation, we regard the missing values in boolean vectors as zeros. We also employed MINI imputation technique [11] to impute the missing values before clustering. Due to the unstable clustering results caused by initial selection of cluster centers in PAM algorithm, we repeated each experiment 10 times and ensemble clustering proposed by Zhou et al [12] was utilized to produce the final clusters. Table 8 shows the performances of k-medoids clustering on ISBSG data set using PAM algorithm with three similarity measures with and without imputation.

We can see from Table 8 that in similarity measure, Kulzinsky coefficient has the best performance among the three measures and Jaccard coefficient has better performance than Dice coefficient. In k-medoids clustering without (with) imputation, Kulzinsky coefficient increases the F-measure by 16.29% (17.45%) and Jaccard Coefficient increases the F-measure by 8.6% (5.78%) using Dice coefficient as the baseline.

This outcome illustrates that the number of common entries as in Equation 7 is more important than other indices in similarity measure of software project in effort prediction using clustering. Imputation significantly improves the quality of clustering results. This validates the effectiveness of imputing missing values of projects represented by boolean vectors in k-medoids clustering.

To have a detailed look at the clustering results, Table 9 shows the projects in the produced clusters across the classes in Table 4. These clusters were produced by k-medoids clustering using Kulzinsky coefficient with imputation (i.e. F-measure is 0.4624). We can see that k-medoids clustering actually has not produced high-quality clusters in the ISBSG data set. The results are not good as acceptable in real practice of software effort prediction. For instance, cluster 2 mixes projects in both class 1 and 2 and, most projects in one class scatter on more than one cluster such as the projects in class 2 and class 3.

Table 10 shows the performance of PAM algorithm on CSBSG data set. Table 11 shows the distribution of projects in clusters across classes. Similarly, k-medoids

Table 8. k-medoids clustering on ISBSG data set

Similarity Measure	F-measure	
	Without imputation	With imputation
Dice Coefficient	0.3520	0.3937
Jaccard Coefficient	0.3824	0.4371
Kulzinsky Coefficient	0.4091	0.4624

Table 9. Clustering result using Kulzinsky coefficient with imputation on ISBSG data set

Similarity	Class 1	Class 2	Class 3	Total
Cluster 1	26	23	28	77
Cluster 2	32	40	27	99
Cluster 3	6	22	45	73
Total	64	85	100	249

clustering has not produced a favorable performance on CSBSG data set. By contrast, the performance of k-medoids clustering on CSBSG data set is worse than that on ISBSG data set. Without (with) imputation, the average F-measure on the three coefficients on CSBSG data set is decreased by 7.5% (3.7%) using the average on ISBSG data set as baseline. We explain this outcome as that CSBSG data set has less ones and more missing values (denoted as "-1") in boolean vectors than ISBSG data set, as can be seen in Tables 3 and 6. Based on the analysis, the predictability of software effort using unsupervised learning is not acceptable by software industry.

Table 10. k-medoids clustering on CSBSG data set

Similarity Measure	F-measure	
	Without imputation	With imputation
Dice Coefficient	0.3403	0.3772
Jaccard Coefficient	0.3881	0.4114
Kulzinsky Coefficient	0.4065	0.4560

Table 11. Clustering result using Kulzinsky coefficient with imputation on CSBSG data set

Similarity	Class 1	Class 2	Class 3	Total
Cluster 1	9	11	16	36
Cluster 2	10	10	17	37
Cluster 3	8	10	13	31
Total	27	31	46	104

Results from Supervised Learning. Table 12 shows the performances of the three mentioned classifiers in classifying the projects in ISBSG data set. On average, we can see that BPNN outperforms other classifiers in classifying the software projects based on efforts. J48 decision tree has better performance than naïve Bayes. Using the performance of naïve Bayes as the baseline, BPNN increases the average accuracy by 16.25% (11.71%) and J48 decision tree by 5.6% (2.5%) without (with) imputation.

We explain this outcome that BPNN has the best capacity to eliminate the noise and peculiarities because it adopts back sweep to change the weights of neurons for reducing errors of predictive model. However, the performance of BPNN is not robust as other classifiers (we observe this point from its standard deviation). The adoption of cross-validation technique may reduce overfitting of BPNN to some extent but, it cannot eliminate the drawback of BPNN entirely. The J48 decision tree classifies the projects using learned decision rules. Due to the adoption of information gain [15], those variables having more discriminative power will be fetched out by J48 in earlier branches in constructing decision rules and thus, the noise and peculiarities connotated in the variables with less discriminative power will be ignored automatically (especially in tree pruning).

naïve Bayes has the worst performance among the three classifiers in classifying software efforts. We explain this as that the variables used for describing projects may not be independent of each other. Moreover, naïve Bayes regard all variables as has equivalent weights as each other in the prediction model. The conditional probabilities of all variables have the same weight when predicting the label of an incoming project. However, in fact, some variables of projects have more discriminative power than other variables in deciding the project effort. The noise and peculiarities are often contained in the variables those have little discriminative power and those variables should be given less importance in the prediction model. We conjecture that this fact is also the cause of the poor performance of k-medoids in project clustering projects. In the same manner as that in k-medoids clustering, MINI technique has significantly improved the performance of project classification by imputing missing values in Boolean vectors.

Table 12. Classification of software project efforts on ISBSG data set

Classifier	Average accuracy $\pm$ Standard Deviation	
	Without imputation	With imputation
J48 decision tree	0.5706 ± 0.1118	0.5917 ± 0.1205
BPNN	0.6281 ± 0.1672	0.6448 ± 0.1517
naïve Bayes	0.5403 ± 0.1123	0.5772 ± 0.1030

Table 13. Classification of software project efforts on CSBSG data set

Classifier	Average accuracy $\pm$ Standard Deviation	
	Without imputation	With imputation
J48 decision tree	0.4988 ± 0.1103	0.5341 ± 0.1322
BPNN	0.1650 ± 0.1650	0.6132 ± 0.1501
naïve Bayes	0.5331 ± 0.1221	0.5585 ± 0.0910

Table 13 shows the performances of the three classifiers on CSBSG data set. The similar conclusion as on ISBSG data set can be drawn on CSBSG data set. However, the performances of three classifiers on CSBSG data set are worse than those on ISBSG data set. The average of overall accuracies of the three techniques without (with) imputation on CSBSG data set is decreased by 6.95% (6.66%) using that on ISBSG data set as the baseline. We also explain this outcome as the lower quality of CSBSG data set than that of ISBSG data set.

We can see from Tables 12 and 13 that, in both ISBSG and CSBSG data sets, all the three supervised learning techniques have not produced a favorable classification on software efforts using project attributes. The best performance that was produced by BPNN is with the accuracy around 60%. The accuracy as 60% is meaningless for software effort prediction in most cases because, that means that at the probability 0.4, the prediction results fall beyond the range of each effort class. Combined with the results of effort prediction from unsupervised learning, we draw that the predictability of software effort using supervised learning techniques is not acceptable by software industry, either.

4   Threads to Validity

The threats to external validity primarily include the degree to which the attributes of projects in ISBSG and CSBSG data set have exactly captured the characteristics of software projects in real practice. For data quality, we only extracted a small portion of data samples from ISBSG and CSBSG data sets. We hope these projects are representative of the population of software projects in these two data sets. These threats could be reduced by more experiments on more data sets of software efforts in future work. The threats to internal validity are instrumentation effects that can bias our results. The uncertainty of values of attributes, the ambiguity of software efforts of projects, and the unbalanced distribution of projects with respect to attributes in the data sets might cause such effects. To reduce these threats, we manually inspected the software projects and their values of attributes and evaluated the reliability of the data for each project. One threat to construct validity is that our experiments involve large amount of data preprocessing, hoping that the preprocessed data can still precisely capture the characteristics of original software projects.

5   Related Work

Srinivasan and Fisher [19] used decision tree and BPNN to estimate software development effort. COCOMO data with 63 historical projects was used as the training data and Kremer data with 15 projects was used as testing data. They reported that decision tree and BPNN are competitive with traditional COCOMO estimator. However, they pointed out that the performances of machine learning techniques are very sensitive to the data on which they were trained. [17] compared three estimation techniques as BPNN, case-based reasoning and regression models using Function Points as the measure of system size. They reported that neither of case-based reasoning and regression model was favorable in estimating software efforts due to the considerable noise in the data set. BPNN appears capable of providing adequate estimation performance (with MRE as 35%) nevertheless its performance is largely dependent on the quality of training data as well as the suitability of testing data to the trained model. Of all the three methods, a large amount of uncertainty is inherent in their performances. In both [17] and [19], a serious problem confronted with effort estimation using machine learning techniques is that huge uncertainty involved in the robustness of these techniques. That

is, model sensitivity and data-dependent property of machine learning techniques hinder their admittance by industrial practice in effort prediction. These work as well as [22] motivates this study to investigate the effectiveness of a variety of machine learning techniques on two different data sets.

Park and Baek [18] conducted an empirical validation of a neural network model for software effort estimation. The data set used in their experiments is collected from a Korean IT company and includes 148 IT projects. They compared expert judgment, regression models and BPNN with different input variables in software effort estimation. They reported that neural network using Function Point and other 6 variables (length of project, usage level of system development methodology, number of high/middle/low level manpower and percentage of outsourcing) as input variables outperforms other estimation methods. However, even in the best performance, the average MRE is nearly 60% with standard deviation more than 30%. This result makes it is very hard that the method proposed in their work can be satisfactorily admitted in practice. For this reason, a validation of machine learning methods is necessary in order to shed light on the advancement of software effort estimation. This point also motivates us to investigate the effectiveness of machine learning techniques for software effort estimation and the predictability of software effort using machine techniques.

[20] proposed a neuron-genetic approach to predict software development effort while the neural network is employed to construct the prediction model and genetic algorithm is used to optimize the weights between nodes in the input layer and the nodes in the output layer. They used the same data sets as that was used in Srinivasan and Fisher [19] and reported that the neuron-genetic approach outperforms both decision tree and BPNN. However, they also reported that local minima and over fitting deteriorate the performance of the proposed method in some cases, even make it a poorer predictor than traditional estimator as COCOMO [21]. The focus of our study is not to propose a novel approach to software effort estimation but to extensively review the usefulness of machine learning techniques in software effort estimation. That is, to how much extent the typical machine techniques can accurately estimate the effort of a given project using historical data.

6   Concluding Remarks

In this paper, we conducted a series experiments to investigate the predictability of software effort using machine learning techniques. With ISBSG and CSBSG data sets, unsupervised learning as k- medoids clustering is used to cluster software projects with respect to efforts and, supervised learning as J48 decision tree, BPNN and naive Bayes are used to classify the projects. Our assumption for this investigation is that the efforts of software projects can be deduced from the values of other attributes in historical data and projects with similar values on attributes other than effort will also have approximately equivalent efforts.

The experimental results demonstrate that neither unsupervised nor supervised learning techniques can provide software effort prediction with a favorable model. Despite of this fact, Kulzinsky coefficient has produced the best performance in similarity

measure for unsupervised learning and, BPNN has produced the best performance among the examined supervised learning techniques. Moreover, the MINI imputation can improve data quality and improve effort prediction significantly.

References

1. Boehm, B., Abts, C., Brown, A., Chulani, S., Clark, B., Horowitz, E.: Software Cost Estimation with COCOMO II. Prentice Hall, New Jersey (2001)
2. Pendharkar, P., Subramanian, G., Roger, J.: A Probabilistic Model for Predicting Software Development Effort. IEEE Transactions on Software Engineering 31(7), 615–624 (2005)
3. Jorgensen, M.: A Review of Studies on Expert Estimation of Software Development Effort. Journal of Systems and Software 70, 37–60 (2004)
4. Fairley, R.: Recent Advances in Software Estimation Techniques. In: Proceedings of International Conference on Software Engineering, pp. 382–391 (1992)
5. Yang, Y., Wang, Q., Li, M.: Process Trustworthiness as a Capability Indicator for Measuring and Improving Software Trustworthiness. In: Wang, Q., Garousi, V., Madachy, R., Pfahl, D. (eds.) ICSP 2009. LNCS, vol. 5543, pp. 389–401. Springer, Heidelberg (2009)
6. Korte, M., Port, D.: Confidence in Software Cost Estimation Results based on MMRE and PRED. In: Proceedings of PROMISE 2008, pp. 63–70 (2008)
7. He, M., Li, M., Wang, Q., Yang, Y., Ye, K.: An Investigation of Software Development Productivity in China. In: Wang, Q., Pfahl, D., Raffo, D.M. (eds.) ICSP 2008. LNCS, vol. 5007, pp. 381–394. Springer, Heidelberg (2008)
8. Krupka, E., Tishby, N.: Generalization from Observed to Unoberserved Features by Clustering. Journal of Machine Learning Research 83, 339–370 (2008)
9. Theodoridis, S., Koutroumbas, K.: Pattern Recognition, 3rd edn. Elsevier (2006)
10. Gan, G., Ma, C., Wu, J.: Data Clustering, Theory, Algorithmsm, and Applications. In: ASA-SIAM Series on Statistical and Applied Probability, pp. 78–78 (2008)
11. Song, Q., Shepperd, M.: A new imputation method for small software project data sets. Journal of Systems and Software 80, 51–62 (2007)
12. Zhou, Z., Tang, W.: Clusterer ensemble. Knowledge-Based Systems 19, 77–83 (2006)
13. Steinbach, M., Karypis, G., Kumar, V.: A comparison of document clustering techniques. In: Proceedings of KDD-2000 Workshop on Text Mining, pp. 109–119 (2000)
14. Quinlan, J.: Programs for Machine Learning, 2nd edn. Morgan Kaufmann Publishers (1993)
15. Rumelhart, D., Hinton, G., Williams, J.: Learning internal representations by error propagation. In: Proceedings of Parallel Distributed Processing, Exploitations in the Microstructure of Cognition, pp. 318–362 (1986)
16. Duda, R., Hart, P., Stork, D.: Pattern Classification, 2nd edn. John Wiley & Sons (2003)
17. Finnie, G., Wittig, G.: A Comparison of Software Effort Estimation Techniques: Using Function Points with Neural Networks, Case-Based Reasoning and Regression Models. Journal of Systems and Software 39, 281–289 (1997)
18. Park, H., Baek, S.: An empirical validation of a neural network model for software effort estimation. Expert System with Applications 35, 929–937 (2008)

19. Srinivasan, K., Fisher, D.: Machine Learning Approaches to Estimating Software Development Effort. IEEE Transactions on Software Engineering 21(2), 126–137 (1995)
20. Shukla, K.: Neuro-genetic prediction of software development effort. Information and Software Technology 42, 701–713 (2000)
21. Boehm, B.: Software Engineering Economics. Prentice Hall, New Jersey (1981)
22. Prietula, M., Vicinanza, S., Mukhopadhyay, T.: Software-effort estimation with a case-based resoner. Journal of Experimental & Theoritical Artificial Intelligence 8, 341–363 (1996)
23. Jorgensen, M., Shepperd, M.: A Systematic Review of Software Development Cost Estimation Studies. IEEE Transactions on Software Engineering 33(1), 33–53 (2007)
24. Zhang, W., Yang, Y., Wang, Q.: Handling missing data in software effort prediction with naive Bayes and EM algorithm. In: Proceedings of International Conference on Predictive Models in Software Engineering, vol. 4 (2011)

Modularizing Different Responsibilities into Separate Parallel Hierarchies

Francisco Ortin and Miguel Garcia

Computer Science Department, University of Oviedo, 33007, Oviedo, Spain
{ortin,be37378}@uniovi.es

Abstract. When tangled inheritance hierarchies lead to code duplication, the *Tease Apart Inheritance* "big" refactoring is commonly used to create two parallel hierarchies, using delegation to invoke one from the other. Under these circumstances, the root class of the refactored hierarchy must be general enough to provide all its services to the other hierarchy, leading to meaningless interfaces that violate the Liskov substitution principle. In order to avoid this limitation, we propose a behavioral design pattern that allows the modularization of different responsibilities in separate hierarchies that collaborate to achieve a common goal. With this design, it is possible to use the specific interface of each class in the parallel hierarchy, without needing to define all the methods provided by every class in the hierarchy, and hence not violating the Liskov substitution principle. The proposed design is type safe and avoids the use of dynamic type checking and reflection; at compile time, the type system ensures that no type error will be produced dynamically.

Keywords: Design patterns, refactoring, software design, parametric polymerphism, generics.

1   Introduction

The *Tease Apart Inheritance* is a "big" refactoring that separates a tangled inheritance hierarchy that has different responsibilities in distinct (commonly) parallel hierarchies that use delegation to invoke from one to the other [7]. Each hierarchy has its own responsibility, and all together collaborate to solve a problem. The delegation used to make different hierarchies collaborate is implemented with an association between the root classes of each hierarchy. This commonly involves too general interfaces that are not detailed enough to be used by the parallel hierarchy. This limitation is more evident with the classes below in the hierarchy, where the required interface is even more specific.

As an example, consider building a retargetable compiler [1] for a high-level object-oriented programming language. Once the *Abstract Syntax Tree* (AST) has been built by the parser and semantic (contextual) analysis has been performed over the AST [2], it will be necessary to generate code for different platforms. We want the compiler of our high-level programming language not only to generate low-level code, but also to translate it to other high-level programming languages.

Similar to semantic analysis, code generation could be implemented using the *Visitor* design pattern [3], traversing the AST –built using the *Composite* design pattern [3]– to generate the target code [4]. As shown in Figure 1, source code for different programming languages can be generated with different *Visitor* classes. Common strategies of high-level code generation can be factored out into a common `VisitorHighLevelCG` superclass using the *Template Method* design pattern [3]. Since many high-level programming languages share similar features, the AST traversal for this kind of languages could be expressed with methods in the `Visitor-HighLevelCG` class. These methods can call all the abstract methods defined in their hierarchy level to implement the template algorithms that generate high-level source code. The same generalization can be done for low-level programming languages (and even for all the target languages, using the `VisitorCG` class). For instance, the *Java Virtual Machine* (JVM) assembly code [5] is quite similar to the Microsoft Intermediate Language (MSIL) [6] because both are based on abstract stack machines. Common code generation templates for both languages could be placed in the `VisitorLowLevelCG` class.
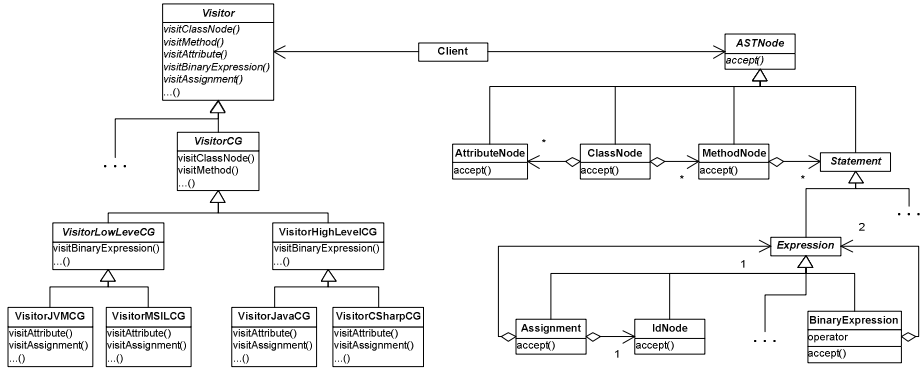
Fig. 1. Using the *Visitor* and *Composite* design patterns to generate code for multiple languages

A benefit of the inheritance hierarchy in the left part of Figure 1 is that common strategies to translate the source code to every target language could be placed in the `VisitorCG` class. For instance, generating the code of a class to Java, C#, JVM and MSIL could be defined as generating the code of its methods and fields (`visitClassNode`). This benefit is obtained with each level of the hierarchy (e.g., high-level or low-level target language). However, since every target language has a different instruction set, this polymorphic behavior needs to be specialized with the instruction set of each particular target language. In order to make the code maintainable, a parallel hierarchy of `CodeGeneration` classes could be defined to generate the specific code of each target language. This is, precisely, the *Tease Apart Inheritance* "big refactoring" [7]. The *Visitor* classes will use the classes of the parallel `CodeGeneration` hierarchy to generate different target languages (Figure 2).
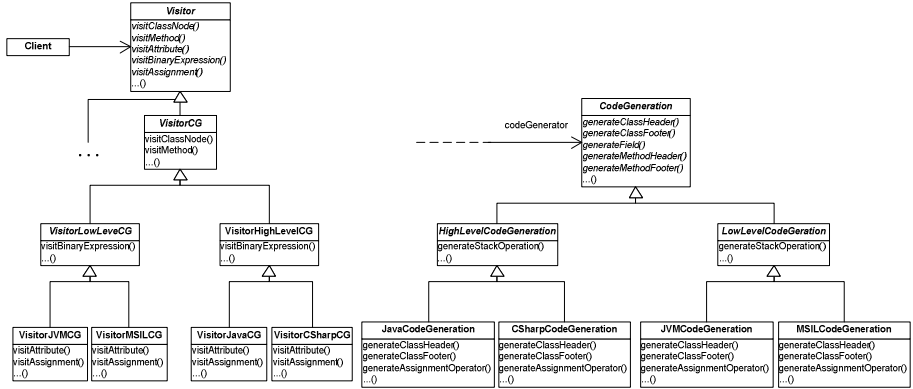
Fig. 2. Tease Apart Inheritance refactoring

Each *Visitor* class relies on a corresponding `CodeGeneration` class to achieve its objective. Each *Visitor* class traverses the AST tree, calling the parallel `CodeGeneration` class to generate the code of a particular target language. Dynamic binding in both hierarchies is used to override all the abstract operations used in the general template algorithms, defining the particular implementation of each specific code generation operation for every target language. For instance, the `generateOpenClass` method of the code generation hierarchy is overridden in Java, JVM, C# and MSIL code generators to describe how a class must be declared in each programming language. This generalization makes it possible to implement the `visitClassNode` method with the simple Java code in Figure 3. It is worth noting that this is the template of a general algorithm. If a new target language needs to be generated, and it does not follow this template, dynamic binding can be used to override the `visitClassNode` method for a particular *Visitor* subclass.

```java
public abstract class VisitorCG extends Visitor {
 public void visitClassNode(ClassNode node) {
   this.codeGenerator.generateClassHeader(node);
   for(FieldNode field : node.getFields())
    this.codeGenerator.generateField(field);
   for(MethodNode method : node.getMethods()) {
    this.codeGenerator generateMethodHeader(method);
    method.accept(this);
    this.codeGenerator generateMethodFooter(method);
   }
   this.codeGenerator.generateClassFooter(node);
  }
}
```

Fig. 3. Implementation of the `visitClassNode` method in the `VisitorCG` class

The benefits of the generalization offered by polymorphism are counteracted by the necessity of recovering the specific interface of a particular `CodeGeneration` class. As an example, we can think about how to generate the code for an assignment expression. The *Visitor* design pattern traverses the AST until the `visitAssignment` is reached. The templates for generating assignment expressions to Java and the JVM

are different. The former uses infix syntax, whereas the latter generates the code of the right-hand side of the assignment first, followed by a store statement indicating the index of the local variable (Figure 5). This difference requires the `visitAssignment` in the `VisitorJVMCG` class to invoke the specific generateStore method in the `JVMCodeGeneration` class, whereas the same method in `VisitorJavaCG` should call to the `generateAssignmentOperator` method in `JavaCodeGeneration`. Therefore, it is required to recover the specific interface of the corresponding code generation class in the parallel hierarchy. Notice than adding these specific methods to the whole code generation hierarchy might produce a design difficult to understand, because some methods are meaningless for specific classes, violating the Liskov substitution principle [8].The main contribution of this paper is the description of a design pattern that provides a way to make two parallel hierarchies collaborate to solve a problem, recovering the specific interfaces of classes in the corresponding hierarchy. The usage of both generalized and specific interfaces in a class hierarchy is obtained without violating the type safety offered by many statically typed programming languages. This approach can be used to solve the expression problem [16] and together with other design patterns such as *Composite*, *Template* or *Visitor* [3].

2   Related Work

The necessity of recovering the specific interfaces of two parallel hierarchies was detected in the *expression problem*, first named by Philip Wadler in 1998 [16]. The issue was to obtain a modular extensibility of data structures. This problem was then revisited by Mads Torgersen that used the Java F-bound polymorphism to recover the type of inherited associations [17]. The solution was applied to the *Composite* and *Visitor* design patterns [3] instead of going into this topic and identifying it as a design pattern.

In [18], the recovery of inherited associations is tackled using the Scala programming language. They emphasize the significance of solving this problem in a type safe way (without runtime type errors), becoming really interesting when it is exposed to a type system that ensures the safe execution of the code. For this purpose, a solution using parameterized classes (generics) is provided. They also identify the possibility of using Scala's virtual types: a mechanism similar to parameterized classes but, instead of giving the types as parameters, a class contains a type variable [18].

Erik Ernst introduced the notion of higher order hierarchies to represent hierarchies of hierarchies [19]. The idea is to define a hierarchy that could be later extended and reused in a type safe way. Although the idea seems to be suitable to model parallel hierarchies, higher-order hierarchies do not allow the specialization of inherited associations in parallel hierarchies.

The structure of this design pattern has been previously recognized as a *Big Refactoring* by Fowler and Beck [7]. It is applied to solve the problem of tangled inheritance [7]. However, they do not describe how both hierarchies can collaborate to obtain a common objective, using the specific interfaces in each hierarchy level; only the generalized polymorphic behavior is described.
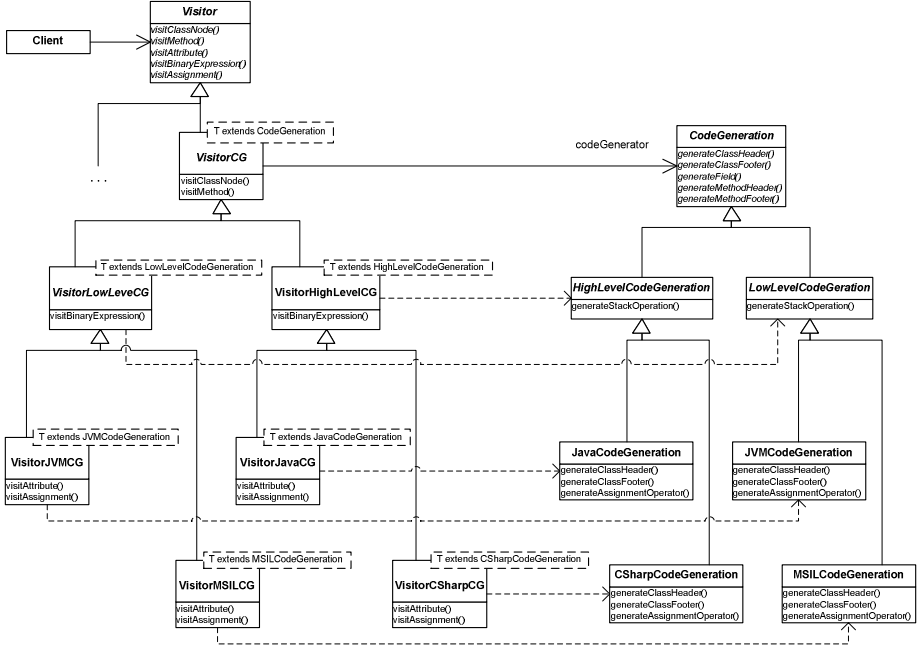
Fig. 4. Structure of the *Parallel Hierarchies* design pattern

Parallel Hierarchies has some relation with several design patterns in the classic catalog [3]. *Parallel Hierarchies* classes commonly appear from refactoring behavioral design patterns that use a hierarchy for their purpose, like *Template*, *Visitor* or *Interpreter*. In the *Memento* pattern, *originators* and *mementoes* often form parallel hierarchies. If recursive polymorphic methods are added to the *Composite* design pattern, their implementation can rely on a parallel hierarchy. Finally, the *Abstract Factory* design pattern creates families of related objects, to use them later separately. The *Parallel Hierarchies* design pattern could be applied when the specific interfaces of these objects need to be used together for a particular purpose.

3   The Parallel Hierarchies Design Pattern

The proposed design pattern, called *Parallel Hierarchies*, allows the recovery of the specific interface used to connect the different hierarchies. Although this can be done with different programming language techniques (see Section 3.4), the use of generics (parametric polymorphism) is appropriate when the implementation language is statically typed. Generics offers the reliability of type safety, plus the runtime performance improvement obtained by avoiding the use of runtime reflection [9]. C++ implements (unbounded) parametric polymorphism (generics), whereas C# and Java offers F-bounded polymorphism [10] (also known as constrained genericity). Both kinds of parametric polymorphism can be used to implement the *Parallel Hierarchies* design pattern. Figure 4 shows how to use it in our motivating example.

Both hierarchies are connected with one association between the VisitorCG class and CodeGeneration –its multiplicity varies depending on the problem. Each visitor object has a code generator attribute to generate code in a particular pro-gramming language. However, the type of this attribute would be the corresponding one in the parallel hierarchy if generics is used, achieving the recovery of the whole particular interface of the corresponding code generation class. If the language offers F-bounded polymorphism (e.g., Java or C#), the VisitorCG class will be declared as generic, parameterized with a *T* type –being *T* a subtype of CodeGeneration. The attribute's type will then be *T*, and hence the CodeGeneration interface could be used in VisitorCG. This constraint (bound) of the *T* type is specialized in all the subclasses of VisitorCG. For instance, in the VisitorHighLevelCG class, *T* must be a subtype of HighLevelCodeGeneration, and a subclass of VisitorJavaCG in the case of JavaCodeGeneration. This modification on the constraints of *T* is possible when constraints are covariant with respect to the types they are applied to [11] –as happens with Java and C#. Consequently, the visitAssignment method in the VisitorJVMCG class will be able to invoke the generateStore method of theparticular interface of the JVMCodeGeneration class (its interface has been recovered) as shown in Figure 5.

```
public class VisitorJVMCG <T extends JVMCodeGeneration>
                    extends VisitorLowLevelCG<T> {
 @Override
 public void visitAssignment(AssignmentNode node) {
  node.getSecondOperand().accept(this);
  this.codeGenerator.generateStore(node.getFirstOperand().getIndex(),
                                node.getFirstOperand().getType());
 }
 //…
}
```

Fig. 5. Implementation of the visitAssignment-Node of the VisitorJVMCG class

3.1   Structure

The static structure of the proposed design pattern is shown in Figure 6, where the participants can be identified:

• *Template* hierarchy. Classes in this hierarchy describe templates of algorithms, defining their structure in the classes above in the hierarchy and the specific primitive operations in subclasses. This hierarchy is intended to have only one clear responsibility, delegating other possible responsibilities in parallel *Interface* hierarchies. The elements of the *Template* hierarchy are:

— *AbstractTemplate* (VisitorCG, VisitorHighLevelCG and VisitorLowLevelCG)

o Defines the common structure of general algorithms. Each general algorithm is implemented in abstract TemplateMethod methods.

o Declares the interface of abstract primitive operations that are used in general algorithms (PrimitiveOperation1 and PrimitiveOperation2).

○ The implementations of general algorithms make use of the specific interface of the parallel class (`GeneralInterface`).

○ Each general algorithm is implemented using both primitive operations and the methods of the parallel *Interface* classes.

○ (optional) Intermediate *AbstractTemplate* classes (`AbstractTemplateB`) may appear to generalize the structure and behavior of *ConcreteTemplate* classes.

— *ConcreteTemplate* (VisitorJavaCG, VisitorJVMCG, VisitorCSharpCG and VisitorMSILCG)

○ Implements the primitive operations to carry out subclass-specific operations of each general algorithm (`PrimitiveOperation1` and `PrimitiveOperation2`).

○ The implementation of its specific primitive operations may use the concrete interface of its parallel *Interface* classes.

○ (optional) General algorithms (`TemplateMethod`) may be overridden in specific `ConcreteTemplate` classes if the default implementation is not appropriate for a particular case.
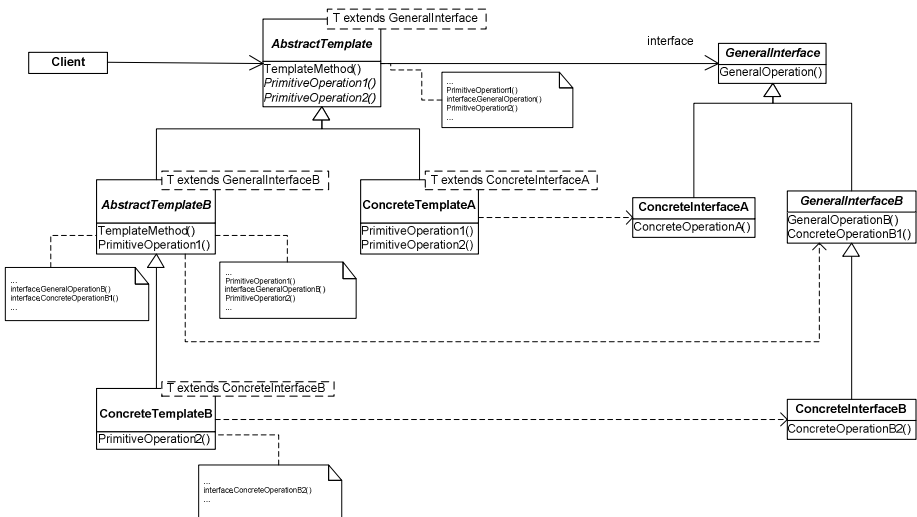
Fig. 6. Structure of the *Parallel Hierarchies* design pattern

• *Interface* Hierarchy. This parallel hierarchy modularizes a responsibility that the *Template* hierarchy may require to achieve its aim. It can also be seen as a set of helper classes used by the *Template* hierarchy to accomplish its objective. In this design pattern, multiple parallel *Interface* hierarchies may be used by the same *Template* abstraction.

— *GeneralInterface* (CodeGeneration, HighLevelCodeGeneration and LowLevelCodeGeneration).

o Defines operations common to all the classes in the *Interface* hierarchy (`GeneralOperation`).

o (optional) Implements default behavior of these general operations, which may be overridden in its subclasses.

o (optional) If intermediate *AbstractTemplate* classes are defined in the parallel *Template* hierarchy, another intermediate *Interface* class will define operations common to that *Template* hierarchy level (`ConcreteOperationB1`).

— ConcreteInterface   (JavaCodeGeneration,   CSharpCodeGeneration, JVMCodeGeneration and MSILCodeGeneration)

o Implements concrete operations applicable only to its specific level in the hierarchy (`ConcreteOperationA` and `ConcreteOperationB2`).
o Overrides general default operation implementations for a particular `ConcreteInterface` class.

• Client

— Invokes the `TemplateMethod` methods of the *AbstractTemplate* class.

3.2   Collaborations

The sequence diagram in Figure 7 illustrates the collaborations between a client, a concrete template object, and its corresponding interface instance.
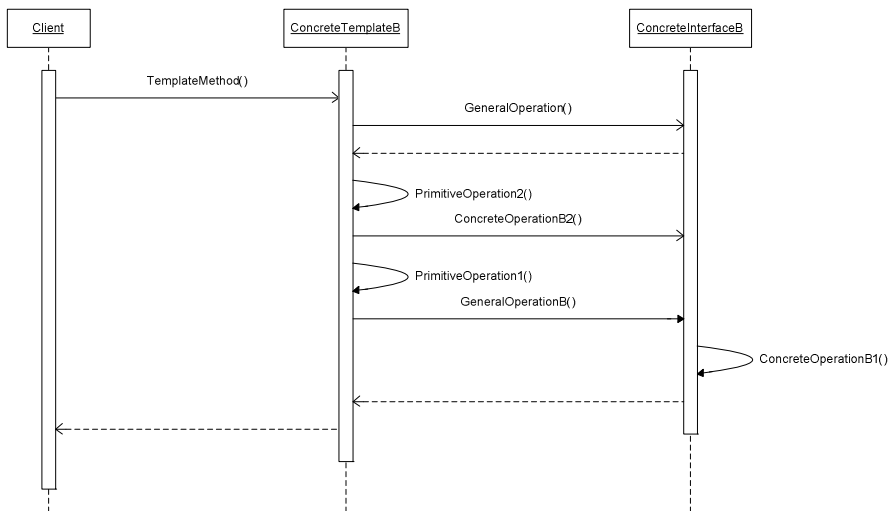
Fig. 7. Example collaboration sequence diagram

• A client that uses the *Parallel Hierarchies* design pattern must create a `ConcreteTemplate` object and call one of the `TemplateMethod` methods this class implements.

- The `TemplateMethod` invokes the `GeneralOperation` on its corresponding *Interface* class.

- To respond to the `TemplateMethod` message, the *Template* object also makes use of its polymorphic primitive operations.

- Thanks to dynamic binding, the `PrimitiveOperation` request is associated to the `ConcreteTemplate` object created by the client. At this moment, the particular *Template* object recovers the whole interface of its parallel class and invokes a specific method of its corresponding *Interface* type (e.g., `ConcreteOperationB2`).

- `GeneralOperation` could be overridden in an intermediate level of the *Interface* hierarchy, and its execution may call to concrete operations of this intermediate level (e.g., the `GeneralOperationB` method could make use of the `ConcreteOperationB1` method).

3.3   Consequences

The use of the Parallel Hierarchies design pattern has the following benefits and limitations:

1. *Parallel Hierarchies Gathers Related Operations and Separates Unrelated Ones*. Related behavior is not spread over the classes defining the template hierarchy. Classes above in the hierarchy define the global structure of the algorithms, whereas particular and primitive cases are implemented as operations in the leaf classes. At the same time, the *Template* classes only define the skeleton algorithms in a problem; other responsibilities will be factored out into parallel *Interface* hierarchies. In our motivating example, the *Template* classes aim at traversing ASTs of source programs (describing both the global algorithms valid to every target language and the particular cases). All the issues concerned with writing code for a particular language are implemented in the code generation hierarchy.

2. *Parallel Hierarchies Makes Adding New Interface Hierarchies Easy*. The implementation of the methods in the *Template* hierarchy could make use of more than one hierarchy of *Interface* classes. At the same time, it is also possible to use another different *Interface* hierarchy without changing the implementation of *Template* classes. For instance, the code generation hierarchy could be replaced by another one that creates an intermediate-representation of programs in memory, to execute it later by means of the *Interpreter* design pattern [3]. For this purpose, the *Interface* hierarchy should first be defined with interfaces, following the *Bridge* design pattern (Figure 8). In that case, each class of the *Interface* hierarchy should implement its corresponding interface.

3. *Type Safety and Runtime Performance*. Although there are different possible implementations of the *Parallel Hierarchies* design pattern (see Section 3.4), the one that uses parametric polymorphism (generics) detects type errors (those regarding to the usage of the `interface` attribute) at compile time. However, if reflection is used instead, type errors will be detected at runtime. Moreover, runtime performance is commonly increased because no dynamic type checking needs to be done [9].

4. *Supporting New Kinds of Templates is Difficult*. The addition of a new class to the *Template* hierarchy in order to include new particular behavior is not a trivial task. That is because each element in the *Template* hierarchy should have a corresponding

element on the parallel *Interface* one. Therefore, a new *Interface* class has to be created, overriding all the appropriate methods to help the new *Template* class achieve its purpose. In our example, if we want to translate the source language to a new target language, two new `Visitor` and `CodeGeneration` classes should be added. At the same time, specific operations of the `Visitor` class and concrete operations of `CodeGeneration` should also be implemented.

5. *Coupling between Template and Interface Hierarchies*. Although one benefit of this design pattern is that the *Template* classes can use the whole interface of the corresponding *Interface* ones, the use of this particular interface produces a coupling between the *Template* classes and the implementation of the *Interface* ones. This limitation can be lessened by applying the *Bridge* design pattern as stated in the second consequence of the *Parallel Hierarchies* design pattern, using the structure shown in Figure 8.
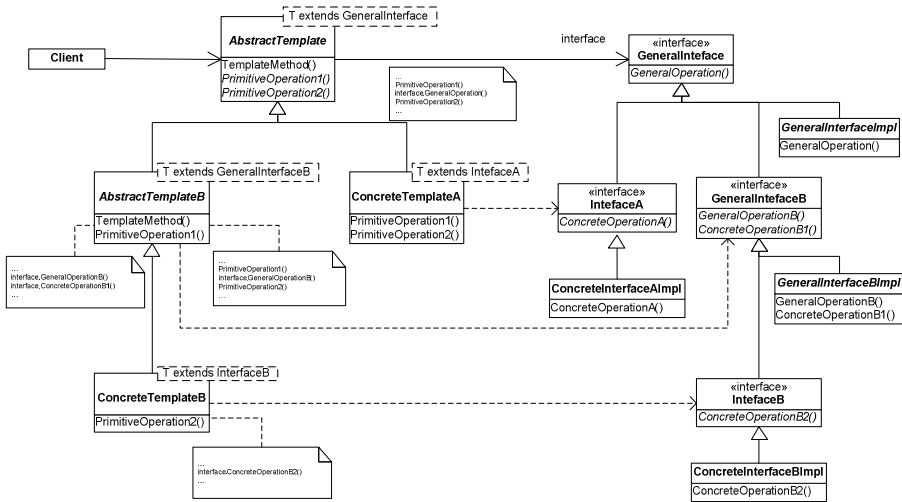
Fig. 8. Decoupling the *Interface* abstraction from its implementation

3.4   Implementation

Some implementation issues of the *Parallel Hierarchies* pattern are worth noting:

1. *Use of Access Control*. The primitive operations defined in the *Template* hierarchy should be declared as protected. This ensures that they are only called by the general template methods. At the same time, the *Template* classes would reduce their interface, making it easier to use for the programmer. If no default implementation can be provided for primitive operations, they should be declared as abstract. When the parallel *Interface* hierarchy has been factored out from the *Template* one, and we do not want to it be used by another component, it could be useful to make it private except for the *Template* classes. This feature is not directly supported by most programming languages. In Java 6, for example, both hierarchies must be placed in the same package, and the *Interface* classes should not be declared as public. With the

superpackages feature to be included in Java 7 [12], the two hierarchies could be implemented in different packages. In C++, friend classes can be used for this purpose; C# provides assembly-level information hiding.

2. *Naming Conventions*. Since this design pattern uses two (or more) parallel hierarchies, many different classes will be required and they will be connected with other corresponding classes in the same hierarchy level. Therefore, following a naming convention makes the code easier to read and more understandable. In our example (see Section 4), being *L* a particular target language, we define an LCodeGeneration class for each corresponding VisitorLCG class in the *Interface* hierarchy.

3. *Favor the Use of Generics*. Parametric polymorphism is a statically typed programming language feature that promotes type safety and allows for efficient implementation. Dynamic casts can also be used to check whether the type of the associated *Interface* object has the appropriate type or not. Source code in Figure 9 is an equivalent implementation of the code shown in Figure 5 that uses dynamic type coercion.

```java
public class VisitorJVMCG extends VisitorLowLevelCG {
 private JVMCodeGeneration getCodeGenerator() {
  if (!(this.codeGenerator instanceof JVMCodeGeneration))
   throw new IllegalStateException("The attribute codeGenerator
                        does not have the appropriate type." );
  return (JVMCodeGeneration)this.codeGenerator;
 }
 @Override
 public void visitAssignment(AssignmentNode node){
  node.getSecondOperand().accept(this);
  this.getCodeGenerator().generateStore(node.getFirstOperand()
             .getIndex(), node.getFirstOperand().getType());
 }
 //…
}
```

Fig. 9. Sample implementation using dynamic type coercion

The problem of this approach is twofold. The first one is that both the instanceof operation and the type cast are evaluated at runtime. Therefore, if some error occurs, it will occur at runtime, reducing the robustness of this approach. The second drawback is the runtime performance penalty that is caused by runtime type inspection [9].

4. *Minimize the General Interface of the Interface hierarchy*. Since a class should only define operations that are meaningful to its subclasses [8], only those messages that are meaningful for every class of the *Interface* hierarchy should be placed in the GeneralInterface class. Recall that the *Parallel Hierarchies* design pattern recovers the specific interface of each *Interface* class. This feature allows reducing the interface of these classes to the exact set of messages that are meaningful to them.

5. *Use the Bridge pattern to decouple hierarchies*. As mentioned in Section 3.3, the *Template* hierarchy is coupled with the implementation of the *Interface* one. It could be necessary to add new implementations of *Interface* classes, or to support different implementations for each *Template* object at the same time –e.g., following the *State*

design pattern [3]. If either of these scenarios occurs, the *Bridge* design pattern [3] should be used in the *Interface* hierarchy, resulting in the pattern structure shown in Figure 8.

3.5   Applicability

Use the parallel hierarchies design in either of the following cases:

• The *Template Method* design pattern is suitable, but it is difficult to generalize a common interface for all the classes in the hierarchy. Although there are methods common to every class, others are only applicable to particular ones.

• More than one responsibility can be identified in a hierarchy and these new responsibilities can be factored out and integrated in another parallel hierarchy. In fact, this is the *Tease Apart Inheritance* "big" refactoring identified by Fowler and Beck [7].

• A problem can be solved with a combination of general algorithms plus specific operations of different particular types.

4   Sample Code

We will follow the motivating example of implementing a retargetable compiler for a high-level object-oriented programming language. A fragment of the AST is shown in Figure 1. The traversal of this AST is done with the *Visitor* design pattern [3]. The specific visitors for code generation play the *Template* role of the *Parallel Hierarchies* design pattern. Since we are interested in generating code for different programming languages, the Java `VisitorCG` class shown in Figure 10 represents the `AbstractTemplate` class of the *Parallel Hierarchies* structure.

Figure 10 only shows one `visit` method of the *Visitor* design pattern. The implementation of this method defines the general `classNode` code-generation template for every programming language. If this template is not appropriate for a specific target language, its corresponding *Visitor* subclass will override it.

```java
public class abstract class VisitorCG <T extends CodeGeneration>
                          extends Visitor {
 protected T codeGenerator;
 public VisitorCG(T codeGenerator){this.codeGenerator=codeGenerator;}
 @Override
 public void visitClassNode(ClassNode node) {
  this.codeGenerator.generateClassHeader(node);
  for(FieldNode field:node.getFields())
   this.codeGenerator.generateField(field);
  for(MethodNode method:node.getMethods()){
   this.codeGenerator.generateMethodHeader(method);
   method.accept(this);
   this.codeGenerator.generateMethodFooter(method);
  }
  this.codeGenerator.generateClassFooter(node);
 }
 //…
}
```

Fig. 10. `VisitorCG` sample code

The `visit` method makes use of two different kinds of operations: methods of the Interface (`CodeGeneration`) hierarchy, and messages of its own hierarchy (`accept` messages). The `accept` method is a double-dispatch implementation that actually represents indirect calls to visit methods. All these `visit` methods play the role of `TemplateMethod` in the *Parallel Hierarchies* design pattern. Figure 10 shows the template that generates the code of a `classNode` relying on the templates that generate its methods and fields.

The other operations that the `visit` methods use are the messages offered by its corresponding parallel class. The `codeGenerator` field reference provides these services. For the `VisitorCG` class, only the methods in the `CodeGeneration` class can be used. This means that the general template for all the target languages can only use the operations of code generation defined for every target language. Source code in Figure 11 shows the `CodeGeneration` class. Its abstract methods identify the operations applicable to every target programming language, but they are not concrete enough to provide a default implementation.

```java
public abstract class CodeGeneration {
 protected FileWriter file;
 public CodeGeneration(String filename) {
  file = new FileWriter(filename);
 }
 public abstract void generateClassHeader(ClassNode klass);
 public abstract void generateClassFooter(ClassNode klass);
 public abstract void generateField(FieldNode field);
 public abstract void generateMethodHeader(MethodNode method);
 public abstract void generateMethodFooter(MethodNode method);
 //…
}
```

Fig. 11. `CodeGeneration` sample code

Since both the JVM and the MSIL are abstract stack machines languages, we can factor out common code generation operations for both languages in a new `LowLevelCodeGeneration` class (Figure 12).

```java
public abstract class LowLevelCodeGeneration extends CodeGeneration {
 public LowLevelCodeGeneration(String filename) { super(filename); }
 public abstract void generateStackOperation(String operator,
                                             TypeNode type);
 //…
}
```

Fig. 12. `LowLevelCodeGeneration` sample code

The specific `generateStackOperation` method offered by `LowLevelCode-Generation` can be used by the `VisitorLowLevelCG` class. Therefore, it is possible to write the `visitBinaryExpresion` shown in Figure 13: for every stack-based abstract machine, most binary expressions can be generated writing the code for the first and second operands, followed by the operation (postfix notation). It is worth noting that, thanks to Java generics, it is type safe to pass the specific `generateStackOperation` message to the inherited `codeGeneration` field,

although it has been declared to be of type "`T extends CodeGeneration`" in the `VisitorCG` class. This shows how the *Parallel Hierarchies* design pattern recovers the original interface of the actual object used in the *Template* hierarchy.

```java
public abstract class VisitorLowLevelCG
             <T extends LowLevelCodeGeneration> extends VisitorCG<T> {
 public VisitorLowLevelCG(T codeGeneration) {super(codeGeneration);}
 public void visitBinaryExpression(BinaryExpressionNode node) {
  // * Postfix notation
  node.getFirstOperand().accept(this);
  node.getSecondOperand().accept(this);
  this.codeGenerator.generateStackOperation(
             node.getOperator(),node.getType());
 }
}
```

Fig. 13. `VisitorLowLevelCG` sample code

The responsibility of the `JVMCodeGeneration` class is to generate JVM code following the *Jasmin assembly* syntax [13]. This class not only overrides methods of the general `CodeGeneration` class (`generateClassHeader` and `generateClassFooter`) and the `LowLevelCodeGeneration` class (`generateStackOperation`), but it also defines its particular interface (`generateStore`) that produces the store JVM instruction). Part of its implementation is shown in Figure 14.

```java
public class JVMCodeGeneration extends LowLevelCodeGeneration {
 // * Methods of the CodeGeneration class
 @Override
 public void generateClassHeader(ClassNode klass) {
  this.file.write(".class " + " "+ klass.getHidingLevel()+
                   " "+ klass.getName() + "\n");
 }
 @Override
 public void generateClassFooter(ClassNode klass) {
  this.file.write(";  end of the " + klass.getName() + " class\n");
 }
 // * Methods of the LowLeveCodeGeneration class
 @Override
 public void generateStackOperation(String operator, TypeNode type ){
  if (!operatorsToJVM.containsKey(operator))
   throw new IllegalArgumentException("Operator '"
                   + operator + "' not defined in the JVM.");
  StringBuilder sb = new StringBuilder();
  sb.append(type.getJVMTranslation());
  sb.append(operatorsToJVM.get(operator));
  this.file.write(sb.toString()+"\n");
 }
 // * Specific methods of the JVMCodeGeneration class
 public void generateStore(int index, TypeNode type ) {
  this.file.write(type.getJVMTranslation()+"store "+index+"\n");
 }
 // …
}
```

Fig. 14. `JVMCodeGeneration` sample code

Finally, it is possible to write the `VisitorJVMCG` class that traverses only the specific nodes for which the default traversal is not appropriate (`visitAssignment` in Figure 15). The assembly syntax of assignments in the JVM is defined as the code that pushes the right-hand expression of the assignment, followed by a `store` instruction whose operand is the index of the variable on left-hand side of the assignment. In the implementation of these specific `visit` methods, any method of the particular interface of the parallel `JVMCodeGeneration` class can be used (`generateStore` is an example of this kind of methods).

```java
public class VisitorJVMCG <T extends JVMCodeGeneration>
                       extends VisitorLowLevelCG<T> {
 public VisitorJVMCG(T codeGeneration) { super(codeGeneration); }
 @Override
 public void visitAssignment(AssignmentNode node) {
  node.getSecondOperand().accept(this);
  this.codeGenerator.generateStore(node.getFirstOperand().getIndex(),
                                   node.getFirstOperand().getType());
 }
 // …
}
```

Fig. 15. `VisitorJVMCG` sample code

5   Conclusions

The association of two (or more) parallel hierarchies to solve a specific problem using delegation is a common design scenario. The problem is that the association in the root classes in the hierarchy provides too general interfaces useless for the classes below in the hierarchy. The *Parallel Hierarchies* design pattern described in this paper provides a type safe solution that could be used whenever the programming language provides either F-bound polymorphism or (unbounded) parametric polymorphism such as Java, C# or C++. The design pattern has been described using the classical sections of structure, collaborations, consequences, Implementation, applicability and sample code [3].

We have successfully used the *Parallel Hierarchies* design pattern in the C# implementation of the *StaDyn* programming language [14,20], to compile the *StaDyn* high-level programming language to MSIL for the CLR, ЯRotor [9,21] and the DLR [15] platforms, as well as produce high-level C# 4.0 source code.

References

1. Hanson, D.R., Fraser, C.W.: A Retargetable C Compiler: Design and Implementation. Addison-Wesley Professional (1995)
2. Appel, A.W.: Modern Compiler Implementation in Java, 2nd edn. Cambridge University Press (2002)

3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Re-usable Object-Oriented Software. Addison Wesley (1994)
4. Watt, D., Brown, D.: Programming Language Processors in Java: Compilers and Interpreters. Prentice Hall (2000)
5. Lindholm, T., Yellin, F.: Java Virtual Machine Specification, 2nd edn. Prentice Hall (1999)
6. ECMA 335, European Computer Manufacturers Association (ECMA). Common Language Infrastructure (CLI), Partition IV: CIL Instruction Set, 4th edn. (2006)
7. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)
8. Liskov, B.: Data Abstraction and Hierarchy. In: Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Orlando, Florida, United States, pp. 17–34 (1987)
9. Ortin, F., Redondo, J.M., Perez-Schofield, J.B.G.: Efficient virtual machine support of runtime structural reflection. Science of Computer Programming 74(10), 836–860 (2009)
10. Canning, P., Cook, W., Hill, W., Walter, O., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, London, United Kingdom, pp. 273–280 (1989)
11. Odersky, M., Wadler, P.: Pizza into Java: Translating theory into practice. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL), Paris, France, pp. 146–159 (1997)
12. JSR 294 Sun Microsystems, JSR 294: Improved Modularity Support in the Java Programming Language (2007), http://jcp.org/en/jsr/detail?id=294
13. Meyer, J.: Jasmin Instructions (1996),
    http://jasmin.sourceforge.net/instructions.html
14. Ortin, F., Zapico, D., Perez-Schofield, J.B.G., Garcia, M.: Including both Static and Dynamic Typing in the same Programming Language. IET Software 4(4), 268–282 (2010)
15. Hugunin, J.: Bringing dynamic languages to .NET with the DLR. In: Proceedings of the Symposium on Dynamic Languages, Montreal, Quebec, Canada, p. 101 (2007)
16. Wadler, P.: The expression problem. Posted on the Java Genericity Mailing List (1998)
17. Torgersen, M.: The Expression Problem Revisited. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
18. Nielsen, E.T., Larsen, K.A., Markert, S., Kjaer, K.E.: The Expression Problem in Scala. Technical Report, Aarhus University (May 31, 2005)
19. Ernst, E.: Higher-Order Hierarchies. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 303–329. Springer, Heidelberg (2003)
20. Ortin, F., Garcia, M.: Union and intersection types to support both dynamic and static typing. Information Processing Letters 111(6), 278–286 (2011)
21. Redondo, J., Ortin, F., Cueva, J.: Optimizing Reflective Primitives of Dynamic Languages. International Journal of Software Engineering and Knowledge Engineering 18(6), 759–783 (2008)

Steering through Incentives in Large-Scale Lean Software Development

Benjamin S. Blau[1], Tobias Hildenbrand[1], Rico Knapper[2], Athanasios Mazarakis[2],
Yongchun Xu[2], and Martin G. Fassunge[1]

[1] SAP AG, Dietmar-Hopp-Allee 16, Walldorf, Germany
{benjamin.blau,tobias.hildenbrand,martin.georg.fassunge}@sap.com
[2] Research Center for Information Technology
Haid-und-Neu-Straße 10-14, Karlsruhe, Germany
{yongchun.xu,knapper,mazarakis}@fzi.de

Abstract. The application of lean principles and agile project management techniques in the domain of large-scale software product development has gained tremendous momentum over the last decade. This results in empowerment of individuals which leads to increased flexibility but at the same time sacrifices managerial control through traditional steering practices. Hence, the design of adequate incentive schemes in order to align local optimization and opportunistic behavior with the overall strategy of the company is a crucial activity from a business perspective.

Following an agent-based simulation approach with reinforcement learning, we *(i)* address the question of how information regarding backlog item dependencies is shared within and in between development teams on the product level subject to different incentive schemes. We *(ii)* compare different incentive schemes ranging from individual to team-based compensation. Based on our results, we are *(iii)* able to provide recommendations on how to design suitable incentive schemes in order to enable a goal-oriented steering of individual behavior in order to support the overall company objectives.

1 Introduction

The application of lean and agile principles in large-scale software product development turns out as non-trivial transition and change management endeavor in most companies [11]. This is partly due to the fact that a simple transfer of known practices from lean manufacturing in other industries cannot be achieved due to differences between production versus product development processes and the nature of knowledge work and immaterial goods—such as software [37,39]. Especially breaking down bigger products to an organization requiring multiple teams and hierarchy levels, dealing with product dependencies, and re-integrating features and functions while keeping the overall market and economics of decisions in mind is yet very challenging in the relatively young software industry [31,28]. As a consequence, phenomena like queued artifacts, delayed product deliveries, and long-tail risks occur [39].

This research aims at gaining a better understanding of the information sharing and motivation mechanics of a complex socio-technical system, such as a large-scale software product development organization. Based on this increased understanding, we want to derive implications for designing the development organization, and issue incentives for the teams in order to *foster overall software product development flow by means of more informed and economic decisions, resulting in a shorter time to market.*

Based on our research goal and the complex, large-scale industrial setting (see section 5), we follow an *agent-based* simulation approach with reinforcement learning. Using this method we *(i)* investigate the information flow in lean large-scale software product development systems in terms of dependency resolution between requirements, user stories, and other software artifacts (cp. [22,47]). In this context, incentives for individuals to share such information are of central importance. Therefore, we *(ii)* furthermore tackle the question of how different types of incentive schemes impact information flow and the overall performance of empowered teams. Based on our simulation results, we *(iii)* provide recommendations on how to design such incentives and how to chose an adequate development structure within an organization. For calibrating our simulation, we rely on three years of experience from one of today's largest lean and agile adoption at SAP AG [41].

The remainder of this paper is structured as follows: Section 2 outlines related research in the context of agile and lean software development. The agent-based simulation methodology and the corresponding field of research is analyzed in Section 3. The basic model underlaying the empirical evaluation is described in Section 4. The simulation, its parametrization and the research hypotheses are specified in detail in Section 5. Evaluation results and their practical implications are discussed in Section 6. Section 7 summarizes our contribution and outlines future work.

2   Related Work

In order to model and understand a complex socio-technical system, such as a multi-level software product development organization, the underlying design principles and processes need to be investigated. In this work, we specifically address the application of lean and agile principles in large software development companies (e.g. [41]). While there is mostly narrative literature on agile principles and Scrum in large-scale enterprise environments "driven by practitioners and consultants" [12, p. 329]—examples include [31,43,28,32,29], there is only little empirical evidence and rigorous research in this field. For instance, there is only little research on the effectiveness and efficiency gains actually achieved by introducing lean and agile principles, Scrum-based project management etc.—in this small set less than 2% exhibit acceptable rigor, credibility, and relevance [15, p. 851], while 75% of these studies only investigated agile projects specifically applying eXtreme Programming (XP, [3,14]).

We close this section by giving the social psychological perspective on teams, their structure, and incentives in team environments.

2.1   Agile Team Practices

The vast majority of research on agile methods and practices focuses on XP [3,4] as team practice and applies a single or multiple case study methodology [56].

Single practices crucial to XP have been examined separately regarding their impact on software quality, e.g. pair programming is said to consume 30% more effort than solo programming [9], resulting in 40-90% fewer defects [55,16,9]. However, with respect to the broad range of agile methods and their increasing prevalence in the software industry [53], there is only very little scientific evidence so far whether or not these models lead to more effectiveness, efficiency, or productivity, respectively, in real-world large-scale development environments [15].

Among the few evidence-based behavioral science contributions [21] on software agility, Lee and Xia [30] investigated the impact of two major agile characteristics (team autonomy and team diversity) on three productivity measures: (1) on-time and (2) on-budget completion as well as (3) functionality provided to customers. Among their findings, it turned out that there are conflicting goals even within the boundaries of one team. Besides these findings, the model exhibits that the dependent productivity variables could only be explained to a degree that leaves substantial room for future behavioral studies.

2.2   Large-Scale Lean and Agile

Lean management or lean thinking – as underlying philosophy and common set of values – as well as lean and agile principles are either already implemented or piloted in many practical scenarios of different scales today, e.g. at Salesforce [18] or SAP [41]. Figure 1 visualizes how specific agile software development practices, such as XP [3], test-driven development (TDD, [5]) and agile project management methods like Scrum [44] build upon agile principles and lean thinking values. While the basic principles and philosophy apply to many industries, some address a specific one more concretely, e.g. Scrum and XP for the software industry.

Based on general principles of lean management [38] and lean thinking as well as basic agile principles [1] and consulting experience, a set of guiding principles and practices for scaling Scrum to larger-scale scenarios evolved, see e.g. [36,38,28,29]. In the same vein, similar large-scale Scrum models have been described by Schwaber [43] and Leffingwell [31]. These ideas on lean management and lean software development have been further elaborated and translated to some practical guidelines based on experience from multiple consulting projects, see e.g. [29]. However, lean software development in large enterprise environments requires scaling team-based approaches such as Scrum (see section 2.1. Nevertheless, first implementation concepts and pilot approaches can be found even for very large-scale software vendors [41]. Hence, empirical research and evidence for complex socio-technical system in the software industry is even more scarce than for team practices (cp. section 2.2 and [15]).

Lean and agile software development is based on lean enterprise characteristics comprising focus on value, synchronization, transparency, and perfection as well as Just-in-Time (JIT) principles such as (one-piece) flow, takted development, customer pull, and zero defects [39].

Combining the lean enterprise perspective with an agile perspective on development teams [1], leads to short iterative development cycles, a uniquely prioritized backlog of requirements and work items, direct customer involvement, as well as tested and potentially shippable software increments.

As a common basis for further studies on agile practices, Conboy has developed a unified definition and formative taxonomy of agility in informations systems development or software engineering, respectively [12, pp. 340]. Such a common definition and/or taxonomy is required to link existing and future contributions in this very interdisciplinary field of research, e.g. from information systems, computer science, organizational science, sociology and psychology.

In context of lean and agile software development, there are to-date only very few related simulation-based contributions, e.g. using a system dynamics approach [9]. Moreover, [35] present potential performance indicators and visualizations for flow simulations (cp. also [39]).

2.3 Social Psychological Aspects

The topic of work groups and teams in organizations is ongoing in social psychology, despite (or maybe because of) the remarkable transformation of organizational structures in the last two decades [27]. Although the term "team" is still used with different meanings, we strike to the following definition: a group (or team) consists of two or more members, they approach together a common goal and they need to cooperate within social interaction [27]. Usually research considers the group development model of Tuckman still as helpful to guide the formation and development of teams [49]. It consists of four different phases:

– The formation phase as the first element, when the forming of a team takes place.
– The storming phase where goal setting and resolving of different problems have to be done.
– The norming phase needs to consolidate the different opinions of the team members.
– The performing phase finally is about doing the work in an efficient way.

The final phase (the adjourning phase, which is about breaking up the team after the achievement of the group goal) is not in the scope of this article. Also it is important to take into account, that teams can evolve and change after a while, e.g. that new members need to be integrated into a team [23].

Motivational Aspects. Research about motivation to work together can be characterized by two different points of view: harms to motivation and incentives for motivation. First we want to show a brief overview of different harms to motivation and later we will discuss the issue of incentives for individuals on team level.

Harms to Motivation to Work Together. Three different harms can be distinguished:

– The sucker effect (each individual reduces its contribution if there is an unequal distribution of workload) [24]
– Social loafing (each individual reduces its contribution, because it is not possible to visualize the effort of each individual) [24]
– Free riding (each individual reduces its contribution, because they don't perceive their work as important for the team) [25]

These three harms can be less harmful, if the team members are not new to each other and if it is intended to work for a longer period together [17].

The mentioned effects can be found in agile and lean development environments:

The sucker effect should not be a major issue in a Scrum team abiding by the rules, i.e. that has a good Scrum Master: collaborative planning, daily stand-up meetings, as well as regular retrospectives on what went well in the team process and what did not, ensure that tasks are taken on by each team member and that recurring process issues, so-called "impediments", are tackled in the following sprint [42]. Some people might still take more time for certain tasks than others; however, the daily synchronization and retrospectives will make this transparent sooner or later.

Social loafing is a phenomenon often perceived when software companies change their development processes towards agile methods such as Scrum: Scrum focuses on team performance and measures the output per team per sprint, e.g. by means of story points [10]. Despite the fact that some teams track the progress of tasks assigned to individuals or teams of two, developers used to traditional methods often fear to "disappear" in the group.

Free riding can be seen if the team performs well on the group level with a more or less constant output of story points, while some individuals reduce their efforts, they might perceive their work as not so important for team. This perception might tempt them to reduce their individual effort and take the team credit for free. In a typical change project, this behavior usually follows resignation and adds to the social loafing effect described above [26].

Incentives to Work Together. Again, three different issues can be distinguished:

- The Koehler effect (less skilled group members can rise their performance to support the other team members) [20]
- Social competition (comparison with other group members can motivate to compete with each other and therefore raise performance) [48]
- Social compensation (better skilled group members work more to compensate for less skilled group members) [54]

Social Psychological Feedback Considerations for Individual and Team Information and Incentives. From a social psychological point of view, it is very important to differentiate between giving information about individual and team performance. One advantage of giving team feedback is that it can strengthen team building processes and emphasize collaboration [19]. At the other hand an informational feedback about the individual performance is important to avoid social loafing and free riding [24,25,34,51]. Finally incentives for individuals and teams from behavioral science are taken into account. Usually monetary and non-monetary incentives are applied to individuals, although it would be more important to distribute the incentives for achieving different team goals. Group incentives can support team building and the identification with the team, whereas individual incentives support competition [51,13]. Recent meta studies show, that group incentives are not always supportive for group performance, which can be partly attributed to social loafing [13,46]. Eventually a combination of team and individual incentives is the best solution to achieve top performance [46].

3   Methodology

Complementary to mere behavioral and design science studies [21], a simulation-based approach allows to analyze and better understand complex development scenarios with hundreds or even thousands of individuals and even more artifacts and process dependencies. Besides deduction and induction, experimenting with simulations is considered a "third way of doing science" [2]. To analyze and optimize complex development scenarios, different analytical and simulation-based approaches can be considered: discrete event simulations, agent-based simulation [6,7], system dynamics etc. Simulating software development processes to answer fundamental questions about agile and lean practices is, though still scarce, rising in number [9, see].

The complexity arising from individual actions and interactions that arise in the real world can be explicitly modeled in agent-based simulations in situations discrete-event simulations or system dynamics cannot [45]. Although being relatively new, agent-based simulations gain more and more momentum in various application areas where the behavior of single individual actors constitute the fundamental issues [33]. An agent-based system consists of autonomous agents following simple behavioral rules while being a direct abstraction from their real-world counterparts. Being autonomous and able to learn from their environment, they behave proactively following their own rule set [45]. Thus, the interaction among the agents directly impacts the system properties [8].

This section has shown that following an agent-based approach is an optimal choice to address the research questions. The following section will introduce, besides the system's structure and further artifacts, the actual model taken for implementation.

4   Assumptions and Model

This paper addresses large-scale business software development organizations with several hundred or thousands of developers. Moreover, we take a development process based on lean management and agile principles as a basis for our assumptions. In addition, this section describes the basic model of our agent-based simulation in a mathematical notation.

4.1   Work Items and Artifacts

Iteration Backlog. This backlogs contain all the user stories (backlog items) one team has committed to for one iteration, or sprint respectively, in Scrum. The backlog items are permanently kept uniquely prioritized by the team's product owner [44].

Iteration Backlog Item. User stories are containers for requirements and currently one of the most popular requirement modeling technique in agile methods. "User stories are the primary currency that carries the customer's requirements through the value stream into code and implementation." [32]. They briefly describe a feature from the perspective of a certain user role, letting the team freedom in implementational details.

Usable Software Increments each Iteration. At the end of each iteration the team produces a new software increment. This increment must be properly tested and fulfill other criteria in order to be accepted by the responsible person with regard to prior defined "done" (non-functional and/or meta-requirements) and functional "acceptance criteria". Agile methods aim at completing potentially shippable product increments, i.e. usable software in each iteration.

4.2 Team Process and Structure

Agile methods, such as Scrum, try to attain a trade-off between pragmatism and discipline, i.e. avoiding chaos on the hand and extensive bureaucracy on the other.

Team Size and Skills. The team must be "fully capable of defining, developing, testing, and delivering working and tested software into the system's baseline" [32]. Usually, such a "cross-functional" team consists of one product owner, a scrum master, 5-10 team members focussing on development, quality and testing as well as other functions and skills [29]. Teams are typically organized around particular software *components* (architectural view) or certain *features* (from a customer's perspective, see [28]). In general, features and components exhibit inherently dependent requirements, i.e. inter-team dependencies. In practice, companies have a mixture of both, feature and component teams, organized in a matrix (see e.g. [41]).

Inter-Team Collaboration in Large Development Organizations. In order to be able to release complex and comprehensive software products, development organizations of several hundred or even thousands of developers in cross-functional teams need to be coordinated, for which hierarchy levels need to be introduced. In our research, we follow the large-scale lean and agile model by Larman and Vodde [28,29]. This is also the basis for the implementation with which we calibrate our model [41]. The mix of feature and component teams (see above) is one of the reasons for occurring inter-team dependencies, which need to be resolved for product delivery. For instance, a certain set of master data requires multiple functional components of an enterprise resource planning application.

4.3 Model Parameters and Behavior

Agents & Teams. Let A^m represent the set of *agents* (e.g. developers and other cross-functional team members) in *team* m (m and n are arbitrary teams in the remainder of this article) with agents $a_1^m, \ldots, a_q^m$ such that $A^m = \{a_1^m, \ldots, a_q^m\}$. Let the agent a_1^m be a special agent ("team owner") representing the *Scrum Master* and the *Product Owner* of team m.[1] A team's *capacity* c^m is determined by the number of its agents n minus the team owner, i.e. a team $A^m = \{a_1^m, \ldots, a_q^m\}$ has the capacity of $c^m = q-1$.

---

[1] Our model is simplified based on the assumption that both, Scrum Master or Product Owner, can take over team tasks with approximately 50% of their capacity—therefore, one full-time equivalent is accounted per team. The team owner parameter is also applied for Area Product Owners and Chief Product Owner depending on the level of hierarchy and aggregation.

Team Backlog. Let furthermore B^m denote the *backlog* of team m with prioritized *backlog items* $b_1^m, \ldots, b_k^m$ such that $B^m = \{b_1^m, \ldots, b_l^m\}$ (b_x and b_y are arbitrary backlog items in the remainder of this article). The index l represents the priority or rank within the backlog – i.e. the backlog item b_{l-1}^m is the unambiguous antecessor of the backlog item b_l^m.

Backlog Processing. It is assumed that until all done criteria are satisfied, the *processing* of a backlog item consumes a well-defined[2] period of time t. The processing function $\lambda : B \rightarrow T$ maps backlog items to a processing time $t \in T$.

Backlog Dependencies. It is further assumed that *dependencies* between backlog items may exist such that the possibility to start processing a specific backlog item depends on the successful processing and finalizing of another item (all done criteria fulfilled). The dependency function $d : B \times B \rightarrow \{0, 1\}$ maps a pair of backlog items (b_x^m, b_y^m) to elements $0, 1$ with 0 representing independent backlog items and 1 denoting that backlog item b_x^m is dependent on item b_y^m.

$$d(b_x^m, b_y^n) = \begin{cases} 0 \text{ , if} & b_x^m \text{ is independent of } b_y^n \\ 1 \text{ , if} & b_x^m \text{ depends on } b_y^n \end{cases} \tag{1}$$

For the sake of simplicity, it is assumed that dependencies are *not directed*, i.e. if backlog items are dependent, neither of them can be processed as long as the dependency persists.

From a team's perspective, it follows that there are two designated types of dependencies, i.e. *(i) intra-team dependencies* with $d(b_x^m, b_y^n) = 1 \ \wedge \ m = n$, i.e. *within* the team's own backlog and *(ii) inter-team dependencies* with $d(b_x^m, b_y^n) = 1 \ \wedge \ m \neq n$, i.e. with other teams' backlog items.

Dependency Resolution. It is further assumed that dependencies between backlog items need to be resolved during the development process. Such a resolution is done by investing additional time and effort for analysis, communication, coordination, and in some cases, idle time. This means that the cost for dependency resolution depends on three factors: *(i)* The point in time during the development process the resolution is conducted and *(ii)* the complexity of the dependent backlog items which is implicitly represented by their processing time as well as *(iii)* the type of dependency (intra- or inter-team dependency). Practically this means: The earlier a dependency is detected and the lower the item complexity is, the less additional time is required to resolve it. The amount of effort, i.e. the additional time to be spent for resolving the dependency, also depends on the type of dependency (inter-team or intra-team). Thus, the resolution function $r : B \times B \times \Theta \rightarrow T$ (Equation 2) maps pairs of backlog items and the point of time within the development process to the period of time that is required for resolving their dependency (for a complete mapping, the resolution functions returns $t = 0$ in case backlog items are independent).

---

[2] As an extension of the model, the processing time might be represented by a probability distribution.

$$r(b_x^m, b_y^n, \theta) = \begin{cases} 0 & \text{, if} \quad d(b_x^m, b_y^n) = 0 \\ \bar{t}_{\text{intra}}\theta^2 \frac{p(b_x)+p(b_y)}{2} & \text{, if} \quad d(b_x^m, b_y^n) = 1 \ \wedge \ m = n \\ \bar{t}_{\text{inter}}\theta^2 \frac{p(b_x)+p(b_y)}{2} & \text{, if} \quad d(b_x^m, b_y^n) = 1 \ \wedge \ m \neq n \end{cases} \qquad (2)$$

The resolution time at least equals the average processing time of both items, i.e. their mean complexity and is mainly determined by the constant $\bar{t}$ representing the type of dependency (intra- or inter-dependency) and the point of time θ the resolution is conducted.

5   Simulation

Thus, the evaluation is conducted by means of an agent-based simulation based on a simple form of a Q-Learning model [52]. In contrast to more sophisticated variants of Q-learning models, the simulation model at hand considers multiple actions but only a single state. This reduction of parameter complexity is done without loss of validity and therefore simplifies the calibration of the simulation. Simplifying the simulation model reduces the number of assumptions, allowing for a better generalization of results.

5.1   Rounds

Reflecting the lean principles, simulation rounds Ω are mapped onto development "takts" (or "sprints" in Scrum [44]). Each round represents a development takt that is further discretized into a fixed number of takt units ω^3.

5.2   Actions

At the beginning of each TAKT, each agent chooses an action k out of the action space K as specified in Section 5.3. The following actions are available to each agent:

Preceding Intra-team Dependency Resolution. The agent focuses on resolving dependencies between backlog items within its team at the beginning of the development takt (preceding). If there is capacity left after this action, the agent continues with processing backlog items.

Preceding Inter-team Dependency Resolution. The agent targets the resolution of dependencies between backlog items that are planned in different teams at the beginning of the development takt (preceding). If there is capacity left after this action, the agent continues with processing backlog items.

Development without Early Dependency Resolution. No resolution of dependencies at the beginning of the development takt are addressed by the agent, i.e. the agent directly starts with backlog item processing. However, when processing a backlog item that is constrained by a dependency, the agent is forced to resolve this dependency at that point in time which might be time consuming due to the elapsed time (cp. Section 4.3).

---

[3] For the sake of simplicity, all time-related model values are discretized accordingly.

Having chosen, the agents execute the particular action which binds their capacity according to the defined time requirements. In case of dependency resolution actions ($k = 1, 2$) the capacity is bound exactly as long as the resolution function specifies the number of TAKT units required. If this period of time is less than the total units within a TAKT, the agent's capacity is free for development activities. In case of the development action ($k = 3$), the agent is processing backlog items during the whole TAKT.

5.3 Feedback and Learning Behavior

At the end of each TAKT Ω, each agent a receives a feedback $\pi_{a,k}^{\Omega}$ as a response to the action k chosen at the beginning of the TAKT.

To analyze the effect of different incentive schemes on the exchange of information within and between teams, we examine three feedback mechanisms:

Individual incentives that reward value creation of the individual developer, i.e. the number of successfully processed backlog items by a single developer.

Team incentives that reward each individual based on the value creation of the whole team the developer belongs to, i.e. the number of successfully processed backlog items accumulated on team-level.

At the end of each takt Ω, the feedback to a chosen action k of an agent a is incorporated in the agent's private fitness function $f_{a,k}^{\Omega}$. Balancing past and present experiences, the learning parameter $\beta \in [0, 1]$ determines to which degree past and present feedback is incorporated into the fitness update. Thus, the fitness update evolves as follows:

$$f_{a,k}^{\Omega} := \beta f_{a,k}^{\Omega-1} + (1 - \beta)\pi_{a,k}^{\Omega} \tag{3}$$

Thus, each agent maintains a fitness value for each possible action that represents the historical "success" of that particular action based on the cumulated feedback over time.

At the beginning of each TAKT Ω, each agent a chooses an action k out of the action space K (cp. Section 5.2) based on its particular probability $p_{a,k}^{\Omega}$ that is based on its fitness value and therefore on its historical "success":

$$p_{a,k}^{\Omega} := \frac{f_{a,k}^{\Omega}}{\sum_{k} f_{a,k}^{\Omega}} \tag{4}$$

5.4 Parametrization and Hypothesis

The simulation model as described previously is parameterized as follows: According to lean development best practices, the team size is set to 10 agents per team. A learning rate $\beta = 0.5$ yields an optimal trade-off between escaping local optima and achieving a quick convergence of strategies. The first 400 rounds of 500 rounds in total are the simulation's training phase in order to achieve a converged state and are therefore not considered for the statistical evaluation. As the number of variable parameters and their

interdependencies are high, heavy statistical noise is likely to be generated. To counteract the high volatility of the simulation model, a large number of 500 problem sets is evaluated and the mean results across all agents and teams are reported. The large size of analyzed problem sets for each observation assures robustness of the t-test to violations of the normality assumption [40].

By means of this agent-based simulation approach we intent to verify the hypotheses outlined in Table 1.

Table 1. Incentive schemes and corresponding hypotheses. NORES denotes the mean fitness value of action $k = 1$ across all agents and teams. INTRARES denotes the mean fitness value of action $k = 2$ across all agents and teams. INTERRES denotes the mean fitness value of action $k = 3$ across all agents and teams.

Incentive Scheme	Hypothesis
Individual Incentives	H1a: (NORES > INTRARES)
	H1b: (NORES > INTERRES)
	H1c: (INTERRES > INTRARES)
Team Incentives	H2a: (NORES < INTERRES)
	H2b: (INTERRES > INTRARES)

The set of hypotheses is derived from existing literature on the effect of incentives in lean development [37] and practical experiences from lean projects in SAP. In settings with individual incentives that reward agents solely on the number of backlog items that are successfully processed on their own, the agents are likely to follow an opportunistic strategy, i.e. they focus on processing backlog items instead of resolving dependencies (neither within their team nor between teams) as stated in hypotheses H1a and H1b. Resolving inter-team dependencies at a later point in time is more time consuming than intra-team dependencies which is likely to incentivize agents to prefer the INTERRES strategy over the INTRARES strategy at the beginning of each sprint. This argumentation holds for either incentive scheme (cp. hypotheses H1c and H2b).

On the other hand, team incentives that reward agents based on the total number of successfully processed backlog items of the whole team are likely to implement incentives for agents to follow actions which are beneficial for the team itself. As the effort to resolve backlog item dependencies at a later point in time is exponentially higher than at the beginning of the sprint, agents are likely to follow an early dependency resolution (cp. hypotheses H2a).

The statistical significance of the stated hypothesis is tested using a one-tailed matched-pairs t-test analyzing the alternative hypothesis, that is, the mean difference of the actions' fitness values is greater than zero. For the statistical analysis, the first 400 simulation rounds/sprints are skipped as they serve as the initial learning phase of the agents until we observe a convergence of strategies and achieve a stable state.

6 Evaluation Results and Implications

This sections describes the main findings of the agent-based simulation for the individual and team incentive schemes. Having been analyzed by means of a sensitivity
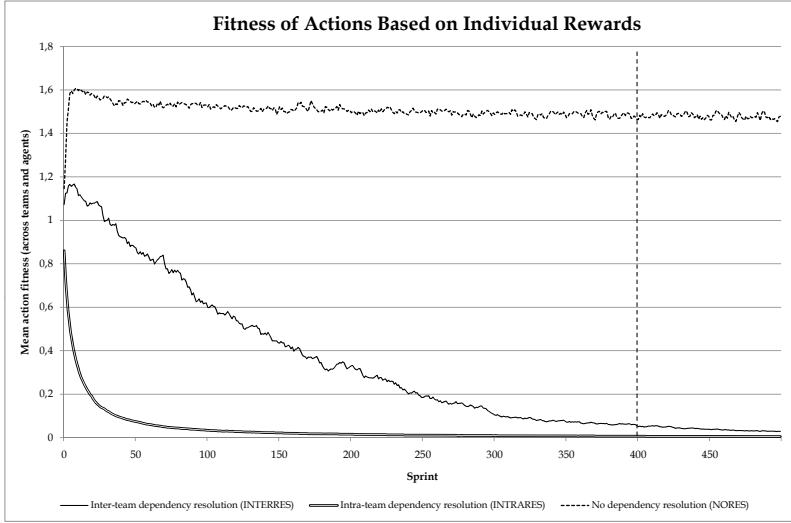
Fig. 1. Mean fitness across all agents and teams with individual rewards for actions inter-team dependency resolution (INTERRES)(mean=0.040, std=0.008), intra-team dependency resolution (INTRARES)(mean=0.006, std=0.0004), and no dependency resolution (NORES)(mean=1.479, std=0.012). The figure shows the training phase (rounds $<= 400$) as well as the convergence phase (rounds > 400). Given values for mean and std refer only to the convergence phase.

analysis, the observations are robust against the simulation parameters "number of agents per team", "number of teams", and "learning rate".

6.1 Individual Incentives

Simulation settings with the individual incentive scheme yield the following results (cf. Figure 1 in a setting with 5 teams consisting of 10 team members):

- The action no dependency resolution (NORES) significantly ($p << 0.01$) yields the highest overall mean fitness across all agents and teams which supports Hypothesis H1a and H1b
- The action inter-team dependency resolution (INTERRES) yields a significantly ($p << 0.01$) higher mean fitness across all agents and teams than the action intra-team dependency resolution (INTRARES) which supports Hypothesis H1c.

6.2 Team Incentives

In settings where agents are rewarded based on the total number of successfully processed backlog items of the whole team, the following results can be observed (cf. Figure 2 (training and convergence phase) and Figure 3 (convergence phase) in a setting with 5 teams consisting of 10 team members):

- The action inter-team dependency resolution (INTERRES) is strictly dominating the action no dependency resolution ($p << 0.01$) which supports hypothesis H2a.
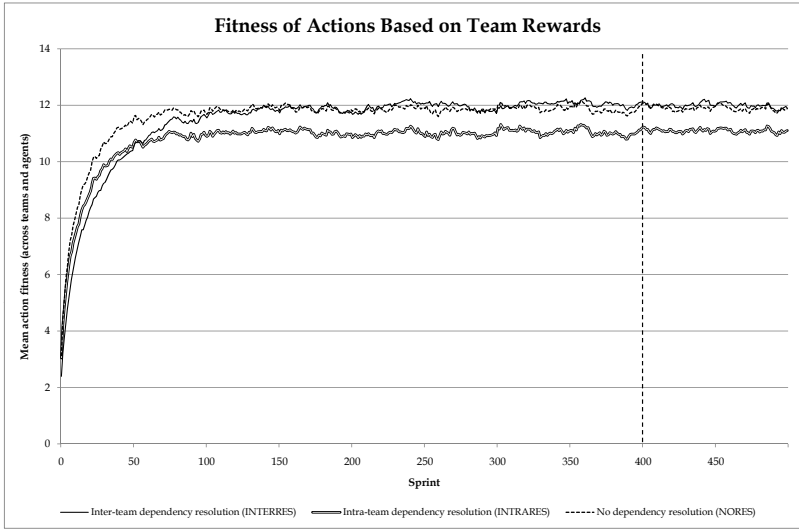
Fig. 2. Mean fitness across all agents and teams with team rewards for actions inter-team dependency resolution (INTERRES), intra-team dependency resolution (INTRARES), and no dependency resolution (NORES) (including training phase)
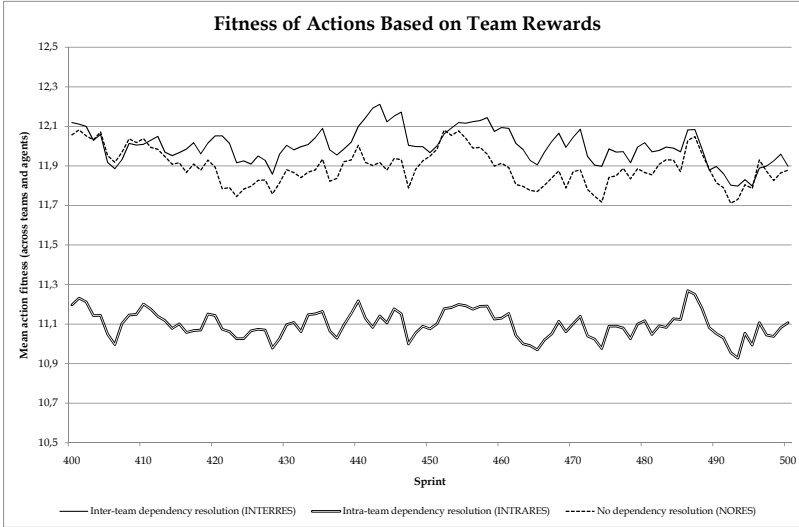
Fig. 3. Mean fitness across all agents and teams with team rewards for actions inter-team dependency resolution (INTERRES)(mean=12.0, std=0.087), intra-team dependency resolution (INTRARES)(mean=11.098, std=0.068), and no dependency resolution (NORES)(mean=11.894, std=0.090) (convergence phase)

– The action intra-team dependency resolution (INTRARES) is significantly ($p \ll$ 0.01) outperformed by the action inter-team dependency resolution (INTERRES) which supports hypothesis H2b.

6.3   Practical Implications

In our work, we analyzed the effect of organizational settings and incentive schemes in large-scale lean software development on the information flow within and between teams as well as performance aspects.

Our analysis has shown that individual rewards foster opportunistic behavior in teams, i.e. actions that serve the team by resolving backlog item dependencies and removing impediments are not conducted by the agents. On the other hand, a team incentive scheme that rewards agents based on the total number of successfully processed backlog items of the whole team promote behavior that is beneficial for the whole team. As the effort to resolve backlog item dependencies at a later point in time is exponentially higher than at the beginning of the sprint, agents follow an early dependency resolution. More precisely, resolving inter-team dependencies at a later point in time is more time consuming than intra-team dependencies which incentivizes agents to prefer a dependency resolution across team boundaries. In general, our results underline the importance of dependency resolution—and therefore, traceability and requirements management, in large software organizations [22].

One of the basic principles of the lean methodology states the empowerment of the teams instead of enforcing a strictly governed process corset [30]. As a trade-off, this implies that managerial monitoring and steering of the development process becomes cumbersome. Therefore, traditional methodologies and tools stemming from well-known project management techniques are partly not applicable in agile environments, which requires new approaches to manage a successful execution of lean projects.

Moreover, our work has shown that a sensible and efficient design of incentive schemes in large-scale lean software development is a promising tool to steer individual behavior, diminish opportunism and local optimization, foster efficient communication across team boundaries, and break silos that clash with the company's overall objectives. Hence, our results indicate that team-based rewarding can prevent opportunistic behavior and silo thinking which is in line with recent literature [37].

7   Summary of Findings and Conclusions

The contribution of our work comprehends the following findings:

– Incentive schemes play a central role for steering large-scale lean software development and to align individual and company objectives.
– In such complex environments, agent-based simulations are a promising method to evaluate different incentive designs and derive practical implications.
– Rewards based on individual performance advocate selfish behavior of team members, i.e. each individual focuses on silo work instead of removing impediments and sharing information within and between teams to resolve dependencies.

– Rewards directly tied to team-based value creation help to diminish opportunistic behavior and implement incentives to foster backlog item dependency resolution through intense communication across team boundaries.

The results of the simulation match very well the findings from social psychology. Support for H1c and H2b can be easily found because a differentiation between an in-group and an out-group (in this particular case between INTERRES and INTRARES) is always in favor of the in-group for an individual. This is due to the fact that being differentiated gives an individual a positive value, especially if someone identifies itself with its in-group [50]. Therefore a combination of team and individual incentives might help to achieve top performance in e.g. software development [46].

Outlook. As future work, we will validate our simulation results more systematically with real-world data from large-scale software enterprises implementing lean and agile practices. More specifically, we plan to analyze existing backlogs, log files, and other documentation of work practices as well as conduct qualitative interviews with a certain number of teams from different product areas. In doing, so we intend to (a) further elaborate the external validity of our simulation results and (b) gain more insights regarding the industrial context of our research questions.

Furthermore, we intend to investigate more sophisticated incentive schemes and their composition into hybrid patterns. We also plan to extend our model regarding hierarchical organizational settings and implications of distributed teams with communication barriers. Questions like how different incentive schemes can be grouped and assessed regarding their applicability and suitability in different organizational settings need to be further investigated. From an economic perspective, we plan to extend the underlying model to capture partly irrational behavior and to vary the feedback quality in terms of timeliness and signal noise.

References

1. Agile Alliance: Agile Manifesto (2001), `http://www.agilemanifesto.org` (April 14, 2004)
2. Axelrod, R.: Advancing the art of simulation in the social sciences. Complex 3, 16–22 (1997)
3. Beck, K.: Embracing change with extreme programming. IEEE Computer 32, 70–77 (1999)
4. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
5. Beck, K.: Test-Driven Development: By Example. Addison-Wesley (2003)
6. Blau, B., Conte, T., van Dinther, C., Weinhardt, C.: A Multidimensional Procurement Auction for Trading Composite Services. Electronic Commerce Research and Applications Elsevier Journal. Special Issue on Emerging Economic, Strategic and Technical Issues in Online Auctions and Electronic Market Mechanisms 9(5), 460–472 (2010); special Section on Strategy, Economics and Electronic Commerce
7. Blau, B., Conte, T., Weinhardt, C.: Incentives in Service Value Networks – On Truthfulness, Sustainability, and Interoperability (December 2010)
8. Bonabeau, E.: Agent-based modeling: Methods and techniques for simulating human systems. Proceedings of the National Academy of Science of the USA (2002)
9. Cao, L., Ramesh, B., Abdel-Hamid, T.: Modeling dynamics in agile software development. ACM Trans. Manage. Inf. Syst. 1, 5:1–5:26 (2010)

10. Cohn, M.: Agile estimating and planning. Prentice Hall (2006)
11. Cohn, M., Ford, D.: Introducing an agile process to an organization (software development). Computer 36(6), 74–78 (2003)
12. Conboy, K.: Agility from first principles: Reconstructing the concept of agility in information systems development. Information Systems Research 20(3), 329–354 (2009)
13. DeMatteo, J.S., Eby, L.T., Sundstrom, E.: Team-based rewards: Current empirical evidence and directions for further research. Research in Organizational Behavior 20, 141–183 (1998)
14. Dingsoyr, T., Dybå, T., Moe, B.: Agile Software Development: Current Research and Future Directions. Springer, Heidelberg (2010)
15. Dyba, T., Dingsoyr, T.: Empirical studies of agile software development: A systematic review. Information and Software Technology 50(9-10), 833–859 (2008)
16. Erdogmus, H., Williams, L.: The economics of software development by pair programmers. Engin. Econom. 48, 283–319 (2003)
17. Erez, M., Somech, A.: Is group productivity loss the rule or the exception? effects of culture and Group-Based motivation. The Academy of Management Journal 39(6), 1513–1537 (1996)
18. Fry, C., Greene, S.: Large scale agile transformation in an on-demand world. In: Proceedings of the AGILE Conference 2010, pp. 136–142 (2007)
19. Hertel, G., Konradt, U., Orlikowski, B.: Managing distance by interdependence: Goal setting, task interdependence, and team-based rewards in virtual teams. European Journal of Work and Organizational Psychology 13(1), 1–28 (2004)
20. Hertel, G., Kerr, N.L., Mess, L.A.: Motivation gains in performance groups: Paradigmatic and theoretical developments on the koehler effect. Journal of Personality and Social Psychology 79(4), 580–601 (2000)
21. Hevner, A., March, S., Park, J., Ram, S.: Design science information systems research. MIS Quarterly 28(1), 75–105 (2004)
22. Hildenbrand, T.: Improving Traceability in Distributed Collaborative Software Development—A Design-Science Approach. Phd thesis, University of Mannheim, Germany, Frankfurt (2008)
23. Katz, R., Allen, T.J.: Investigating the not invented here (NIH) syndrome: A look at the performance, tenure, and communication patterns of 50 r & d project groups. R&D Management 12(1), 7–20 (1982)
24. Kerr, N.L.: Motivation losses in small groups: A social dilemma analysis. Journal of Personality and Social Psychology 45(4), 819–828 (1983)
25. Kerr, N.L., Bruun, S.E.: Dispensability of member effort and group motivation losses: Free-rider effects. Journal of Personality and Social Psychology 44(1), 78–94 (1983)
26. Kotter, J.: Leading change. Harvard Business Press (1996)
27. Kozlowski, S., Bell, B.: Work groups and teams in organizations. In: Handbook of Psychology (2003)
28. Larman, C., Vodde, B.: Scaling Lean and Agile Development: Thinking and Organizational Tools for Large-Scale Scrum. Addison-Wesley Longman (2008)
29. Larman, C., Vodde, B.: Practices for Scaling Lean and Agile Development: Large, Multi-site, and Offshore Product Development with Large-Scale Scrum. Addison-Wesley Longman (2010)
30. Lee, G., Xia, W.: Toward agile: An integrated analysis of quantitative and qualitative field data. MIS Quarterly 34(1), 87–114 (2010)
31. Leffingwell, D.: Scaling software agility: best practices for large enterprises. Addison-Wesley (2007)
32. Leffingwell, D.: The big picture of enterprise agility by dean. Whitepaper, pp. 1–16 (2009)
33. Macal, C.M., North, M.J.: Agent-based modeling and simulation: desktop abms. In: Proceedings of the 39th Conference on Winter Simulation: 40 Years! The Best is Yet to Come, WSC 2007, pp. 95–106. IEEE Press, Piscataway (2007)

34. Matsui, T., Kakuyama, T., Onglatco, M.U.: Effects of goals and feedback on performance in groups. Journal of Applied Psychology 72(3), 407–415 (1987)
35. Petersen, K., Wohlin, C.: Measuring the flow in lean software development. Software - Practice and Experience (2010)
36. Poppendieck, M., Poppendieck, T.: Lean software development: an agile toolkit. Addison-Wesley Professional (2003)
37. Poppendieck, M.: Unjust deserts. Better Software, 33–47 (July/August 2004)
38. Poppendieck, M., Poppendieck, T.: Implementing Lean Software Development: From Concept to Cash. The Addison-Wesley Signature Series. Addison-Wesley Professional (2006)
39. Reinertsen, D.G.: The Principles of Product Development Flow: Second Generation Lean Product Development. Celeritas Publishing (2009) ISBN 978-1935401001
40. Sawilowsky, S., Blair, R.: A more realistic look at the robustness and type II error properties of the t test to departures from population normality. Psychological Bulletin 111(2), 352–360 (1992)
41. Schnitter, J., Mackert, O.: Introducing agile software development at sap ag - change procedures and observations in a global software company. In: Proceedings of the 5th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE (2010)
42. Schwaber, K.: Agile project management with Scrum, vol. 7. Microsoft Press, Redmond (2004)
43. Schwaber, K.: The Enterprise and Scrum. Microsoft Press (2007)
44. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice Hall (2001)
45. Siebers, P.O., Macal, C.M., Garnett, J., Buxton, D., Pidd, M.: Discrete-event simulation is dead, long live agent-based simulation! J. Simulation 4(3), 204–210 (2010)
46. Snell, S.A., Dean, J.W.: Strategic compensation for integrated manufacturing: The moderating effects of jobs and organizational inertia. The Academy of Management Journal 37(5), 1109–1140 (1994)
47. Sommerville, I.: Software Engineering, 9th edn. Addison-Wesley Longman (2010) ISBN-13: 978-0137053469
48. Stroebe, W., Diehl, M., Abakoumkin, G.: Social compensation and the koehler effect: Toward a theoretical explanation of motivation gains in group productivity. In: Witte, E., Davis, J. (eds.) Understanding Group Behavior: Consensual Action by Small Groups, vol. 2, pp. 37–65. Erlbaum, Mahwah (1996)
49. Tuckman, B.W.: Developmental sequence in small groups. Psychological Bulletin 63(6), 384–399 (1965)
50. Turner, J., Tajfel, H.: Social categorization and social discrimination in the minimal group paradigm: Studies in the social psychology of intergroup relations. In: Tajfel, H. (ed.) Differentiation between Social Groups, European Monographs in Social Psychology, vol. 14, pp. 101–140. Academic Press, London (1978)
51. Wageman, R.: Interdependence and group effectiveness. Administrative Science Quarterly 40(1), 145–180 (1995)
52. Watkins, C., Dayan, P.: Q-Learning. Machine Learning 8(3), 279–292 (1992)
53. West, D., Grant, T.: Agile development: Mainstream adoption has changed agility. Tech. rep., Forrester Research (January 2010)
54. Williams, K.D., Karau, S.J.: Social loafing and social compensation: The effects of expectations of co-worker performance. Journal of Personality and Social Psychology 61(4), 570–581 (1991)
55. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the case for pair programming. IEEE Softw. 17, 19–25 (2000)
56. Yin, R. K.: Case study research: Design and methods. Sage Publications (2007)

Comparing and Evaluating Existing Software Contract Tools

Janina Voigt, Warwick Irwin, and Neville Churcher

Department of Computer Science and Software Engineering, University of Canterbury
Christchurch, New Zealand
`jvo24@pg.canterbury.ac.nz`,
`{warwick.irwin,neville.churcher}@canterbury.ac.nz`

Abstract. The idea of using contracts to specify interfaces and interactions between software components was proposed several decades ago. Since then, a number of tools providing support for software contracts have been developed. In this paper, we explore eleven such technologies to investigate their approach to various aspects of software contracts. We present the similarities as well as the areas of significant disagreement and highlight the shortcomings of existing technologies. We briefly introduce PACT, a software contract tool under development, explaining its approach to various aspects of software contracts.

Keywords: Software contracts, Design by contract, Formal software specification.

1 Introduction

When writing software, we aim to create programs which not only work correctly, but are also reliable, easy to use, understand and maintain. These and other factors combine to determine the level of quality in software.

Developing high quality software is a difficult, complex and time-consuming task. The sheer size and complexity of software contribute to these difficulties; it is not unusual for a single program to contain millions of lines of code, far too much for one person to understand. To manage this size and complexity, we break large systems into smaller components which can be developed independently. A developer working on one component does not need to know the internal details of other components of the system; he or she only needs to understand the other components' interfaces in order to use their services.

Software contracts (a subfield of formal specifications) are used to explicitly define the interfaces of software components, specifying the responsibilities of both the client using a service and the supplier of the service. This formalises the interactions between components of the software and ensures that two components interact correctly [27].

When software contracts are not used, clients of a service usually have access to information about the service's interface, including method signatures, as well as, optionally, documentation about how to use the service. Software contracts elaborate on this by formally specifying protocols of interaction which otherwise may have remained

implicit. Consequently, we regard contracts as a natural extension of explicit type systems; they specify interfaces fully rather than just specifying signatures.

We believe that software contracts can mitigate some of the problems surrounding large scale software development. They not only improve the correctness of software by explicitly specifying interaction protocols, but also serve as documentation and clarify correct use of inheritance [27].

Further, formal specifications such as software contracts "represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can easily be manipulated" [31, page 119]. In particular, software contracts describe valid inputs and outputs to methods; this information can be used by automatic testing tools to find valid test inputs and decide if particular test outputs are correct.

Despite the fact that the main ideas of software contracts were proposed several decades ago, they are still not commonly used in mainstream software development. Meyer remarks that

> In relations between people and companies, a contract is a written document that serves to clarify the terms of a relationship. It is really surprising that in software, where precision is so important and ambiguity so risky, this idea has taken so long to impose itself. [27, page 342]

However, more recently several different technologies supporting software contracts have been developed, including tools for mainstream programming platforms such as Java and .NET. Along with these technologies, a number of supporting tools are emerging. Testing tools such as AutoTest for Eiffel [28] and Pex for .NET [3,32] automatically extract unit tests from contracts without the need for input from developers. Static analysers such as Boogie for the .NET contract language Spec# [1] and ESC/Java for the Java contract language JML [13] attempt to prove the correctness of software at compile-time.

As more technologies supporting software contracts emerge and their usage becomes more common, it is important for us to take stock of current developments and uncover any issues and areas of disagreement which need to be addressed in the future. This is what we attempt to do in this paper, as part of a wider project in which we seek both to strengthen the theoretical underpinnings of contracts and to develop tools to support the adoption of contracts in modern software engineering environments.

We are currently developing our own software contract tool, PACT, applying the lessons learned from evaluating and comparing existing contract tools [34]. Although we do not aim to describe PACT in detail here, we highlight how it approaches some aspects of software contracts as we discuss them.

The rest of this paper is structured as follows: Section 2 explains the background of software contracts. Section 3 presents a comparison of several contract technologies, highlighting the similarities and differences. A discussion of the issues and criticisms of existing approaches follows in Sect. 4 before we present our conclusions in Sect. 5.

2   Background

The roots of software contracts run very deep in the field of computer science; although it has been little recognised in the literature, the origins of the idea can be traced as far back as Turing, who first presented the idea of assertions to check program correctness in 1949 [33].

In 1969, Hoare introduced *Hoare triples*. He used the notation $P\{Q\}R$ to mean that "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion" [14, p. 577]; P is commonly called the *precondition*, while R is the *postcondition*. Three years later, Hoare also presented the concept of the *class invariant*, a logical predicate I where "each operation (except initialisation) may assume I is true when it is first entered; and each operation must in return ensure that it is true on completion" [15, p. 275].

In the late 1980s, Meyer applied Hoare's work in his development of *Design by Contract*^{TM} and the programming language EIFFEL which included the concepts of preconditions, postconditions and class invariants [25]. Preconditions specify what the client must ensure before calling the service provider; this could for example include ensuring that the parameters are not null. Postconditions define what the service provider promises in return, given that the client has fulfilled the preconditions.

As an example, we define the contract for a simple `Stack` class using pseudo code with the standard methods `push(Object obj)` and `pop()`:

```
class Stack {
  private Object[] stack;
  private static final int MAX_SIZE = 100;
  private int size;

  Invariant: size >= 0 && size <= MAX_SIZE;

  Precondition: !isFull()
  Postcondition: peek() == obj
      && size == old size + 1
  public void push(Object obj){
    stack[size++] = obj;
  }

  Precondition: !isEmpty()
  Postcondition: size == old size - 1
  public Object pop() {
    return stack[--size];
  }
}
```

We have defined preconditions and postconditions for `push` and `pop`. The precondition for `pop` ensures that the methods are not called when the `Stack` is empty; the precondition for `push` makes sure the method is not called if the `Stack` is already full. The postconditions of the two methods check that the size of the `Stack` has changed in the correct way by comparing it to the `old` size of the `Stack`; that is, the size before the method's execution.

The invariant of the `Stack` ensures that its size never drops below zero or exceeds the `Stack`'s capacity. This invariant must be satisfied in all observable states of every instance of a class [25], specifically after the constructor has finished constructing a class instance and before and after each call to an exported method of the class; that is, a method accessible from outside the class. This implies that while methods of the class are executing, they may violate the class invariant, as long as it is again satisfied when the method returns [27].

Software contracts also apply in the presence of inheritance. When using inheritance, a subtype becomes substitutable for its supertype. This means, for example, that a method expecting an object of type A may be given an object of type B as long a B inherits from A. Whenever a client makes use of a supplier, it does not need to know whether the supplier is an immediate instance of the specified type or an instance of some subtype. Therefore, for contracting to continue to work, the subclass must adhere to the contract specified by the superclass [25]. This means that

- Preconditions must be the same or weaker than in the superclass. The subclass cannot expect more of the client, although it may expect less;
- Postconditions must be the same or stronger than in the superclass. The client expects certain results which must be delivered by the subclass. In addition, the subclass may choose to deliver more than promised by the superclass; and
- Class invariants are inherited from the superclass. The subclass may introduce additional class invariants [27].

In the next section, we present different contract technologies and contrast their approaches to the implementation and interpretation of software contracts.

3   Contract Technologies

We investigated a number of technologies and programming languages which allow the addition of software contracts to programs, with a particular focus on the following eleven:

- Java contract tools, including
  - JAVA MODELING LANGUAGE (JML) [19,21,20];
  - ICONTRACT [18];
  - CONTRACT JAVA [12];
  - HANDSHAKE [10];
  - JASS [6];
  - JCONTRACTOR [16,17]; and
  - JMSASSERT [23].
- .NET contract languages, including
  - SPEC# [4,22]; and
  - CODE CONTRACTS [11,29].
- EIFFEL [25,26,27]; and
- OBJECT CONSTRAINT LANGUAGE (OCL) [30,35]

The large number of existing software contract tools made it impractical to consider all of them and we therefore focused our investigation on the main technologies which add contract support to the popular programming platforms Java and .NET. In addition, we looked at Eiffel, the original software contract language. OCL was included because of its close links to Java technologies.

All the tools we investigated aim to support software contracts, most of them at the implementation level. OCL is the only technology to work exclusively at the software design level; it allows contracts including preconditions and postconditions to be added to UML diagrams, while all other tools we looked at allow developers to augment source code using contracts.

We have identified significant differences and shortcomings in what they deliver. Table 1 gives an overview of the similarities and differences of the tools. In the following section, we describe the main characteristics of the technologies, and in the subsequent section we summarise the most important themes and highlight areas of inconsistency.

3.1   Core Contract Support

All of the technologies we looked at provide core contract support, allowing the specification of preconditions, postconditions and class invariants, with the exception of CONTRACT JAVA which omits class invariants.

In addition to the basic contract specifications, some technologies offer additional constructs. SPEC#, JML and JASS allow the specification of *frame conditions*. Frame conditions specify which parts of the memory a method is allowed to modify. This ensures that a method does not unexpectedly change the value of variables it should not be allowed to modify [4,22]. A variable is deemed to have been modified if it is accessible at the start and the end of a method and its value has been changed. This means that newly created objects and local variables are not included in the restrictions of frame conditions [19].

SPEC#, CODE CONTRACTS and JML further allow the definition of *exceptional postconditions*, which specify conditions that need to be satisfied if the method terminates with an exception.

Of the technologies we considered, JML provided the most extensive contract support. Among other constructs, it also supports history constraints which describe how the value of a field is allowed to change between two publicly visible states. This can for example be used to express that the value of a field may only increase [19]. JML further introduces the concept of *model fields* which can be used when the inner data representation of a class needs to be changed but the developer does not want to update all of the contracts to the new data format. The model field of the old data format can be used from within the contracts and a correspondence is defined between the new data format and the model field [20].

3.2   Special Operators and Quantifiers

The different technologies also offer varying amounts of special operators and quantifiers for use in contracts. All allow postconditions to refer to the return value of the method; this functionality is usually provided by the `result` or `return` operator. In

Table 1. Overview of Contract Tools

		JML	iContract	Contract Java	Handshake	Jass	jContractor	JMSAssert	Spec #	Code Contracts	Eiffel	OCL
Contract Support	Pre / Postconditions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Class Invariant	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
	Frame Conditions	✓					✓		✓			
	Exceptional Postconditions	✓							✓	✓		
Operators	Result	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Old	✓	✓			✓	✓	✓	✓	✓	✓	✓
Contract Language	Original Language						✓		✓	✓	✓	✓
	Modified Language	✓	✓	✓	✓	✓						
	Scripting Language							✓				
Contract Placement	Comment	✓					✓	✓				
	Annotation		✓									
	With Program	✓	✓				✓	✓	✓	✓	✓	
	Separately			✓	✓		✓					✓
Method Purity	Enforced	✓							✓			✓
Precondition	Visible Members Only	✓				✓			✓			
Invariant Check	After Method		✓			✓				✓		
	Before and After	✓		N/A	✓		✓	✓			✓	N/A
	Expose Block								✓			
Invariant Check	All Methods					✓						
	Non-private Methods		✓	N/A	✓			✓		N/A	✓	N/A
	Public Methods Only	✓					✓			✓		
Contract Inheritance	Enforced	✓	✓	✓	✓		✓	✓	✓	✓	✓	N/A
	Precondition Weakening	✓	✓		✓	✓	✓	✓	✓		✓	N/A
Multiple Inheritance	Fully Supported										✓	✓
	For Interfaces Only	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Contract Compilation	Preprocessor		✓			✓		✓				
	Custom Compiler	✓		✓								N/A
	Standard Compiler								✓	✓	✓	
	Runtime Linking				✓		✓					

addition, all except CONTRACT JAVA and HANDSHAKE also allow postconditions to refer to the value of a variable before method execution, often through the `old` operator. This is important to check that the value of a field is changed correctly by a method, as we did in our `Stack` example above.

Most technologies also offer some quantifiers such as *for all* and *exists*; no such quantifiers are available in EIFFEL, but Meyer argues that they can be easily emulated using conventional programming language constructs [25]. Several tools, including JML, SPEC#, JCONTRACTOR and OCL, have a sophisticated range of additional operators including quantifiers, counting functions and predicate logic operators.

3.3   The Contract Language

Contract specifications for Java and .NET represent additions to an existing programming language. Some tools, including CODE CONTRACTS and JCONTRACTOR, specify contracts in the existing language. EIFFEL and SPEC# are both languages which natively support contracts and thus the language used to specify contracts is part of the wider programming language. The advantage of this approach is that there is no need for a separate compiler and contracts can be processed by standard tools along with the remainder of the program. In CODE CONTRACTS, contracts are specified by calling the static methods of the `Contract` class; for example, preconditions are defined by calling the `Requires` method of the `Contract` class. In JCONTRACTOR, method contracts are specified in *contract methods*, using standard Java. Postcondition methods take the additional parameter `RESULT`, which can be used to refer to the return value of the method. Postconditions also have access to a special object called `OLD`, which contains the state of the object as it was before the method executed [17].

The remaining tools we considered take a slightly different approach: they take the original programming language as a basis but augment it using additional keywords and operators. This approach is taken by ICONTRACT, JML and others; it requires special tools to translate the contracts into the original programming language.

JMSASSERT takes this approach a step further by using a full scripting language, JMScript, for contract specification. While JMScript is similar to Java, the underlying programming language, it differs sufficiently that developers need to learn the scripting language before being able to write contracts, significantly steepening the learning curve.

3.4   Integration of Contracts into Source Code

There are several ways in which contracts can be incorporated into source code. Some contract technologies, including JML, JASS and JMSASSERT, require contracts to be added in the form of comments, while in ICONTRACT they are defined as annotations. The advantage of these two approaches is that they work when the contract language is not the same as the standard programming language; the contracts are simply ignored by the standard compiler, meaning that no special compiler is needed when working with contracts. Instead, the contracts are inserted into the source code by a preprocessor and the program is then compiled using the standard compiler.

In EIFFEL, SPEC#, CODE CONTRACTS and JCONTRACTOR, contracts are defined as an integral part of the program and are compiled and checked by the standard compiler. This approach works for these technologies because the contracts are expressed in the same language as the rest of the program.

The placement of contracts in the programs also varies between different technologies. In most cases, for example in JML, ICONTRACT and SPEC#, method contracts including preconditions and postconditions are specified as part of the method header. In CODE CONTRACTS, preconditions and postconditions are placed inside the method body along with the method implementation. These two approaches have the advantage of clearly showing which contracts apply to which methods.

Other technologies enforce a separation between contracts and the code to which they apply. In HANDSHAKE, specifications are placed in separate contract files [17];

in CONTRACT JAVA they are placed in separate interfaces [12]. This approach has the advantage of clearly separating contracts from standard code, allowing them to be considered independently of implementation. It further allows the addition of contracts even when source code is not available, for example when working with third party software.

JCONTRACTOR allows both of these approaches: contract methods to define preconditions and postconditions may be placed in the same class as the methods to which they apply; alternatively, they can be defined in a separate contract class named `ClassName_CONTRACT`, which must extend the class to which it is adding contracts in order to inherit relevant behaviour and to make the objects with contracts substitutable for objects without contracts [17].

3.5   Side Effects in Contracts

Preconditions, postconditions and invariants should not call methods which cause side effects since this can create bugs which are difficult to trace. Some technologies, including SPEC# and JML, enforce this and allow only methods which have been declared free of side effects (pure methods) to be called from within contracts. Pure methods may only call other pure methods and may not modify any part of the memory. For example, the two query methods we used to define our `Stack` contract, `isEmpty` and `isFull`, have no side effects and can therefore safely be called from within a contract.

Most of the technologies do not explicitly enforce method purity; they only recommended that no methods with side effects are called from within contracts. CODE CONTRACTS is expected to enforce purity in the future [29]. OCL is a modeling language and all its code is implicitly free of side effects and thus any methods called from the contract are guaranteed to have no side effects.

3.6   Precondition Visibility

Contract theory requires clients to ensure that preconditions hold; in order to ensure that clients can perform such checks, it is important to ensure that preconditions do not refer to any data or methods which are not visible to clients [25,11]. Some contract technologies enforce this restriction, while others do not.

CODE CONTRACTS ensures that anything used to define the precondition is visible to clients. JASS and JML require anything referred to by the precondition to be at least as visible as the method itself. Thus, the preconditions of `public` methods must be defined using only `public` members; preconditions for `protected` methods may refer to both `public` and `protected` ones.

3.7   Checking of Class Invariants

Class invariants are constraints that need to be maintained in all visible states of the objects of a class; that is they must be true at the start and the end of each method that can be called by a client. For this reason, Meyer asserts that each invariant essentially represents an additional precondition and postcondition for each exported method in a

class [25]. EIFFEL, JML and JMSASSERT therefore check class invariants at the start and end of each method execution.

However, seeing the invariant as an addition to each method's precondition raises a new problem:

> The object invariant of class T is a condition on the internal representation of T objects, the details of which should be of no concern to a client of T, the party responsible for establishing the precondition. Making clients responsible for establishing the consistency of the internal representation is a breach of good information hiding practices. [2, page 30]

For this reason, other technologies, including CODE CONTRACTS, ICONTRACT and JASS, check the class invariant only at the end of method executions; that is, only in the postconditions, not the preconditions.

SPEC# takes a more complex approach to invariant checking. It allows changes to memory only inside special `expose` blocks because such changes could invalidate class invariants. At the start of each `expose` block, the object's invariant is set to `false`. Changes to data are now allowed and at the end of the `expose` block the invariant is re-checked. This protects invariants even in the presence of concurrency and reentrancy: an `expose` block can only be entered when the object's invariant is `true`; that is, it can only be entered by one thread of execution at a time [4]. While this approach has the advantage of working in the presence of concurrency, it greatly increases the complexity of writing programs with contracts.

Apart from the disagreement over when the invariant needs to be checked, there is also some debate about which methods this check applies to. Strictly speaking, the class invariant must be maintained in all externally visible states but may be broken while internal methods are executed. For example, a recursive method needs to maintain the invariant only for its outermost invocation. `Private` methods should be allowed to break the invariant; only methods called by the client should need to maintain it.

Of the technologies we considered, only JASS checks the invariant after each method execution, effectively forcing all methods, including `private` methods, to maintain the invariant. EIFFEL, ICONTRACT, HANDSHAKE and JMSASSERT require all non-private methods to maintain the invariant, while CODE CONTRACTS, JML and JCONTRACTOR only require `public` methods to do so.

Some of the Java technologies allow only `private` methods to break the class invariant, while others allow `private`, `package` and `protected` methods to do so. The latter approach is problematic, since calls to `package` and `protected` methods may come from a different class, and therefore should be forced to maintain the invariant. On the other hand, this allows methods from the subclass to call methods in the superclass while the invariant is broken, which may provide valuable flexibility.

3.8   Inheritance of Contracts

Inheritance is an important mechanism in object oriented programming and consequently contract tools need to support it. In many technologies, including EIFFEL, ICONTRACT, JML and JCONTRACTOR, correct contract inheritance is enforced by

disjuncting inherited preconditions and conjuncting postconditions and invariants; this leads to a weakening of preconditions and a strengthening of postconditions and invariants. CODE CONTRACTS and CONTRACT JAVA take a more restrictive approach: while postconditions and invariants may be added by subclasses, no new preconditions may be defined. This ensures that preconditions are not strengthened, but also makes developers unable to weaken them.

While almost all technologies we investigated always enforce correct use of contract inheritance, JASS takes a more flexible approach. It can check for correct inheritance using refinement checks, but this is optional and can be turned off by the developer. In OCL, the semantics of contract inheritance are not fully specified because it is a general purpose modelling language rather than a concrete implementation.

Multiple inheritance is often seen as more flexible and elegant than single inheritance [9,27]. Both .NET and Java support only single inheritance of classes and consequently none of the contract tools based on .NET and Java support multiple inheritance for classes; however, multiple inheritance is allowed between interfaces. EIFFEL, on the other hand, fully supports multiple inheritance, making it more flexible and expressive. Multiple inheritance is also allowed in UML diagrams and therefore handled by OCL.

3.9   Conversion of Contracts into Runtime Checks

Once contracts have been written, they are usually turned into runtime checks that report whenever a contract is violated. This conversion may be done in several ways.

Programs written in EIFFEL, CODE CONTRACTS and SPEC# can simply be compiled using a standard language compiler, since contracts are expressed in the same language as the rest of the code. The EIFFEL and SPEC# compilers insert runtime checks for contracts during compilation; CODE CONTRACTS uses library classes to implement contract checking. JML and CONTRACT JAVA provide a customised Java compiler which not only compiles the program but also generates the runtime checks. ICONTRACT, JASS and JMSASSERT all use a preprocessor which inserts Java statements into the code before it is compiled by the standard Java compiler. This has the advantage that the standard Java compiler can be used after preprocessing is completed. HANDSHAKE and JCONTRACTOR use a dynamic library and class loader to inject runtime checks when the program is executed, rather than at compile time.

4   Discussion

In our investigation of existing software contract technologies we have found some areas of significant disagreement. The approaches of the technologies vary widely and from Table 1 it becomes clear that no two tools take exactly the same approach.

Interestingly, we have uncovered some relatively basic issues which are handled inconsistently, for example concerning the checking of class invariants. We believe that it is important that the inconsistencies are resolved — or at least justified — in order to increase developers' confidence in contract tools and the practice of using software contracts in general.

We found good support for core contract concepts, including preconditions, postconditions and invariants, in nearly all tools. We believe that any contract tool which does

not support these basic constructs is inadequate for practical use. CONTRACT JAVA, for example, does not support the specification of class invariants, representing a serious gap in this tool.

In addition to preconditions, postconditions and class invariants, we find the concept behind frame conditions useful. It is often difficult to know what data is changed when calling a method, particularly if this method calls other methods. In some cases, unexpected data changes can be difficult to trace to their origins. Defining frame conditions forces developers to think carefully about which parts of the memory a method should be able to access and modify. They inform the programmer of inappropriate memory modifications, reducing the incidence of unexpected data changes.

Some contract technologies provide a wide range of special operators and quantifiers; most tools provide at least two: the `result` or `return` operator to access the return value of a method and the `old` operator to refer to the value of variables before the method execution. However, two tools, CONTRACT JAVA and HANDSHAKE, do not provide an `old` operator. This is a serious omission and severely restricts what contracts can be expressed, such as the size checks in our Stack example.

Our own contract tool, PACT, takes a different approach from the contract tools we have considered here; it does not provide an explicit `old` operator but rather allow developers to define variables in the precondition which can later be accessed in the postcondition. In this way, a value can be stored when the precondition is executed and evaluated in the postcondition.

Most tools we considered here declared contracts using the same programming language as for the rest of the program, although many introduced additional operators and quantifiers. Only one tool, JMSASSERT, used a significantly different language to define contracts. We suggest that this is an unnecessary burden on developers and is likely to inhibit uptake of the technology.

With the exception of CODE CONTRACTS, all of the technologies we investigated use contract definition syntax that groups contract information with method declaration information. CODE CONTRACTS places contracts inside the actual methods. We feel that this approach is not ideal, since it mixes contracts with implementation code and makes it difficult to distinguish between them. We suggest that contracts should ideally be declared separately from the implementation as part of a type definition. This is consistent with existing literature, which suggests that public interfaces, or types, should be separated [7,8]; that is, the type definition should contain signatures of visible methods, but no internal details. By extension, such a type definition should include contracts for publicly visible methods since, similarly to method signatures, contracts provide vital information to clients wanting to use a service. For these reasons, PACT explicitly distinguishes between types and implementations and includes software contracts in type definitions.

Some tools do not allow contracts to call methods with side effects since this can create bugs which are difficult to trace; other technologies do not impose this restriction. We agree with Barnett et al., who claim that the latter approach gives developers too much freedom and is unsound. [5] As we argued above, it can be difficult to see which parts of the memory a method modifies; similarly, it can be difficult to determine whether or not a method is pure, particularly when this method calls other methods,

which in turn may not be pure. This makes both frame conditions and explicit declarations of pure methods useful.

Clients are responsible for ensuring that preconditions are met before calling a method. We are therefore surprised that not more tools ensure that methods and data referred to in preconditions are visible to clients. If this is not the case, clients may not be able to check preconditions and may therefore fail to fulfill their responsibilities under the contract. Contracts are based on the idea of shared responsibility between clients and service providers and having potentially invisible preconditions violates the foundation of software contracts. Our tool PACT forces preconditions to be fully accessible to clients, ensuring that they can be checked correctly by clients.

In the tools we studied, we found a particularly variable approach to invariant checking. Some tools check invariants after each method, others before and after; some tools require the invariant to hold at the start and end of all methods while others only apply this restriction to `public` methods. In our view, the wide range of approaches stems from the incomplete body of theory about this aspect of contracts. We have found no research that explains when invariants should be checked and what implications the different approaches have. Given the wide range of different approaches, we feel that this is an area where further investigation is warranted.

Most of the technologies allow private methods to break the invariant temporarily. This makes sense because the internal operations of an object may not always maintain the invariant at all times; however, it needs to be restored before returning control to the client to ensure that the object is left in a consistent state. We therefore argue that ideally the invariant should be checked before and after every method call originating from outside the object. This would allow the object to break its own invariant temporarily (possibly while calling code in the superclass) but would also ensure that the object remains in a consistent state when it returns control. We aim to implement such invariant checking in PACT in the future.

In the context of invariant checking, SPEC#'s approach is far more complex than that of any other technology we investigated. It requires the object to be explicitly exposed whenever its state is modified to ensure that its invariant cannot be violated by operations from the outside or through the presence of reentrancy and concurrency. Although this approach is sound, we argue that it is too complex; it requires the use of complicated constructs even when writing simple programs. We believe that this complexity is likely to alienate new users and slow the uptake of SPEC# and software contracts in general.

Support for inheritance of contracts is essential for their use in OO programming. We found that all the tools with the exception of JASS ensure that contracts are inherited correctly. JASS also allows correct contract inheritance to be enforced but makes this optional. Most tools allow only single inheritance between classes, making them less flexible. Like OCL and EIFFEL, PACT supports more expressive multiple inheritance.

Overall, we are encouraged by the high level of support we found for correct inheritance. Using inheritance correctly is notoriously difficult and our intuition sometimes leads us to use it incorrectly. This is, for example, evident in the well-known *square-rectangle problem* [24]. Our own experience shows that contracts are very valuable when creating inheritance hierarchies because they force us to ensure that an instance

of the subclass is substitutable for an instance of the superclass; problems with contract inheritance usually signal incorrect use of inheritance.

Most of the tools we looked at enforce the correct use of contracts by allowing weaker preconditions through disjuncting inherited preconditions and allowing stronger postconditions through conjuncting inherited postconditions. CODE CONTRACTS uses this approach to ensure postconditions are strengthened; however, the tool does not allow the weakening of preconditions because "We just haven't seen any compelling examples where weakening the precondition is useful" [29, page 15]. CODE CONTRACTS forces developers to declare all preconditions on the root method of an inheritance chain. In our work with CODE CONTRACTS, we have found this approach very frustrating because it does not allow for flexible precondition definition. In particular, problems arise when a class inherits the same method from multiple interfaces. In this situation, the preconditions of this method in all ancestors must be compatible; this is an example where we feel that allowing precondition weakening is essential.

5   Conclusions

In our investigation of existing software contract tools we have uncovered a range of differences, clearly demonstrating a level of confusion and conflict surrounding even some basic concepts of software contracts. This indicates to us that more work is needed in this area to resolve these issues and create a consensus or at least a clear taxonomy of the different semantics of software contracts. We have identified a number of shortcomings of existing tools and areas that require more research, including:

- The checking of class invariants;
- The separation of contracts and implementation; and
- The inheritance of contracts, particularly the weakening of preconditions.

We believe that using software contracts has the potential to greatly increase the quality of software and speed up software development. Not only do they ensure that different components of a system know how to interact with each other correctly, but they also serve as documentation of developers' intentions and can be used as a basis of automated testing tools. Furthermore, we believe that they are a highly valuable tool for creating correct inheritance hierarchies.

The results presented in this paper have motivated the design and implementation of our own contract tool, PACT, described in detail in [34] and separately in a forthcoming publication. No single tool has all the functionality we consider essential, and we have identified a number of additional features which will make contracts more attractive and practicable for mainstream software engineering. In particular, PACT aims to:-

- Fully separate types from their implementations;
- Enhance the specification of types with contracts, including preconditions, postconditions and class invariants;
- Enforce correct inheritance of contracts;
- Support multiple inheritance;

- Support more expressive specification of contracts by allowing variables and other control statements to be used within contracts;
- Ensure that preconditions only make use of members which are accessible to clients; and
- Support checking of the class invariant at the end of each method call originating from outside the object.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., Deline, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3, 27–56 (2004)
3. Barnett, M., Fähndrich, M., Halleux, P.D., Logozzo, F., Tillmann, N.: Exploiting the synergy between automated-test-generation and programming-by-contract. In: Proceedings of ICSE 2009, 31th International Conference on Software Engineering, Companion, pp. 401–402 (2009)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
5. Barnett, M., Naumann, D., Schulte, W., Sun, Q.: 99.44% pure: useful abstractions in specifications. In: ECOOP Workshop on Formal Techniques for Java-Like Programs, FTfJP (2004)
6. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass - Java with assertions. Electronic Notes in Theoretical Computer Science 55 (2001)
7. Bruce, K.B.: Foundations of object-oriented languages: types and semantics. MIT Press, Cambridge (2002)
8. Canning, P.S., Cook, W.R., Hill, W.L., Olthoff, W.G.: Interfaces for strongly-typed object-oriented programming. In: OOPSLA 1989: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, pp. 457–467. ACM, New York (1989)
9. Cardelli, L.: A semantics of multiple inheritance. Information and Computation 76(2-3), 138–164 (1988)
10. Duncan, A., Hoelzle, U.: Adding contracts to Java with Handshake. Technical Report TRCS98-32, University of California at Santa Barbara, Santa Barbara, CA, USA (1998)
11. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: SAC 2010: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 2103–2110. ACM, New York (2010)
12. Findler, R., Felleisen, M.: Behavioral interface contracts for Java. Technical Report TR00-366, Rice University (2000)
13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 234–245. ACM, New York (2002)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12, 576–580 (1969)
15. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica 1, 271–281 (1972)

16. Karaorman, M., Abercrombie, P.: jContractor: Introducing design-by-contract to Java using reflective bytecode instrumentation. Formal Methods in System Design 27, 275–312 (2005)
17. Karaorman, M., Hölzle, U., Bruno, J.: *jContractor*: A Reflective Java Library to Support Design by Contract. In: Cointe, P. (ed.) Reflection 1999. LNCS, vol. 1616, pp. 175–196. Springer, Heidelberg (1999)
18. Kramer, R.: iContract - the Java(tm) design by contract(tm) tool. In: TOOLS 1998, p. 295. IEEE Computer Society, Washington, DC (1998)
19. Leavens, G., Baker, A., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Software Engineering Notes 31, 1–38 (2006)
20. Leavens, G., Cheon, Y.: Design by contract with JML (2006)
21. Leavens, G., Cheon, Y., Clifton, C., Ruby, C., Cok, D.: How the design of JML accommodates both runtime assertion checking and formal verification. Science of Computer Programming 55, 185–208 (2005)
22. Leino, K.R.M., Monahan, R.: Program verification using the Spec # programming system (2008), http://research.microsoft.com/en-us/projects/specsharp/etaps-specsharp-tutorial.ppt
23. Man Machine Systems: Design by contract for Java using JMSAssert (2009), http://www.mmsindia.com/DBCForJava.html
24. Martin, R.: The Liskov Substitution Principle. C++ Report 8, 16–17, 20–23 (1996)
25. Meyer, B.: Writing correct software. Dr. Dobb's Journal 14, 48–60 (1989)
26. Meyer, B.: Applying "design by contract". Computer 25, 40–51 (1992)
27. Meyer, B.: Object-oriented software construction, 2nd edn. Prentice-Hall (1997)
28. Meyer, B., Ciupa, I., Leitner, A., Liu, L.L.: Automatic Testing of Object-Oriented Software. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 114–129. Springer, Heidelberg (2007)
29. Microsoft Corporation: Code contracts user manual (2010), http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf
30. Object Management Group: Object constraint language version 2.2 (2010), http://www.omg.org/spec/OCL/2.2
31. Offutt, A.J., Xiong, Y., Liu, S.: Criteria for generating specification-based tests. In: ICECCS 1999: Proceedings of the 5th International Conference on Engineering of Complex Computer Systems, p. 119. IEEE Computer Society, Washington, DC (1999)
32. Tillmann, N., de Halleux, J.: Pex–White Box Test Generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
33. Turing, A.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, pp. 67–69 (1949)
34. Voigt, J.: Improving object oriented software contracts. Master's thesis, University of Canterbury, Christchurch, New Zealand (2011)
35. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)

Continuous Improvement of Business Processes Realized by Services Based on Execution Measurement

Andrea Delgado[1], Barbara Weber[2], Francisco Ruiz[3],
Ignacio García-Rodríguez de Guzmán[3], and Mario Piattini[3]

[1] Computer Science Institute, Faculty of Engineering
University of the Republica, Montevideo, Uruguay
adelgado@fing.edu.uy
[2] Quality Engineering Research Group, Computer Science Institute
University of Innsbruck, Innsbruck, Austria
barbara.weber@uibk.ac.at
[3] Alarcos Research Group, Technologies and IS Department
University of Castilla-La Mancha, Ciudad Real, Spain
{francisco.ruizg,ignacio.grodriguez,mario.piattini}@uclm.es

Abstract. Business Process Management (BPM) is being rapidly adopted by organizations wanting to focus on their business processes as key elements for controlling and improving the way they perform their business. The realization of business processes by services also helps in the improvement of their implementation, by decoupling the definition level from the technical one. In this article we present an approach to the continuous business process improvement based on execution measurement. It comprises an execution measurement model defining several measures for business process and service execution, and a continuous improvement process to guide the introduction of improvements. We have integrated several different approaches, techniques and methodologies in a single proposal to guide the improvement effort in an organization, providing support for business people from the definition to the analysis of the execution measures defined.

Keywords: Business process management (BPM), Continuous improvement of business processes, Execution measurement.

1 Introduction

The Business Process Management (BPM) [1][2][3] paradigm is being used increasingly in organizations to manage their business. The explicit modelling of business processes (i.e. using BPMN [4]) together with information regarding its execution constitute the main elements with which to compare the functioning of the organization as it moves towards achieving its business goals. The measurement of their business process execution is a key issue to be able to analyze its operation to see if business goals are being achieved. If they are not, the idea is to find improvement opportunities that would modify the business process so that it could reach the goals defined.

MINERVA framework [5][6] provides support for the business process lifecycle [1] and its continuous improvement, implementing them with services [7] using model driven development [8]. It is made up of three dimensions: conceptual [9], methodological [10][11] and tools [12]. It thereby integrates concepts, models, methodologies and processes for both development and improvement, and tools. In this article we extend the definition of MINERVA describing the Business Process Continuous Improvement Process (BPCIP) for guiding the improvement effort and the Business Process Execution Measurement Model (BPEMM) to guide the selection, implementation, collection, analysis and evaluation of execution measures for Business Process (BP) implemented by services. The rest of the article is organized as follows: in Section 2 the BPCIP is presented detailing its phases and activities, Section 3 describes the BPEMM along with an example of its use, Section 4 sets out related work and in Section 5 conclusions and future work are discussed.

2   BP Continuous Improvement Process (BPCIP)

The Business Process Continuous Improvement Process (BPCIP) is defined in the methodological dimension of MINERVA, and its main objective is to guide the improvement effort in the organization. It integrates the phases of the business process lifecycle in [1] and those from the continuous improvement process PmCOMPETISOFT in [13]. The measures defined in the Business Process Execution Measurement Model (BPEMM) are used to relate the BP execution to the organization's business goals explicitly, as well as to implement, register and asses the associated data. In Section 3 BPEMM will be described in detail. The general framework of BPCIP is shown in Figure 1.
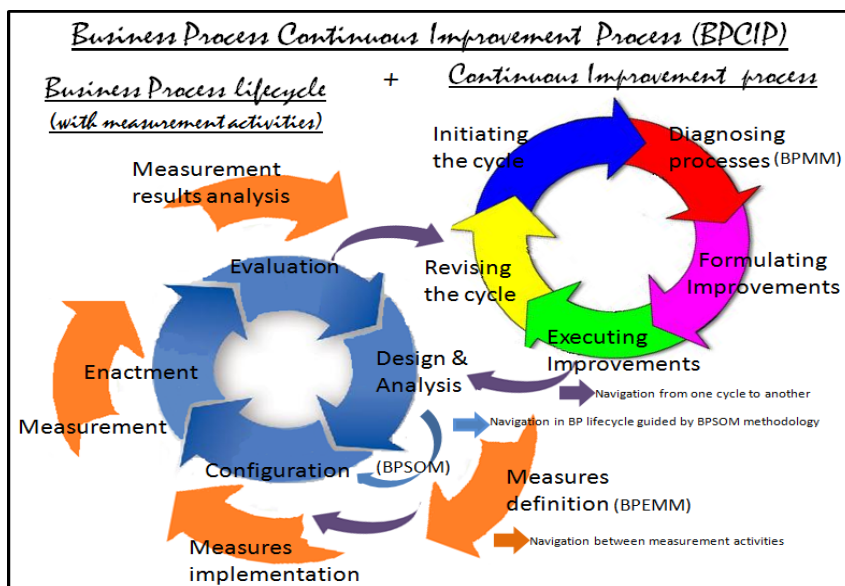

Fig. 1. MINERVA Business Process Continuous Improvement Process (BPCIP)

A complete execution cycle through the MINERVA framework begins with modeling a new BP or redesigning an existing one in BPMN, whose execution is then measured and evaluated, aiming to identify improvement opportunities. BPMN was selected for many reasons mainly as it is an OMG standard widely adopted and MINERVA is a standardized framework. These improvements can then be fed back into the business process following a systematic approach based on the continuous improvement process defined. Finally, the measures of the new version of the business process comprising the improvements made are compared with the previous version, to evaluate the results of the changes carried out. On the left side of Figure 1 the business process lifecycle (bottom left circle) defines four phases: Design & Analysis, Configuration, Enactment and Evaluation. For each of these phases we show explicitly the corresponding measurement activities in which the BPEMM is used (outer four arrowed circle). In addition, the arrow from Design&Analysis to Configuration indicates the use of the Business Process Service Oriented Methodology (BPSOM) to guide the implementation of business processes with services. On the right side the continuous improvement process (upper right circle) defines five phases: Initiating the cycle, Diagnosing processes, Formulating improvements, Executing improvements and Revising the cycle. Three arrows indicate the navigation from one cycle to another: from the Executing improvements phase of the improvement process to the Design &Analysis or the Configuration phases to re-enter BP lifecycle, and from the Evaluation phase to the Revising the cycle phase to return to the improvement process.

2.1   Process Phases and Activities

This section sets out the particular phases and the activities in executing a complete BPCIP cycle of MINERVA framework, as shown in Figure 1.

Design & Analysis. The cycle begins with the design and specification of a business process by means of BPMN models as part of the Design & Analysis phase. These models are then validated through simulation or analytical techniques to determine their relevance to the specified business goals, or to evaluate different design options for it. Moreover, to asses quality characteristics of the model created (i.e., complexity) as well as to detect potential problems in early stages, design measures not presented here can be used [14][15][16][17]. Finally, the BPEMM of MINERVA is used to select execution measures according both with the business objectives defined for the BP and the business strategy of the organization.


Fig. 2. Main activities in Design & Analysis phase

Configuration. In the *Configuration phase* the BPs are implemented by services with model driven development, guided by the BPSOM methodology (Delgado et.al, 2009a). BPSOM defines the disciplines Business Modeling, Services Design and Implementation with activities, input and output deliverables, and roles needed to carry out the service development starting from the BP that has been defined. In addition, QVT [18] transformations are defined and executed to generate SoaML [19] service models from BPMN models. The execution measures selected are implemented to be integrated directly into the process engine or into software systems, in the form of execution logs to register the information needed.
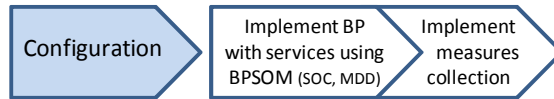

Fig. 3. Main activities in Configuration phase

Enactment. In the *Enactment phase* the BPs are executed in an appropriate process engine according to their implementation (BPEL/XPDL), from which to invoke the services realizing BP activities, sub-process or even the complete BP. The execution measures implemented are collected as BP cases (instances) are executed, registering the events and information needed for the execution measures to be calculated later.
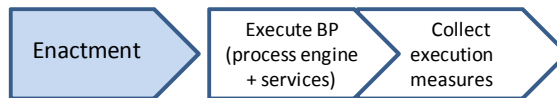
Fig. 4. Main activities in Enactment phase

Evaluation. The BP execution is then assessed in the *Evaluation phase* analyzing the measurement results. For this to be done, the execution measures are calculated on the basis of the information registered in the execution logs using the Process Mining [20] framework ProM [21]. By means of several plug-ins ProM allows different views of the associated information to be analyzed. Using the analysis performed it is possible to identify improvement opportunities for redesigning the BP, which can be related to the BP modelling level as well as to the software realizing the BP (implemented services), such as bottlenecks in the BP or service execution delays. For the redesign of the BP several existing approaches can be used [22][23][24].
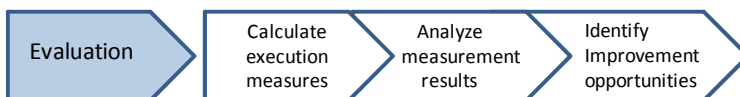
Fig. 5. Main activities in Evaluation phase

Initiating the Cycle. Once the improvement opportunities have been identified, the continuous improvement process to carry out the improvement effort is undertaken,

executing the corresponding phases. This implies the introduction of the improvements in a systematic way in order to assure the achievement of the results specified for the improvement of some or several BP characteristics. In the *Initiating the cycle* phase the improvements to be included in this iteration are established, including the BPs and the characteristics to be improved, as well as the results expected after the introduction of the improvements that have been defined. This can also lead to a revision of the execution measures chosen.
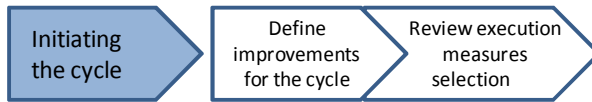

Fig. 6. Main activities in Initiating the cycle

Diagnosing Processes. In the *Diagnosing processes* phase other aspects of the BP definition (i.e., management) can be assessed using the OMG Business Process Maturity Model (BPMM) [25]. This standard which follows the format defined by the software maturity models (CMM, CMMI) includes several Process Areas and defined Key Activities that when performed, allows the BP to gain maturity by evolving through the model's five maturity levels. Based on this diagnosis new improvement opportunities for the BP can be found, which can be included in this iteration. For a description of the BPMM and BP measuring activities we refer the reader to [26].

Fig. 7. Main activities in Diagnosing processes

Formulating Improvements. The *Formulating improvements* phase aims to define how (by doing what) the selected improvements for this iteration will be introduced. To do so, the changes have to be defined specifically, i.e., if an activity in a BP has been identified as a bottleneck and its execution time should be improved, it could be specified that for this activity several redesigns must be evaluated to obtain better results. The same applies if the problem detected involves the execution of services which realizes the BP. In any case, the improvement to be made will be set out in detail in the associated improvement document.
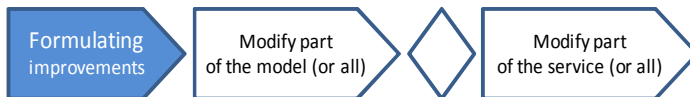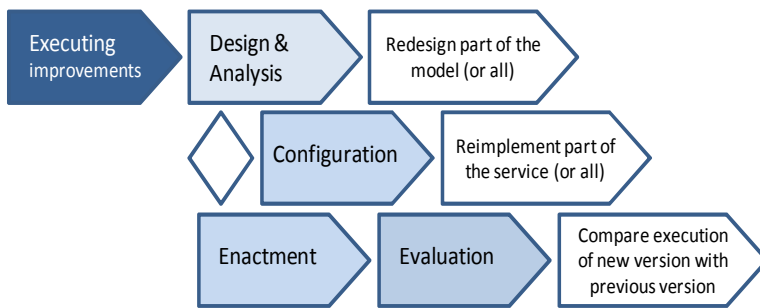

Fig. 8. Main activities in Formulating improvements

Executing Improvements. In the *Executing improvements* phase the BP lifecycle is re-entered exactly where the improvements have to be made. If the improvement refers to the BP model then the lifecycle is re-entered in the *Design & Analysis phase*,

where parts of the BP model or its entirety will be redesigned to introduce the improvements. Afterwards the whole BP lifecycle will again be executed with the new version of the BP. The existing traceability between the BPMN BP models and its implementation with services, will allow the identification of the impact of the changes in related services and/or other software artifacts in the Configuration phase.

On the other hand, improvement might only refer to the implementation of the BP (i.e., the BP model will not be changed but only the software realizing the BP). In that case the BP lifecycle will be re-entered in the *Configuration phase*, to implement changes in the services. Once the BP model and/or the services realizing it are modified, along with the implementation of the execution measures to be collected for the new version of the BP, this new version of the BP is executed registering the associated data in the specific execution logs. Finally, the defined activities are executed in the *Evaluation phase*, along with a comparison between measurement results from the new version of the BP and the previous version used as the basis for improvements. This comparison will also allow assessing if the goals set out for the improvement that has been brought in have been achieved.

Fig. 9. Main activities in Executing improvements

Revising the Cycle. In the *Revising the cycle phase* the data registered about the execution of the continuous improvement process itself is analyzed, to identify improvement opportunities in the improvement process also.


Fig. 10. Main activities in Revising the cycle

3   BP Execution Measurement Model (BPEMM)

Measurement of BP execution to analyze the achievement of business goals as well as to detect improvement opportunities is a key aspect in MINERVA framework. Although the execution measurement activities are not new, it is the BPEMM model

proposal. Its goal is to define a set of pre-defined measures for BP execution based on services, to support the improvement effort, relating business goals for the BP to its real execution, and helping in finding improvement opportunities. In Figure 11 the relation between BPCIP and BPEMM is shown, along with the BPEMM activities.

3.1   BPEMM Definition

BPEMM aims to help in relating business strategy and goals to business process implementation and execution, thus facilitating the selection of predefined execution measures for each business goal. BPEMM definition focuses on organizations which implement their BPs with services, proposing a set of execution measures from three defined views: generic BP execution (i.e., generic and domain specific measures for domains such as medical, software, production), focus on lean philosophy (e.g., eliminating waste and encouraging optimization), and services execution (i.e., for execution of services realizing the BP). These views were defined to cover as much information as possible to be able to get accurate knowledge from the execution logs of the BP, and then focus BP improvements on the specific parts of the BP. In addition to the views the dimensions defined by the "Devil's quadrant" [27][22]: time, cost, flexibility and quality, were taken into account. These dimensions refer to the trade-off that has to be taken into account when designing or redesigning a BP. For example, adding activities to improve the quality of the BP can have a negative impact on its performance. It is therefore important to collect information on the BP execution for each dimension, to analyze the improvements. Measures are organized in a three-level hierarchy. At the third level measures for the execution of each activity are registered. At the second level these measures are combined to calculate the BP case measures. Finally, in the first level case measures are combined to calculate the measures for the BP definition (e.g., averages, percentages, etc.).
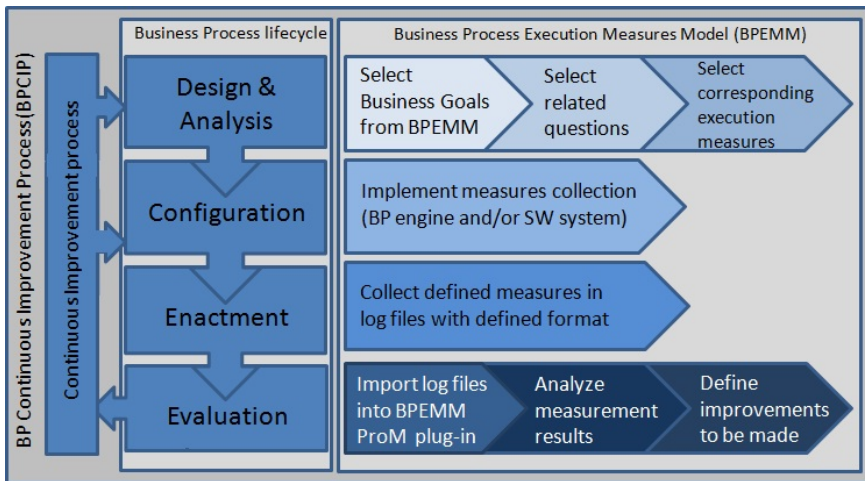
Fig. 11. Detailed measurement activities from BPEMM and its use in the BPCIP context
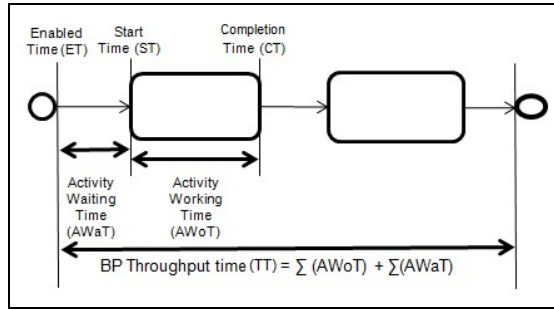
For the definition of the execution measures of the model, we used the Goal, Question, Metrics (GQM) paradigm [28] which is based on the idea that an organization must first specify its goals if it is to measure in a meaningful way what the organization does. It provides a systematic approach to establish and asses a set of operational goals based on measurement. It integrates goals with process models, products, resources and different perspectives, depending on the needs of the organization and project. Initially defined to evaluate defects in software projects, its use has been expanded to improvement efforts in software organizations. As our proposal includes a continuous improvement process that also comes from the software area, the use of GQM to define BPEMM is set in the same direction. BPEMM measures are then defined by three main elements:

• Goal: it is defined for the organization, section, project or process, from various points of view with respect to different models.
• Question: it is used to describe how each goal will be evaluated from the point of view of a quality characteristic.
• Metric: a set of data associated with each question to be answered quantitatively. Measures can be objective or subjective.

For the specification of BPEMM execution measures we use the Software Measurement Ontology (SMO) [29]. The SMO defines, among other concepts, different types of measures: base and derived measures and indicators, which are calculated by a measurement approach. In addition to the specification of GQM elements for each view and dimension in natural language, we model the execution measures and associated concepts in a graphical way. To do that, we use the SMTool [30] which implements SMO. It provides a quick overview of the measures set out to satisfy the information needs of the organization, helping in the communication with stakeholders. The execution measures views defined in BPEMM allow the measures to be organized according to the three relevant perspectives defined.

Generic BP Execution View. In the first view defined, the Generic BP execution view, the measures are related to BP characteristics that are common to all processes regardless of the associated domain, such as their duration or the duration of their activities, the associated cost, the roles involved, etc. However, some of these generic measures have to be instantiated for the BP domain, for example when they involve label definitions such as "successful branch", where activities comprising the branch have to be identified for each BP. Generally, these kinds of measures are specified as Key Performance Indicators (KPIs) by the business management area. The "Devils quadrant" dimensions are used to group these measures. We present as an example of this view of BPEMM, some execution measures defined in the Time dimension related to the BP performance, i.e., its Throughput Time (TT) or Cycle Time. This is defined, for a BP case (instance) as the total time incurred from the moment in which the case is initiated until it is completed [22][31]. Several different times are defined to calculate time measures such as: enable, start, change, suspension, queue, processing (or working), service, setup, waiting and completion time, for activities and cases [22][31][32][24]. From these we used as base measures for an activity, the enabled time (i.e., when an activity becomes available for execution), start time (i.e., when it actually starts its execution) and completion time (i.e., when an activity completes its execution). The explanation of the basic concepts about time measures defined for an activity and a BP case is shown in Figure 12.

Fig. 12. Time concepts definition for BP execution

Based on the base measures defined the derived measures and indicators are calculated as shown in Table 1. In addition to throughput time (TT), the generic execution measures view defines other measures for this and the rest of the dimensions of the Devil's quadrangle, which are not presented here.

Table 1. Generic BP execution view, time dimension measures sub-set

Goal	G1	Minimize the throughput time (TT) of the BP
Question	Q1	which is the actual throughput time of the BP
Metrics	M1 (base)	Enabled time of an Activity (ET)
	M2 (base)	Start time of an Activity (ST)
	M3 (base)	Completion time of an Activity (CT)
	M4 (derived)	Working time of an Activity (AWoT = CT – ST)
	M5 (derived)	Waiting time of an Activity (AWaT = ST – ET)
	M6 (derived)	Total Working time of a BP case (TWoT = $\sum$ (AWoT))
	M7 (derived)	Total Waiting time of a BP case (TWaT = $\sum$ (AWaT)
	M8 (derived)	Throughput Time of a BP case (BPTT = TWoT + TWaT)
	M9 (indicator)	Activity Working time vs. Waiting time index (ATI = AWaT/AWoT)
		Decision criteria=Index DC: R1: 0 <= TTI <=L1="LOW"→G; R2: L1 <=
	M10 (indicator)	TTI < L2="MEDIUM"→Y; R3: L2 <= TTI ="HIGH"→R
		Total BP Working time vs. Waiting time index (TTI =TWaT/TWoT)
	M11 (indicator)	Decision criteria = Index DC.
		Percentage of total BP Working time in total BP TT (PWoT=TWoT* 100/BPTT). Decision criteria = Percentage DC:
	M12 (indicator)	R1:0<=TTI<L1="LOW"→R;R2:L1<=TTI<L2="MEDIUM"→Y;R3:L2<=TTI<=100="HIGH"→G
	M13 (indicator)	Percentage of Total BP Waiting time in Total BP TT (PWaT=TWaT* 100/BPTT) Decision criteria = Inverse Percentage DC (G, Y, R)
	M14 (indicator)	Average BP Throughput Time for all BP cases (ABPTT = $\sum$ BPTT / Total BP cases) Decision criteria = Inverse Percentage DC (G, Y, R)
	M15 (indicator)	Average BP total Working time for all BP cases (ABPTWoT= $\sum$ TWoT/Total BP cases) Decision criteria = Percentage DC
		Average BP total Waiting time for all BP cases (ABPTWaT=$\sum$ TWaT /Total BP cases) Decision criteria = Inverse Percentage DC (G, Y, R)

As can be seen in Table 1 measures have been defined for the Goal "Minimize the TT of the BP" at three different levels: activity, BP case and set of all BP cases. By analyzing the measurement results for each level, improvement opportunities can be detected from global BP execution measures (i.e., average, percentage) to the

corresponding activities or BP parts that have to be changed to improve the TT of the BP. For indicators decision criteria have to be defined, i.e., the different ranks to which the measurement result can belong. To define the ranks we use labels that have to be changed to actual numbers for each BP and organization when selecting the execution measures (e.g. $0 <=$ Measurement result $<= L1$). This allows the ranks to be flexible enough to be used in different contexts using different numbers instead of the labels. Associated with the meaning of the ranks defined, we also use semaphores as supported in ProM. The semaphores show the meaning of the ranks by means of colors, where Green (G) means "OK", Yellow (Y) means "Warning" and Red (R) means "Problems". In Figure 13 some of the measures presented in Table 1 are shown graphically using the SMTool, which provides special icons for each concept defined in the SMO, its attributes, associations and restrictions defined between the ontology elements (i.e., rule for base measure, rule with figures for derived measures, rule with figures and lamp for indicators, among others).
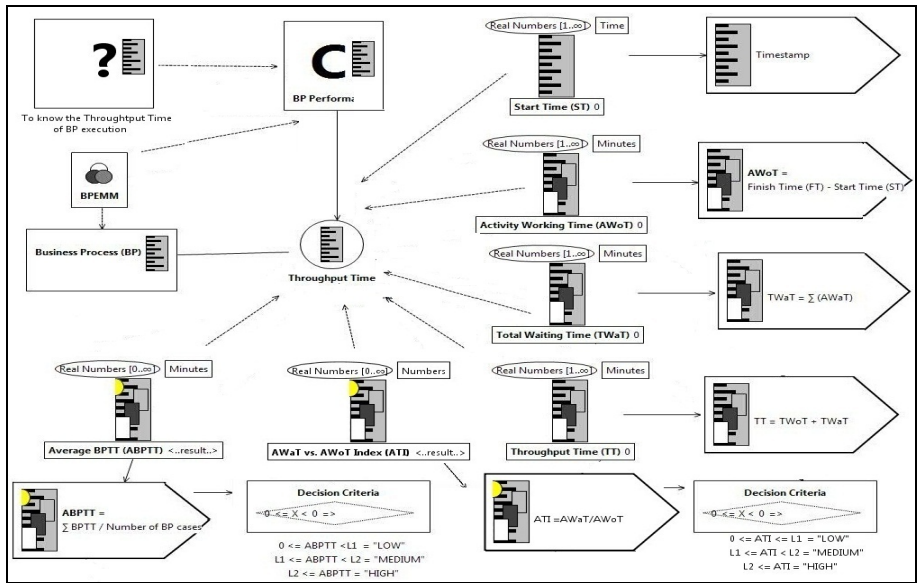
Fig. 13. Some measures from Table 1 shown graphically in the SMTool [30]

Other goals defined in the Generic BP execution view are, among others: "Minimize the cost of the BP" and "Minimize the use of resources for the BP" both of which correspond to the cost dimension, "Maximize the BP cases ending normally" (i.e., normal completion of the instance, successful or unsuccessful, with no abortion due to errors or user cancellation) corresponding to the quality dimension, and "Maximize the BP cases ending successfully" (i.e., executing the successful branch of the BP involving the execution of defined activities, such as making and paying for the reservation of flight, room and others in a travel agency BP) corresponding to a

domain specific execution measure. Table 2 shows an example of execution measures to be instantiated for domains.

Table 2. Generic BP execution view, time dimension measures sub-set

Goal	G1	Maximize the BP cases ending successfully (executing BP successful branch) of the BP
Question	Q1	which is the actual number of BP cases ending successfully
Metrics	M1 (base)	Successful branch execution of BP case (SB= branch with activity X)
	M2 (base)	Unsuccessful branch ex. of BP case (USB = branch with activity Y)
	M3 (derived)	Number of BP cases ending successfully (BPSB = count of SB)
	M4 (derived)	Number of BP cases ending unsuccessfully (BPUSB = count of USB)
	M5 (indicator)	Percentage of BP ending successfully in total BP cases (PBPSB= BPSB*100/TCBP) Decision criteria=Percentage DC:R1:0<=TTI <L1 ="LOW"→R;R2:L1<=TTI<L2="MEDIUM"→Y;R3:L2<=TTI<=100="HIGH"→G

Lean Execution View. The second BP execution view defined focuses on the Lean thinking philosophy, aiming to find elements in the BP that could be unnecessary or replaceable, or parts of the BP that if made as efficient as possible can lead to an optimization and improvement of the complete BP definition [31]. Lean thinking was first introduced in the Toyota Production System (TPS) and is based mainly on the identification and elimination of waste. It defines as key principles the specification of value from the customer viewpoint, the removal of waste, making valuable flow, delivering what the customer wants when it is wanted and pursuing perfection. There are seven types of waste defined: overproduction, waiting, transport, extra processing, inventory, motion and defects. These principles and waste types have been adapted to several areas other than the manufacturing sphere, such as lean software development [33], lean information management [34] and healthcare [35], among other realms, thus making lean thinking usable in several BP domains. As an example the GQM for the goal "Minimize the rework in loops of the BP" is shown in Table 3, which focuses on the detection of defects on the products or services delivered by the BP.

Table 3. Lean BP execution view measures sub-set

Goal	G1	Minimize the rework in loops of the BP
Question	Q1	which is the actual quantity of rework due to BP loops
Metrics	M1 (derived)	Activity rework in a loop (ARL = counts each execution in a loop)
	M2 (derived)	Activity Working time for the rework in a loop (AWoTRL = $\sum$(AWoTe$_i$) being e$_i$ each execution of the activity in the loop)
	M3 (derived)	Total Working time for the rework in a loop of the BP (TWoTRL = $\sum$(AWoTRLa$_i$) where a$_i$ represents an activity in the loop)
	M4 (derived)	Total Working time for rework in all loops of BP case (BPTWoTRL = $\sum$(TWoTRLl$_i$) where l$_i$ represents a loop in the BP)
	M5 (indicator)	Percentage of rework time in BP case due to loops in the total BP TT (PBPTWoTRL = BPTWoTRL*100/BPTT) Decision criteria =Percentage DC: R1: 0 <= TTI <L1 = "LOW"→G; R2:L1<=TTI<L2="MEDIUM"→Y;R3:L2<=TTI<=100="HIGH"→R

**Table 4.** Services execution view measures sub-set

Goal	G1	Guarantee response time to L1 seconds (label to be changed) for the service (implementing an activity/sub-process/process)
Question	Q1	which is the actual response time of the service
Metrics	M1 (derived)	Service processing time (SPoT=CT–ST)(idem AWoT for the service)
	M2 (derived)	Service latency time (SLaT = ST – ET) (idem AWaT for the service)
	M3 (derived)	Response Time of a service in a BP case (SRpT = SPoT + SLaT)
	M4 (indicator)	Average service Response Time in all BP cases (ASRpT = ∑ SRpT/ Total services execution in all BP cases) Decision criteria = Average DC:R1:0<=TTI<L1="LOW"→G;R2:L1<=TTI<L2="MEDIUM"→Y; R3:L2 <= TTI<=100="HIGH"→R

Services Execution View. Finally, the third view corresponding to the Services execution, aims to define measures to assess the execution of services realizing the BP. Several issues have to be taken into account to identify the most important features as regards Quality of Services (QoS) requirements specified in Services Level Agreement (SLA) [36][37]. To define these measures we used the Quality Attributes (QA) concepts for non-functional requirements and the taxonomies from [38][39][40]. Services measures then include quality attributes such as: performance (i.e., response time including processing time and latency, throughput, capacity), dependability (i.e.,availability, reliability), security (i.e., confidentiality, availability). Services execution measures defined for performance are related to the Generic execution measures for BP performance. They focus, however, on the automatic activities (i.e., tasks, sub-process, process) that are implemented by services, adding information about the execution of the specific software infrastructure. Table 4 presents the GQM for the Goal "Guarantee response time to L1 (i.e., label to be changed) seconds for the service", as an example of this.

4   Example

To give an example of the use of the BPEMM in the context of the BPCIP from the MINERVA framework we present the "Patient Admission and Registration for Major Ambulatory Surgery (MAS)" business process in Figure 14.

In the following we describe the possible execution of the improvement cycle based on the defined measures. The organization is the "Hospital" whose business management area we assumed has chosen, in the Design & Analysis phase, the set of execution measures of the Generic execution view for the time dimension and services execution measures (cf. Table 1, Table 4). Further assume that guided by the BPSOM methodology services to realize the BP have been implemented which will be externally invoked by other participants, and services have been defined to be invoked by the Hospital from other parties. In the Configuration phase assume the collection of chosen measures is implemented in the software for BP execution, and then the defined execution information is registered in the execution logs. Based on
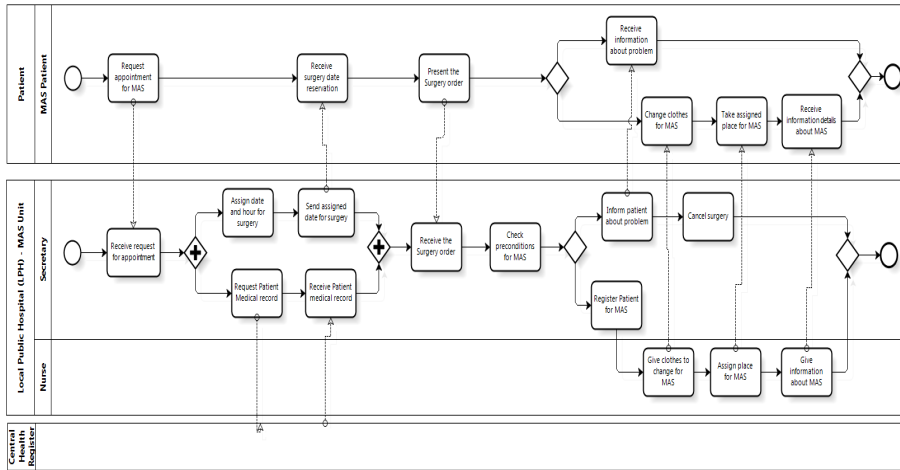
Fig. 14. "Patient Admission and Registration for MAS" specified using BPMN

the measures defined for calculating BP Throughput Time (TT) in BPEMM, times corresponding to base measures for BP activities have to be logged: enabled, start and finish time. Table 5 shows an example of some events related to the execution of activities simulating two BP cases. It can be seen that as defined, the specified times are registered for each activity, indicating to which event the timestamp corresponds (enabled, start, completed). Based on this information the execution measures for the BP Throughput Time (TT) are then calculated. Other information that can be registered corresponding to other execution measures defined such as the role or person/system performing the activity is not shown in the table.

Table 5. Example of execution logs information

BP case	Activity	Timestamp	Event
Case 1	Receive request MAS	10-01-2010: 09:30	Enabled
Case 1	Receive request MAS	10-01-2010: 09:30	Start
Case 1	Receive request MAS	10-01-2010: 10:00	Completed
Case 2	Receive request MAS	10-01-2010: 09:30	Enabled
Case 2	Receive request MAS	10-01-2010: 09:35	Start
Case 2	Receive request MAS	10-01-2010: 10:15	Completed
Case 1	Assign date for MAS	10-01-2010: 10:00	Enabled
Case 2	Assign date for MAS	11-01-2010: 10:15	Enabled
Case 2	Assign date for MAS	13-01-2010: 12:15	Start
Case 2	Assign date for MAS	13-01-2010: 12:45	Completed
Case 1	Assign date for MAS	13-01-2010: 12:45	Start
Case 1	Assign date for MAS	13-01-2010: 13:00	Completed
Case 1	Send assigned date for MAS	13-01-2010: 13:00	Enabled
Case 1	Send assigned date for MAS	13-01-2010: 13:02	Start
Case 1	Send assigned date for MAS	13-01-2010: 13:05	Completed
Case 2	Send assigned date for MAS	13-01-2010: 12:45	Enabled
Case 2	Send assigned date for MAS	13-01-2010: 12:46	Start
Case 2	Send assigned date for MAS	13-01-2010: 12:50	Completed

In the Evaluation phase based on the information registered in the execution logs, the defined execution measures can be calculated, for example:

- Average TT (ABPTT) = 8640 minutes (6 days)
- Case max.TT(BPTT) =21600 minutes (15 days)
- Case min. TT (BPTT) = 2880 minutes (2 days)

The Average TT for all BP case executions is 6 days instead of 4 days as defined by the business area for performing the BP. The maximum value of 15 days shows that there are cases which take significantly longer than 4 days. As these values are not the expected ones, other measurement results can be evaluated for BP case executions and for key activities of the BP. The M14 indicator of Average BP Working time (ABPTWoT) as well as the M10 indicator of the Index (TTI) between BP Total Working time vs. Total Waiting time, which are not shown due to space reasons, show that the TT of the BP is increased by waiting times in the execution of some activities. After analyzing the values for several BP cases, the M9 indicator of the index (ATI) between Working time and Waiting time of the activity "Assign date and hour for the surgery" is found to be in the rank "High" in 90%, i.e., the activity's waiting time is unreasonably high compared to its working time. Then, the origin of the BP execution problem is located in the activity mentioned, so an improvement effort with focus on this activity is initiated, to redesign the BP model.

The activities defined in the improvement process have to be performed then, to specify the improvements to be integrated in the cycle, how to integrate them into the BP, and finally to execute the particular improvements re-entering the BP lifecycle again. In this case, re-entry is in the Design & Analysis phase as the BP model has to be redesigned. To do so, there are approaches that propose different options [22][23][24]. In the example, one option could be to combine the activity with the one called "Send assigned date for surgery" in an activity of higher granularity, which performs both tasks automatically, thus eliminating the manual intervention in the first one. Figure 15 shows the two BP versions before and after the improvement.
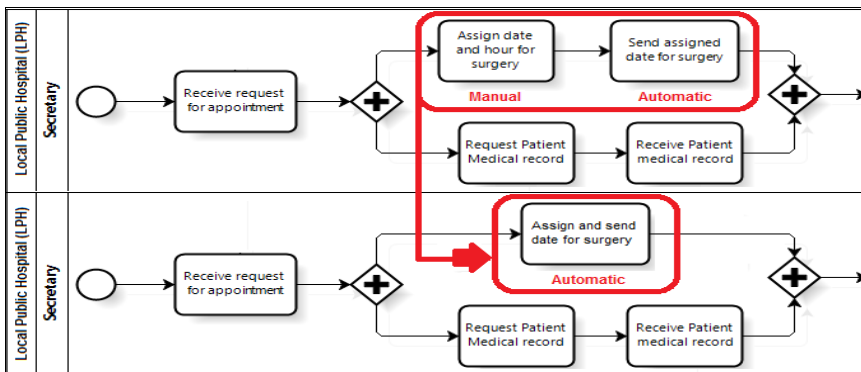
Fig. 15. Versions Comparison for "Patient Admission and Registration for MAS"

After selecting a redesign, a new version of the BP is generated and executed up to the calculation of the associated execution measures. Finally, the measures results for the new BP version are compared to the ones from the previous BP version, to assess whether the defined goals have been achieved with the improvement. In the example the goal is to reduce the BP Average Throughput Time (TT) from 6 to 4 days.

5   Related Work

Regarding BP execution measurement our definitions are based on the works on [22] [31][32][20] where several concepts and measures, are presented and analyzed. Process Mining [20] uses execution logs information to help finding BP models from BP execution, checking conformance between BP models and its execution, and extending BP models with execution information. Analytical techniques are used in [22][31] to analyze and predict BP performance and other BP characteristics, and simulation is also used in [31][24]. To redesign BP models based on improvement opportunities found several options are proposed in [22][24]. Using a data warehouse is proposed in [32] to store, analyze and evaluate BP execution. Design measures defined in [14][16][17] are complementary to ours and can be used to find improvements opportunities in earlier stages of the BP lifecycle. Several proposals exist from the business area but they focus mostly on the definition of Key Performance Indicators (KPIs) related to the flow and domain of the BP, not taking explicitly into account the infrastructure which realizes it. Some tools from the software area such as ProM [21] has a plug-in to make basic performance analysis based on measures defined. ARIS [42] has a Process Performance Manager (PPM) which also provides insight into performance and other BP execution measures, which have to be defined. Other techniques like Balance Scorecard [43] are proposed and used by the business area to align the BP with the strategic goals of the organization and to define the associated measures; a comparison with GQM can be seen in [44]. Several tools provide support to BSC.

6   Conclusions and Future Work

The MINERVA framework provides support for the continuous business process improvement based on its lifecycle management and its implementation by services with model driven development. It defines a BPCIP improvement process integrating explicit measurement activities into the BP lifecycle as well as a process to introduce improvements in a systematic way. Moreover, a BP execution measurement model (BPEMM) consisting of several BP execution measures is defined to be used in the BPCIP improvement process. BPEMM provides several execution measures related with the defined business strategy and goals, allowing the selection and implementation of execution measures regarding the needs of the organization. BPEMM execution measures are defined using the GQM paradigm, to provide traceability from business goals to execution measures, and visualized using SMTool.

Execution measures are defined for: time, cost, quality and flexibility dimensions for the views of generic BP and domain specific execution, lean focus and services.

The major contribution of the approach we have defined lies in the integration of all the different methods presented, including existing execution measures and new ones defined, to support the continuous improvement of BP implemented by services with traceability from business goals to software implementation. We believe that the approach presented contributes to the academic community working on BP execution measurement, as well as organizations wanting to improve the way in which they manage their BPs, providing an integrated approach to guide their execution measurement, analysis, evaluation and improvement efforts. As current and future work we are implementing the BPEMM as a ProM plug-in allowing the import of execution logs, the calculation of execution measures and the visualization of the results. This will be presented to the business management area, thus providing support in finding improvement opportunities with respect to the achievement of the specified business goals.

References

1. Weske, M.: BPM Concepts, Languages, Architectures. Springer, Heidelberg (2007)
2. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business Process Management: A Survey. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 1–12. Springer, Heidelberg (2003)
3. Smith, H., Fingar, P.: Business Process Management: The third wave, Meghan-Kieffer
4. OMG (2008a), BP Modeling Notation (BPMN) (2003)
5. Delgado, A., Ruiz, F., García-Rodríguez de Guzmán, I., Piattini, M.: MINERVA: Model drIveN and sErvice oRiented Framework for the Continuous Business Process improVement and relAted Tools. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave 2009. LNCS, vol. 6275, pp. 456–466. Springer, Heidelberg (2010)
6. Delgado, A., Ruiz, F., García-Rodríguez de Guzmán, I., Piattini, M.: A Model-driven and Service-oriented framework for BP improvement. Journal of Systems Integration (JSI) 1(3) (2010)
7. Papazoglou, M., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenge. IEEE Computer Society (2007)
8. Mellor, S., Clark, A., Futagami, T.: Model Driven Development-Guest eds.int. IEEE Computer Society (2003)
9. Delgado, A., Ruiz, F., García-Rodríguez de Guzmán, I., Piattini, M.: Towards an ontology for service oriented modeling supporting business processes. In: 4th. IC Research Challenges Inf. Sci. (RCIS 2010). IEEE (2010)

10. Delgado, A., Ruiz, F., García-Rodríguez de Guzmán, I., Piattini, M.: Towards a Service-Oriented and Model-Driven framework with business processes as first-class citizens. In: 2nd IC on Business Process and Services Computing, BPSC 2009 (2009)
11. Delgado, A.: García-Rodríguez de Guzmán, I., Ruiz, F., Piattini, M.: From BPMN BP models to SoaML service models: a transformation-driven approach. In: 2nd Int. Conf. on Sw. Tech. Engineering, ICSTE 2010 (2010)
12. Delgado, A., García-Rodríguez de Guzmán, I., Ruiz, F., Piattini, M.: Tool support for Service Oriented development from Business Processes. In: 2nd Int. Work. Model-Driven Service Engineering, MOSE 2010 (2010)
13. Pino, F.J., Hurtado Alegría, J.A., Vidal, J.C., García, F., Piattini, M.: A Process for Driving Process Improvement in VSEs. In: Wang, Q., Garousi, V., Madachy, R., Pfahl, D. (eds.) ICSP 2009. LNCS, vol. 5543, pp. 342–353. Springer, Heidelberg (2009)
14. Rolón, E., García, F., et al.: Evaluation Measures for Business Process Models. In: 21st Symposium on Applied Computing (SAC 2006) (2006)
15. Cardoso, J.: Process control-flow complexity metric: An empirical validation. In: IEEE International Conference on Services Computing, SCC 2006 (2006)
16. Mendling, J.: Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness. Springer, Heidelberg (2008)
17. Sánchez González, L., García, F., et al.: Assessment and Prediction of Business Process Model Quality. In: 18th Int. Conf. Cooperative Inf. Systems, CoopIS 2010 (2010)
18. Object Management Group (OMG), Query/Views/Transformations (QVT) (2008)
19. Object Management Group (OMG), Soa Modeling Language (SoaML), beta 2 (2009)
20. van der Aalst, W.M.P., Reijers, H.A., Medeiros, A.: Business Process Mining: an Industrial Application. Information Systems 32(5) (2007)
21. ProM, Process Mining Group, Eindhoven University of Technology, The Netherlands
22. Reijers, H.A.: Design and Control of Workflow Processes. LNCS, vol. 2617. Springer, Heidelberg (2003)
23. Maruster, L., van Beest, N.: Redesigning business processes: a methodology based on simulation and process mining techniques. Knowl. Inf. Syst. Journal (2009)
24. Netjes, M.: Process Improvement: The creation and Evaluation of Process Alternatives. Eindhoven UT (2010)
25. Object Management Group (OMG), BP Maturity Model (BPMM), v.1.0 (2008)
26. Sánchez, G.L., Delgado, A., Ruiz, F., García, F., Piattini, M.: Measurement and Maturity of BP. In: Cardoso, J., van der Aalst, W. (eds.) Handbook of Research on BP Modeling. Inf. Science Ref., IGI Global (2009)
27. Brand, N., Van der Kolk, H.: Workflow Analysis and Design (1995)
28. Basili, V.R.: Software Modeling and Measurement: The GQM Paradigm, CS-TR-2956, University of Maryland (1992)
29. García, F., et al.: Towards a Consistent Terminology for Software Measurement. Inf. and SW Tech. 48 (2005)
30. Mora, B., García, F., Ruiz, F., Piattini, M.: SMML: Software Measurement Modeling Language. In: 8th Int. Work. Domain-Specific Modeling, OOPSLA 2008 (2008)
31. Laguna, M., Marklund, J.: BP Modeling, Simulation and Design. Prentice Hall (2005)
32. zur Muehlen, M.: Workflow-based Process Controlling, Foundation, Design, and Application of Workflow-driven Process IS. Logos Verlag (2004)
33. Poppendieck, M.: Principles of Lean Thinking, Poppendieck. LLC (2002)
34. Hicks, B.J.: Lean information management: Understanding and eliminating waste. Int. Journal of Information Management (2007)

35. Jimmerson, C., Weber, D., Sobek, D.: Reducing waste and errors: Piloting Lean Principles at Intermountain Healthcare. Journal Quality & Patient Safety (2005)
36. Wetzstein, B., Karastoyanova, D., Leyman, F.: Towards Management of SLA-Aware BP Based on Key Performance Indicators. In: 9th Work. BP Modeling, Development and Support, BPMDS 2008 (2008)
37. Cardoso, J., Sheth, A., Miller, J.: Workflow quality of service. In: Int. Conf. on Enterprise Integ. and Mod. Tech., Int. Enterprise Mod. Conf, ICEIMT/IEM 2002 (2002)
38. O'Brien, L., Bass, L., Merson, P.: Quality Attrs. and SOA, CMU/SEI-20055-TN-014. SEI (2005)
39. Clements, P., Kazman, R., Klein, M.: Evaluating SW Archs: Methods and Case Studies. Addison Wesley (2001)
40. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison Wesley (2003)
41. Barbacci, M., Klein, M., et al.: Quality Atributtes, CMU/SEI-95-TR-021, SEI (1995)
42. ARIS, IDS Scheer, Software AG, Germany
43. Kaplan, R.S., Norton, D.P.: The balanced Scorecard Measures that drive performance. Harvard Business Review 10(1) (1992)
44. Buglione, L., Abran, A.: Balance Scorecards and GQM: what are the differences? In: FESMA-AEMES Software Measurement Conference (2000)

Structure Editors: Old Hat or Future Vision?

Andreas Gomolka and Bernhard Humm

Hochschule Darmstadt – University of Applied Sciences
Haardtring 100, 64295 Darmstadt, Germany
`andreas.gomolka@gmail.com, bernhard.humm@h-da.de`

Abstract. Structure editors emphasise a natural representation of the underlying tree structure of a program, often using a clearly identifiable 1-to-1 mapping between syntax tree elements and on-screen artefacts. This paper presents layout and behaviour principles for structure editors and a new structure editor for Lisp. The evaluation of the editor's usability reveals an interesting mismatch. Whereas by far most participants of a questionnaire intuitively favour the structure editor to the textual editor, objective improvements are measurable, yet not significant.

Keywords: Programming, Structure editor, Evaluation, Lisp, Eclipse.

1 Introduction

Structure editors have fascinated designers of development environments for decades [1, 2, 3, 4]. The idea is simple and convincing. The elements of the syntax tree of a program are mapped to on-screen artefacts and can be edited directly.

The basis for this is the awareness that programs are more than just text [5]. A programmer designing a piece of code thinks in structures: classes, methods, blocks, loops, conditions, etc. Using a textual program editor he or she has to codify those syntactically using parentheses such as '{...}', '(...)', '[...]' or using keywords such as 'begin ... end'. The compiler then parses the syntactic elements and re-creates the structures in the form of an abstract or concrete syntax tree – the same structures which the programmer originally had in mind. This just seems inefficient and not intuitive.

Structure editors fill this gap: What the programmer thinks is what he or she sees in the editor. Surprisingly enough, structure editors, although around for decades, have never become mainstream. So somehow, there has to be a catch in this quite simple and straight forward idea. In this paper, we try to find out whether it may be possible to avoid the drawbacks of former implementations and whether structure editors are maybe more than an "old hat" – perhaps even a "future vision" of programming environments?

To be able to answer this question, we analyse the requirements of a usable structure editor and describe layout and behaviour principles for structure editors. Based on this, we present a new structure editor for Lisp and an evaluation of the editor's usability based on a questionnaire – with interesting results.

The remainder of the paper is structured as follows: Section 2 describes layout and behaviour principles for structure editors. In Section 3, we present a new structure editor for Lisp via samples and screenshots and give some insights into its implementation. Section 4 describes how we evaluated the usability of the editor and in Section 5 we position our work in relation to other approaches. Section 6 concludes the paper with a critical discussion.

2 Layout and Behaviour Principles

A structure editor should improve the readability and comprehensibility of the code whilst not compromising useful features of textual editors. To this end, we postulate the following layout and behaviour principles for structure editors:

1. Focus on the Net Code. The code layout should support the programmer in focussing on the net program code, i.e., keywords, identifiers, and literals. The structure of the code should be visualized in a clear but discrete manner. A look into the related literature reveals that there is no overall agreement, which kind of representation fits this intention. Dimitriev, for example, states that programmers always translate program text to tree structures in their mind [6] and argues that editors should emphasise this view. In contrast, Edwards claims that tree structures are not satisfying to display conditionals and therefore proposes to visualize programs using tables [7]. We think that the representation should emphasise the structure of the program, but also enable the programmer to recognise the original code. Therefore, we propose, similar to the approach of Ko and Myers [4], to replace syntactic elements for structuring the code (e.g., parentheses for block structures, separators like semicolons, and delimiters like double quotes for string literals) by graphical elements.

2. Do not Restrain the Programmer. The editor should help, but not unnecessarily restrain the programmer. For an editor it is only possible to visualize the structure of the program correctly if it does not contain any syntactical errors. Some former approaches handled this problem by preventing the creation of syntactical errors at all [1, 2]. This had the effect that simple operations which change the structure of the program became quite complex, e.g., removing a parenthesis and inserting it somewhere else. It is essential for the usability of a structure editor how it handles this problem.

3. Keep the Layout Compact. Apart from editing, a programmer uses an editor also for reading and understanding a piece of code. The structured representation should support the programmer in quickly getting an overview of the whole program. Therefore, the structured representation should be as compact as the plain text representation.

4. Keep Common Look and Feel. The behaviour of the structure editor should be as similar as possible to the look and feel of widely used editors. Examples are shortcuts, colouring, and behaviour during typing. This facilitates getting accustomed with it for experienced programmers.

5. Do not Introduce New Dependencies. A structure editor is just one of many more tools to work with a program. The textual form of a program makes it easy to change between different editors. This independence should not be dismissed without a good reason. Thus, a structure editor should not necessitate changes to the programming language or the way programs are stored.

6. Make the Layout Configurable. Where possible, the programmer should be able to configure the presentation of the code. For example, colours that are used in the layout should be configurable.

7. Leave the Choice to the Programmer. Some programming tasks might be easier to achieve with a simple structure editor, some with an advanced structure editor and yet others with a textual editor. Therefore, the programmer should be able to freely and easily swap between different editors respectively editor modes.

3   A Structure Editor for Lisp

This section presents a new structure editor for the programming language *Lisp* that was developed as a research prototype. It follows the principles we proposed above.

3.1   Why Lisp?

The main reason why we decided to build the research prototype for Lisp – or, to be more precise, *Common Lisp* – is Lisp's uniform syntax. Lisp data is expressed as a so called *S-expressions* [8]. The term S-expression means *symbolic expression* and includes symbols and nested lists. As there is no syntactical difference between data and code, a Lisp program also consists of S-expressions. This simplicity and uniformity and the ability to treat Lisp code as data make it particularly easy to develop a structure editor for Lisp.

Also, in a different research context, we use Lisp as a base language for developing *domain-specific languages* (DSLs) in the context of *language-oriented programming* [9]. A structure editor may be particularly useful for developing programs using DSLs that are based on Lisp.

3.2   Code Presentation

The structure editor is based on the Eclipse plug-in *CUSP* [10]. CUSP already provides an environment for developing Lisp programs using Eclipse including a *Navigator View* for browsing Lisp projects, a *REPL* (*Read-Eval-Print-Loop*) and an *Outline* of the currently displayed Lisp file. The new structure editor has been integrated into this environment as an additional *Editor Window* (see Fig. 1).

The Editor Window consists of two separate representations of the code. Besides the structured representation, we also provide a textual one. The user is able to switch between these two using the tabs at the lower left corner of the editor window.

The following Figures 2-4 demonstrate the different possibilities of viewing the code that are provided. All three figures show the same snippet of Lisp code defining a new function called "hello-world" which prints a string n times.
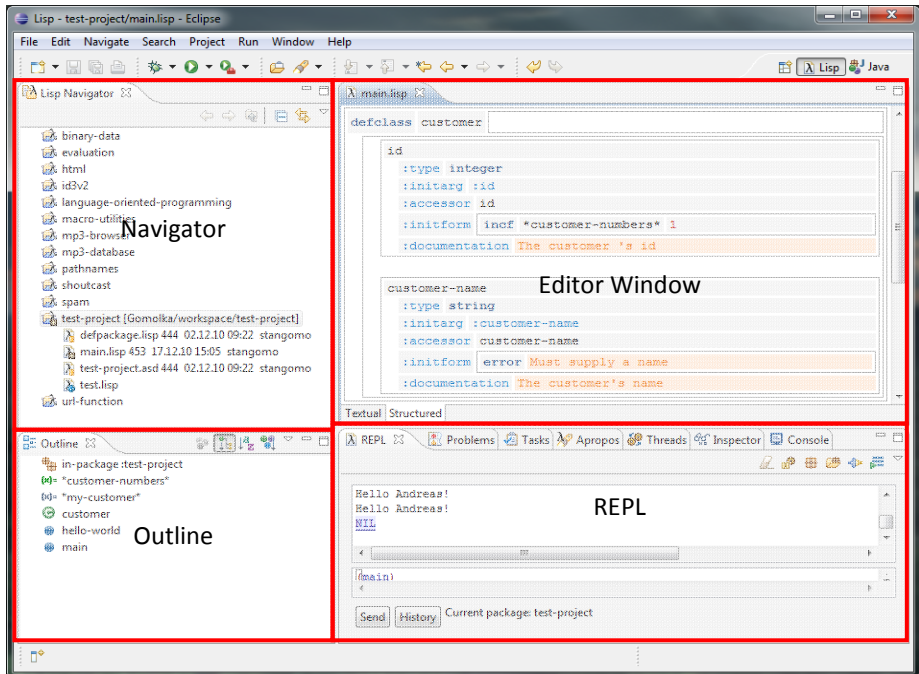
Fig. 1. Overview of the structure editor GUI
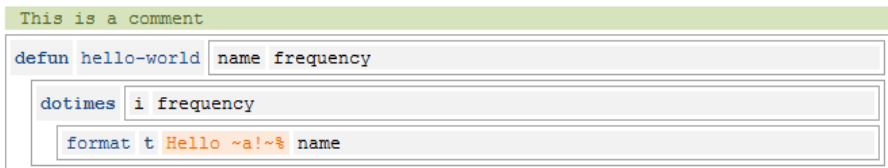

Fig. 2. Textual representation

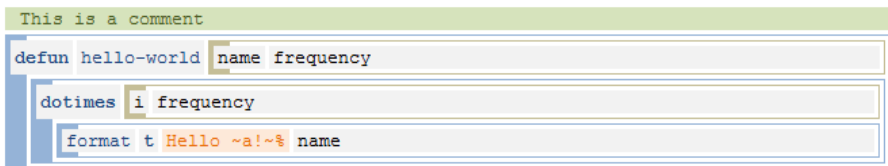

Fig. 3. Default structured representation

Fig. 4. Coloured structured representation

Fig. 2 shows the snippet using the textual representation. The structure of the code is visualized by the indention of the lines and the individual symbol types (e.g., keywords, string literals, comments, etc.) are indicated by different colours.

Fig. 3 shows the same snippet displayed in the structure editor. All parentheses are replaced by grey boxes which visualize the block structure. Also, the double quotes delimiting the string literals are hidden and expressed by the light orange background. Similarly, the leading semicolons marking the beginning of a comment are hidden and the comment is indicated by the light green background. All this removes syntactic delimiters from the code and accentuates the net code, which satisfies Principle 2 in Section 2.

A slightly different representation of the same code snippet is shown in Fig. 4. There, in addition, coloured bars are displayed at the left side of each box and the boxes themselves are also coloured. The colours indicate whether a block contains a call to a function or macro (e.g., "defun") or just an ordinary list (e.g., the parameter list of the function "hello-world").

According to Principle 70, the programmer may decide which representation to use and enable or disable the additional information expressed by those colours via the preferences menu. Furthermore, all colours to be used (background and foreground) can be configured (Principle 6).

3.3   Editing
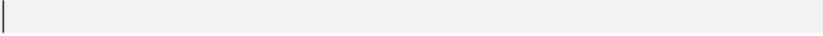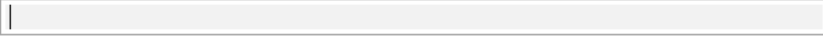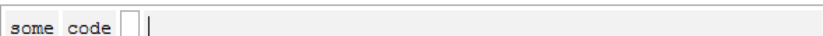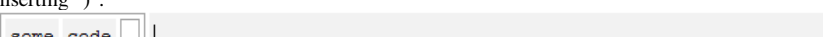
The code can be edited directly in the nested block structure. There are no additional commands or shortcuts necessary compared to editing the code in the textual representation. As shown in Fig. 5, typing an opening parenthesis will open a new box. Typing a closing parenthesis will close the current box and move the caret outside. In each step, the layout rearranges itself according to the changes. This satisfies our claim to enable the programmer doing the same typing as using a textual editor (Principle 4).

The caret can be moved around using the mouse or the keyboard. The arrow keys will move it one character to the left or right or one line up or down. Using <Tab> respectively <Shift><Tab>, it is moved one field forward backward. <Pos1> will place the caret at the beginning of the first field of the current line and <End> at the end of the last field.

The programmer may decide about line breaks or blank lines. They will be inserted by typing <Return>. Each new line is inserted to the current block. Line indention is calculated automatically depending on the context of the current block, because this is part of the block structure.

The structure editor provides code completion which also shows additional information about the selected symbol as shown in Fig. 6. As common in Eclipse, this is invoked using <Ctrl><Space>.

Common actions like undo/redo or cut, copy and paste may be called via the "Edit" menu or by using the usual shortcuts, for example <Ctrl><C> for "copy" (Principle 4).

1. Empty row:

2. Inserting "(":

3. Inserting the string "some":

    some|

4. Inserting <Space>:

    some |

5. Inserting the string "code":

    some code|

6. Inserting "(":

    some code |

7. Inserting ")":

    some code |

8. Inserting ")":

    some code |

9. Inserting <Return>:

    some code

Fig. 5. Behaviour during typing

defc

defclass
defconstant
defconstant-uneql
defconstant-uneql-name
defconstant-uneql-new-value
defconstant-uneql-old-value

(defconstant NAME VALUE OPTIONAL DOCUMENTATION)
Documentation for the symbol DEFCONSTANT:

Function:
Define a global constant, saying that the value is constant and may be
compiled into code. If the variable already has a value, and this is not
EQL to the new value, the code is not portable (undefined behavior). The
third argument is an optional documentation string for the variable.

Fig. 6. Code completion

3.4   Implementation

The implementation of the new Editor Window containing the structure editor is based on the Graphical Editing Framework (GEF) provided by Eclipse [11].

GEF applies the Model-View-Controller (MVC) design pattern that explicitly separates the data structures themselves and the way they are displayed in the user interface. GEF is designed in a generic way so that any kind of model can be used. In our case, the model is the syntax tree that was parsed from the Lisp code. We extended the parser that came with CUSP to enrich the individual tree elements (e.g., to distinguish between different kinds of symbols like function names and keyword symbols).

Each model element is mapped to a figure which visualizes the different type of expression or symbol. Each change which is done using the structured representation in the user interface is reflected to the model. In some cases, an operation causes more than one change. For example, typing an opening parenthesis changes the whole structure because the following elements have to be moved into a newly created box. These modifications are performed in the model and afterwards the affected elements of the view are adjusted accordingly.
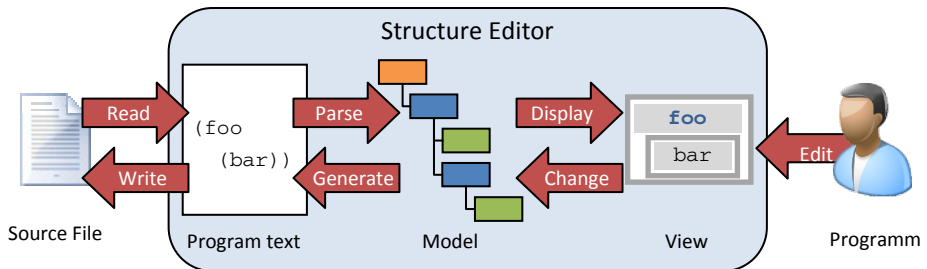
Fig. 7. Changing the model using the structure editor

Fig. 7 displays the whole process of editing a piece of code using the structure editor. First, the text is read from the source file and directly parsed to get the corresponding syntax tree. This is mapped to the figures that represent the individual elements of the tree. As mentioned before, each change which is done by the programmer is reflected back to the model. The corresponding Lisp code is not touched until the user saves the current document or changes to the textual representation. This means, using the structure editor, the programmer directly works on the syntax tree of the program.

The editor takes care of performing editing operations only if they result in a valid syntax tree. For example, it is not possible to paste code that contains unbalanced parentheses. If this is necessary, the programmer may circumvent this restriction (Principle 2) by switching to the textual representation to fix the appearing parsing errors. The code that was edited using the structure editor will not contain any structural parsing errors at all.

The following numbers give an impression of the extent of the implementation of the editor. The first one describes the newly created part of the plug-in (including some code that was taken from GEF samples) and the second one also incorporates the code of the already existing CUSP-plug-in.

Lines of code (structure editor):   8,290
Lines of code (entire plug-in):   25,571

4   Evaluation

In order to evaluate the usability of the structure editor in comparison to a common textual editor we conducted a survey.

4.1   Survey Preparation

Following Dumas and Redish, we presume that "usability means that the people who use the product can do so quickly and easily to accomplish their own task" [12, p.4].

We defined that the task we analyze by this evaluation is to understand the meaning and the structure of a piece of Lisp code – in other words: how the structure editor supports the readability and comprehensibility of the code. Considering that the users just need to read a piece of code, we decided to conduct a survey in terms of examining screenshots of the editor.

In literature, there are many metrics for analyzing the usability of software such as *effectiveness*, *efficiency*, *measures of learning*, and *subjective usability* [13], [14]. We focused on measuring the efficiency and a subjective rating of the usability.

To this end, three questionnaires were composed. Two of them show screenshots showing a piece of Lisp code and ten multiple-choice questions related to the meaning of the displayed code. We produced two versions of each questionnaire: one containing a screenshot of the code in textual representation and one containing a screenshot of the structure editor. This made the results comparable. In the third questionnaire, the participants were asked to rate how they experienced code reading in the two different representations and to give statements about things they liked or disliked in the screenshots of the structure editor.

4.2   Conducting the Survey

We conducted the survey with two different groups of participants. The first group consisted of second semester Bachelors' students (37 people). They had not known Lisp before. The second group was a team of Masters' students (13 people) who were engaged in a development project using Lisp and Prolog.

Both groups were randomly (according to their last names) divided into two groups and each group got one version of the first questionnaire. After exactly five minutes the students were told to stop working and to mark how far they got in answering the questions. For the second questionnaire, the groups were swapped: the group that worked on a questionnaire containing screenshots of the textual editor first then got the ones containing screenshots of the structure editor and vice versa. Again, the students had five minutes time to answer the questions. Finally, the students answered the third questionnaire.
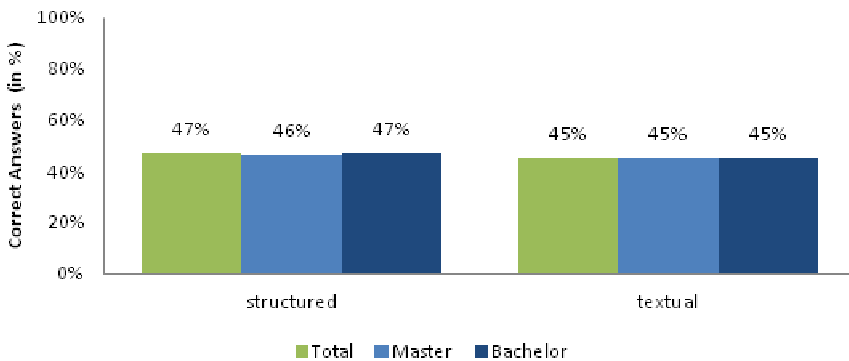
Fig. 8. Efficiency results

4.3   Results

As explained in Section 4.2, the first two questionnaires contained questions for comparing the efficiency in reading and understanding code in the two different representations.

Fig. 8 shows the cumulated results of this part of the survey in terms of the percentage of correct answers. As one can see, the results using the structure editor are slightly better (2%) but there is no significant difference.

We also examined how many questions the students managed to answer in the rather short period of five minutes. Fig. 9 shows the results. The students working with the structure editor did a bit better but, again, the difference is not significant.
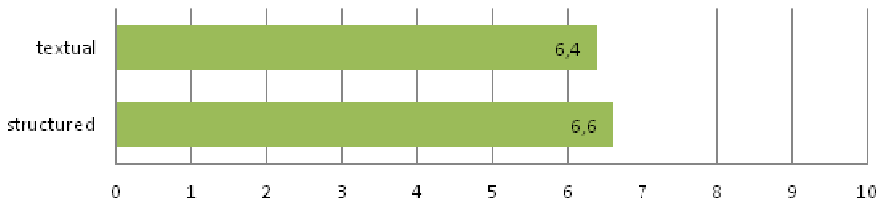
Fig. 9. Number of finished questions
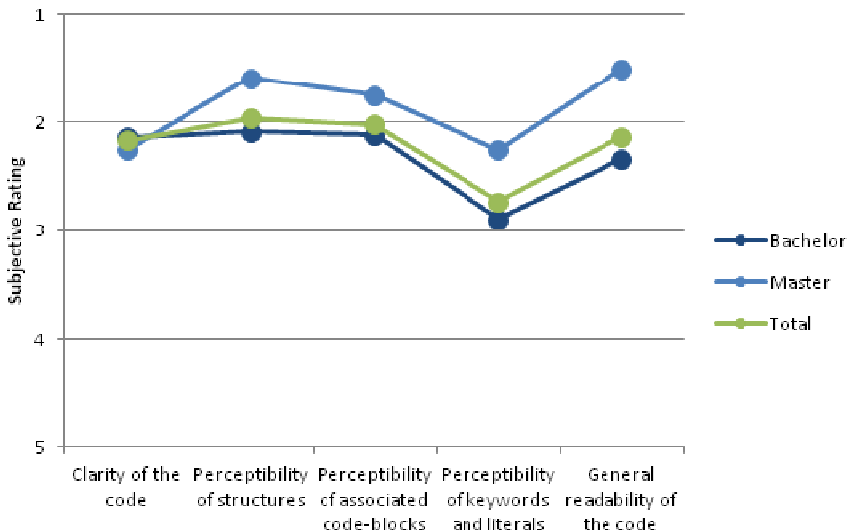
Fig. 10. Subjective rating of the structure editor

In the third questionnaire, the participants were asked to rate the structure editor compared to the textual editor regarding:

- Clarity of code
- Perceptibility of structures
- Perceptibility of associated code blocks

- Perceptibility of keywords and literals
- General readability of the code

The rating was possible within a range from "significantly better" (1) to "significantly worse" (5). A value of 3 means "no difference". Fig. 10 shows the result. All ratings are in the positive half of the spectrum. Most ratings are close to 2 which means "better". The Master students who were already experienced in working in Lisp gave better rates than the Bachelor students.

Most of the statements the participants gave about what they liked regarding the structure editor pointed in a similar direction. Several people wrote something like "code is clearly arranged" or "the structure is clearly visible". However, a few people contrarily stated that they were confused by the structured representation of the code.

In general, the diagram indicates the subjective feeling of the participants that the structure editor helps them reading and understanding the code better.

As a last question, we asked the participants whether they would use such a structure editor if there was one for their favourite programming language. Fig. 11 shows that the majority (61% in total, 82% of the Master students) would at least give it a try. Students that voted negatively argued that they got used to their current editor and do not want to spend time in learning how to use a different one.
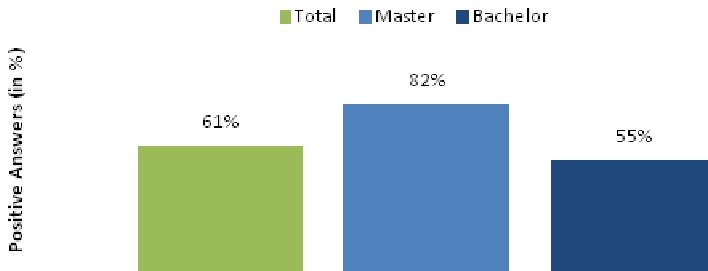
Fig. 11. Question "If there were a structure editor for your favourite programming language - would you use it?"

The evaluation does not reveal significant benefits of the structure editor as one may have expected. Nevertheless, it shows an interesting mismatch between the subjective ratings of the participants in the third questionnaire and the actual results from the first two questionnaires. This will be discussed in Section 6.1.

5   Related Work

We are not the first ones thinking about visualising the structure of a program in the editor and directly working on the syntax tree that was created from the code. In this section we present other approaches that were developed to achieve these goals.

5.1   Early Structure Editors

The idea of an editor which visualizes the structure of the underlying code is not new. In 1971, Wilfred J. Hansen presented a system called "Emily" [1] which was, in fact,

a structure editor for PL/I. The basic idea was to create a program by recursive replacement of placeholders according to their role in the Backus-Naur Form (BNF) notation of the programming language. The structure of the program and even of every command was fixed by structures of placeholders. Emily physically stored the whole program in a hierarchical structure that supported descending into sub-structures along the hierarchy. From the programmer's point of view, it was technically not possible to create programs that contained syntactical errors.

Other systems that follow a similar approach are "MENTOR" [15] and the "Cornell Program Synthesizer" [5]. Particularly in the Lisp community, programmers were fascinated by the idea of working directly on the structure of the code instead of a textual representation. An example of a structure editor for Lisp is Interlisp-D [16].

However, these early structure editors that are mostly summarized as *syntax-directed editors* could not satisfy the expectations and did not become widely accepted. Looking at these ancient examples which ran on terminals, restricted the programmers in several ways (violating Principle 2) and were quite tedious to use compared to a textual editor, this seems comprehensible. But what about newer systems based on the same idea?

5.2   Program Tree Editor

A more recent example of a structure editor of a different flavour is the "Program Tree Editor" [17]. This system visualises a piece of code written in a common programming language as a tree, similar to a file browser. It supports C, C++, C#, Java, Java Script, J#, XML, XHTML. Each tree node represents a structure from the underlying code and can be contracted and expanded. The tree structure is created upon opening a file containing source code and is translated back to the textual representation when a file is saved.

The user navigates through the tree using the keyboard and is able to edit the individual nodes directly in the tree. Nodes can be added or removed without the need of a mouse. Features like auto completion are provided.

This type of editor literally implements working on the underlying tree structure of the code. However, we question that this kind of visualization is particularly useful. We believe that programmers do not think in such file browser-like tree structures when they program. They more likely think in block structures. This is why we designed the GUI of our structure editor in a different way, consisting of nested boxes that emphasise the structure in a more discrete, but nevertheless clear way.

5.3   Subtext

A totally different approach of representing the structure of a program is presented as a system called *Subtext* [7]. Subtext is not based on an existing programming language. Instead, it introduces its own programming language that is not based on a textual representation of code any longer but stores its code in a database.

Using Subtext, the programmer composes programs from combining so called *schematic tables* which the author of the system describes as "a cross between

decision tables and data flow graphs" and which are intended to replace all kinds of conditional constructs. The basic idea behind such schematic tables is to visualize the structure of the program in two-dimensional way. The horizontal axis contains the different cases of a conditional statement ("deciding") and the vertical axis determines what happens if the individual cases become active ("doing").

Subtext seems to be quite an interesting approach for visualizing decision structures such as nested case statements. The greatest drawback appears to be its lack of compatibility. Subtext cannot be used to visualize the structure of already existing programs written in a common programming language (Principle 5).

5.4   A Structure Editor for C#

The most similar approach to our structure editor we are aware of is a *Structured Editor for C#* [18]. We regard it as the most capable editor of the ones we compared. This editor also represents the structure of the code in a discreet way by coloured bars at the beginning of each line. The actual bounds of a code block are shown as soon as one clicks on it using the mouse. All syntactic delimiters like curly brackets and semicolons are hidden, because they are not needed any longer.

One difference is, that the programmer is forced to change his way of typing. The delimiters are not only hidden, they are also not typed at all. For example, for entering the body of a C# class, the programmer has to press the <Return> key instead of typing a curly bracket. Our philosophy is that the programmer may type exactly the same code with the textual and the structural editor in order to minimize the learning curve and to easily switch between editors (Principle 4).

5.5   Structure Editors and Language-Oriented Programming

All structure editors that were mentioned so far try to be an alternative or extension to the textual editors that are normally used to read and edit programs. In different ways they visualize the structure of a program. In the context of language-oriented programming (LOP) [6, 9] there is one more step of abstraction where structure editors can be useful.

One main idea of LOP is to enable domain experts to contribute more directly to the programmatic solution of a problem by using a suitable Domain-Specific Language (DSL). Such a DSL may be an extension to an existing programming language (*internal DSL*) or a new language (*external DSL*), not necessarily a textual one. In the latter case, the program is created using a special kind of structure editor (sometimes called a *projecting editor*) that also performs a mapping from the DSL to an executable program. In this case, the structure editor is more than just an alternative view on the program – it is actually part of the language workbench.

A language workbench that provides a complete development environment for external DSLs is the "Meta-Programming-System" [6] by JetBrains, which includes an "editor language" to create structure editors for each newly developed DSL. Another example is the system called "Intentional Software" that was proposed by Simonyi [19].

6   Conclusions

In this paper we described layout and behaviour principles for structure editors and presented a new structure editor for Lisp. We also presented an evaluation of our editor. Taking this into account, we now try to answer the question: Are structure editors an old hat or a future vision?

6.1   An Interesting Mismatch

Structure editors have been around for decades. However, they have not succeeded in replacing classical textual program editors. We think that this is an interesting mismatch: on the one hand, the concept of displaying the underlying structure of a program and directly working on the syntax tree is intuitively attractive. On the other hand, this kind of editor has gained low acceptance in practice so far.

The results of our survey revealed this mismatch, too. The majority of the participants had the intuitive feeling that the structure editor was superior to the textual editor. However, the quantitative results showed no significant improvement. All this seems to suggest that structure editors are rather "old hat" than "future vision".

Certainly, structure editors are no silver bullet for software engineering [20]. Understanding the concepts of a programming language or paradigm is far more difficult than coping with a particular syntax. For example, a programming novice who has understood the concept of classes, inheritance, and polymorphism will not have a major problem in getting acquainted with different syntaxes, be it curly or other parentheses or, instead, boxes in a structure editor. Insofar, one should not expect an extraordinary measurable improvement in usability.

The structured representation even has a drawback which most of the former implementations of structure editors were not really able to handle: Structure editors require a syntactically correct program to be able to determine the structure of the code and to display the structured representation. So, modifications that are quite small but lead to a change of the structure (e.g. moving a bracket from one line to another) are not possible without the support of the editor. So even though the idea of a structure editor itself is quite simple – the implementation is not.

Also, many programmers are reluctant to change their way of programming. Our survey confirmed this opinion. Some participants conceded that the structure editor might be useful, but they got accustomed to their favourite IDE and do not want to change tools without really having to. The benefit seems to be too small for most programmers.

This is why we integrated the structure editor into a popular IDE like Eclipse and also provided the textual editor as part of the plug-in. As a result, the programmers may just use the structured representation where this seems helpful – and perhaps find out that this applies in more cases than expected.

6.2   Structure Editors Are Still Useful

Taking into account the arguments from the previous section, structure editors most likely will not be able to replace textual editors that are embedded in powerful IDEs. However, we still feel that they can be useful.

A first step is, as just mentioned, to plug structure editors into an IDE like Eclipse and offer programmers the possibility to use it as an alternative to the textual editor. For example, the programmer may use the structured representation for reading and understanding the code because it provides a better overview. To edit the code, he or she then may switch to the textual perspective. Anyway, this kind of use would not justify describing structure editors as "future vision" of programming environments.

However, we see a growing field of special-purpose programming issues where, indeed, structure editors could provide a significant improvement: configuring an application, defining business rules, designing the layout of a GUI, specifying a business process, etc. For all those special-purpose programming issues, DSLs are becoming more and more popular. The ever-increasing number of XML dialects is an indication for this. We feel that structure editors are particularly useful for programming DSLs, or, in general, for Language-Oriented Programming (see Section 5.5).

Fig. 12 provides an example of a code snippet in some XML dialect in textual form and in the structure editor. The representation in the structure editor is by far more clearly arranged than in the textual XML syntax. This is particularly useful for novices or rare users of this particular XML dialect.
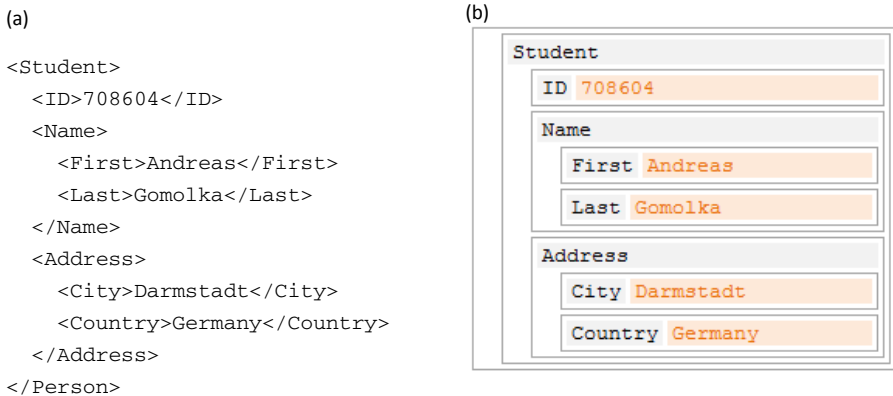
Fig. 12. A snippet of XML code in textual (a) and structured representation (b)

Conway et al. [21] and Myers et al. [22] have shown in comprehensive analyses that structure editors and graphical editors are most useful for programming novices, e.g., children. Since users of special-purpose DLSs are usually rare users and often novices we are confident that structure editors may be most useful in this context: a future vision for DSL editors.

6.3   Future Work

As future work, we plan to extend our evaluation of the structure editor towards its use in Language-Oriented Programming. In addition to readability and understandability of code we will examine the effects of the editor on the learning curve for DSLs as well as the effectiveness and efficiency of programming.

A program editor is a tool and no silver bullet. In the end, it is a matter of taste which kind of editor a programmer feels most appropriate for achieving a task – and this is a case for structure editors.

References

1. Hansen, W.J.: User engineering principles for interactive systems. In: Proceedings of the 1971 Fall Joint Conference (AFIPS 1971), pp. 523–532 (1971)
2. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: the system. In: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 14–24 (1988)
3. Ballance, R.A., Graham, S.L., van de Vanter, M.L.: The Pan language-based editing system. ACM Transactions on Software Engineering Methodology (TOSEM) 1, 95–127 (1992)
4. Ko, A.J., Myers, B.A.: Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2006), pp. 387–396 (2006)
5. Teitelbaum, T., Reps, T.: The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. Communications of the ACM 24, 563–573 (1981)
6. Dimitriev, S.: Language Oriented Programming: The Next Programming Paradigm. onBoard 1 (2004)
7. Edwards, J.: No Ifs, Ands, or Buts - Uncovering the Simplicity of Conditionals. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA 2007 (2007)
8. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, Part I. Communications of the ACM 3, 184–195 (1960)
9. Humm, B.G., Engelschall, R.S.: Language-oriented Programming via DSL Stacking. In: Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFT 2010), pp. 279–287 (2010)
10. Jasko, T., Ritchey, T.: CUSP. A Lisp plugin for Eclipse,
    `http://www.bitfauna.com/projects/cusp/`
11. The Eclipse Foundation: GEF and Draw2d Plug-in Developer Guide,
    `http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.`
    `gef.doc.isv/guide.html`
12. Dumas, J.S., Redish, J.C.: A practical guide to usability testing. Intellect Books, Exeter (1999)
13. Bevan, N.: Measuring usability as quality of use. Software Qual. J. 4, 115–130 (1995)
14. Schalles, C., Rebstock, M., Creagh, J.: Ein generischer Ansatz zur Messung der Benutzerfreundlichkeit von Modellierungssprachen. In: Engels, G., Karagiannis, D., Mayr, H.C. (eds.) Modellierung 2010, Klagenfurt, Austria, März 24-26, vol. 161, pp. 15–30. GI, Bonn (2010)
15. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B.: Programming Environments based on Structured Editors: The MENTOR Experience. Institut National de Recherche d'Information et d'Automatique (INRIA), Rocquencourt (1980)
16. Burton, R.R., Masinter, L.M., Bobrow, D.G., Haugeland, W.S., Kaplan, R.M., Sheil, B.A.: Overview and status of DoradoLisp. In: Proceedings of the 1980 ACM Conference on LISP and Functional Programming (LFP 1980), pp. 243–247 (1980)

17. Yurov, A.: Program Tree Editor, `http://www.programtree.com/`
18. Osenkov, K.: Designing, implementing and integrating a structured C# code editor. Brandenburg University of Technology, Cottbus (2007)
19. Simonyi, C., Christerson, M., Clifford, S.: Intentional software. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2006 (2006)
20. Brooks, F.P.: No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer 20, 10–19 (1987)
21. Conway, M., Audia, S., Burnette, T., Cosgrove, D., Chistiansen, K.: Alice: lessons learned from building a 3D system for novices. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2000), pp. 486–493 (2000)
22. Myers, B.A., Pane, J.F., Ko, A.: Natural programming languages and environments. Communications of the ACM 47, 47–52 (2004)

A Framework for Aspectual Pervasive Software Services Evaluation

Dhaminda B. Abeywickrama and Sita Ramakrishnan

Faculty of Information Technology
Monash University, Clayton Campus, Australia
Dhaminda.Abeywickrama@gmail.com, Sita.Ramakrishnan@monash.edu

Abstract. Context-dependent information has several qualities that make *pervasive services* challenging compared to conventional Web services. Therefore, *sound software engineering practices* are needed during their development, execution and *validation*. This establishes an evaluation framework to evaluate pervasive service-oriented software architectures. The framework consists of two views: vertical, horizontal. Vertical evaluation compares several research tools to the `Aspectual FSP Generation tool` developed here. They are compared across the platform-independent and platform-specific levels of the architecture. The horizontal evaluation view is designed to validate several desired key features mainly required at the platform-specific level. The vertical evaluation has demonstrated that our tool has unique features in context-dependent behavioral modeling and code generation. The horizontal evaluation has shown that the formal methods and tools employed, and the customization approach used in the services, are effective towards the overall objectives of this research. The approach is explored using a real-world case study in intelligent transport.

Keywords: Pervasive services, Model-driven development, Model checking, Aspect-oriented modeling.

1 Introduction

A *pervasive service* is a special type of service that adapts its behavior or the content it processes to the context of one or several parameters of a target entity in a transparent way (e.g. restaurant finder services, attractions and activities recommendation services) [1]. With the proliferation of ubiquitous computing devices and Internet, pervasive services continue to evolve from simple proof of concept implementations created in the laboratory to large and complex real-world services developed in industry. Context-awareness capabilities in service interfaces introduce additional challenges to the software engineer. Context information is characterized by several qualities that make pervasive services challenging compared to conventional Web services, such as a highly dynamic nature, real-time requirements, quality of context information and automation. The additional complexities associated with these special services necessitate the use of *solid software engineering methodologies* during their development, execution and *validation*. Most state-of-the-art approaches to pervasive services relate to the

detailed design or implementation stages [2,3] of the software life-cycle, such as pervasive Web services. Little work focuses on the early phase of design such as architecture design, which is of a higher level and abstract in design.

This systematic, architecture-centric approach [4,5,6,7] for modeling and verifying pervasive services integrates the benefits of sound software engineering principles such as model-driven architecture, aspect-oriented modeling, and formal model checking. It adopts model-driven development to represent complex crosscutting context-dependent functionality in service interfaces in a modular manner, and to automate the generation of state machine-based adaptive behavior. The crosscutting context-dependent information of the interacting pervasive services is modeled as aspect-oriented models in UML. Aspect-oriented modeling [8] is an aspect-oriented software development extension applied to the early stages of the software life-cycle, which supports separation of concerns at the modeling level. Using model transformations (`Aspectual FSP Generation` tool), we ensure the correct separation of concerns of the crosscutting context-dependent functionality at both semi-formal UML modeling and formal behavioral specification levels. The generated context-dependent adaptive behavior and the core service behavior for the pervasive services are rigorously verified using formal model checking against specified system properties.

This paper establishes a framework to evaluate pervasive service-oriented software architectures. The method of evaluation is based on key features comparison. The evaluation framework consists of two dimensions or views: vertical and horizontal. The vertical evaluation compares several research tools to the `Aspectual FSP Generation tool` developed in the current research. The tools are compared across the platform-independent model (PIM) and platform-specific model (PSM) levels of the model-driven architecture (MDA). The horizontal evaluation view is designed to validate several desired key features that are mainly required at the PSM level of the aspectual pervasive software services specification. These criteria mainly cover two aspects: formal methods and tools employed, and the context and adaptation dimensions of the customization approach used in the services. The vertical evaluation has demonstrated that the `Aspectual FSP Generation tool` has unique features in context-dependent behavioral modeling and code generation. The horizontal evaluation of the approach has shown that the formal methods and tools employed, and the customization approach used in the services are indeed effective towards the overall objectives of this research. The approach is explored using a real-world case study in intelligent tagging for transport.

The rest of the paper is organized as follows. Section 2 provides background information on this study. An overview of the evaluation framework established here is provided in Section 3. In Section 4 vertical evaluation of the research is discussed while Section 5 addresses the horizontal evaluation. Section 6 concludes this paper.

2   Background

This section provides background information on (1) the overall pervasive services engineering process, (2) the case study, and (3) the context-dependent adaptive behavior generation process applied in the research.

2.1   Pervasive Services Engineering for Service-Oriented Architectures

The overall pervasive service-oriented development process is divided into three stages (Figure 1) [6,7]. First, using the case study we extract use cases and define a service specification for the system under consideration using message sequence charts. Second, the architecture for the system is defined using a component configuration and an architecture model in Finite State Processes (FSP) using the LTSA-MSC tool. Third, the architecture model synthesized from the previous step is modularized with aspect-oriented models in UML called the `contextual-FSP aspects` (`c-FSP aspects`), and automatically transformed into FSP before applying model checking using the Labeled Transition System Analyzer tool (LTSA).

2.2   Case Study: Awareness Monitoring and Notification Pervasive Service

The research approach is explored using a real-world case study in intelligent tagging for transport known as the ParcelCall project. ParcelCall [9] is a European Union project within the Information Society Technologies program. The case study describes a scalable, real-time, intelligent, end-to-end tracking and tracing system using radio frequency identification (RFID), sensor networks, and services for transport and logistics. This case study is particularly appealing to the current research as it provides several scenarios for representing software services that interoperate in a pervasive, mobile and distributed environment. A significant subset of the ParcelCall case study is exception handling that needs to be enforced when a transport item's context information violates acceptable threshold values. The reference scenario used here describes an `awareness monitoring and notification pervasive service`, which alerts with regards to any exceptional situations that may arise on transport items, primarily to the vehicle driver of the transport unit. The threshold values for environment status (e.g., temperature, pressure, acceleration) of transport items and route (location) for the vehicle are set by the carrier organization in advance. The service alerts if items' environment status exceeds acceptable levels or if an item is lost or stolen during transport. The primary context parameters modeled in the study include item identity, location, temperature, pressure and acceleration.

2.3   Context-Dependent Adaptive Behavior Generation

The *notion of context* used in this research is based on a definition provided by Analyti *et al.* [10] for context in information modeling. They describe context as a set of objects, each of which is associated with a set of names and another context called its reference. Furthermore, they enhance the definition for context by stating that each object of a context is either a simple object or a link object (attribute, instance-of, ISA) and each object can be related to other objects through attribute, instance-of or ISA links. Analyti *et al.* [10] use traditional object-oriented abstraction mechanisms of attribution, classification, generalization and encapsulation to structure the contents of a context.

The model transformation tool created in our study is called the `Aspectual FSP Generation tool`. The transformations have been applied to the reference scenario in intelligent transport. We use model transformations to automate the application of
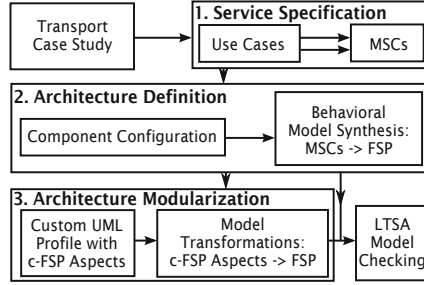
Fig. 1. Pervasive services engineering for service-oriented architectures

design patterns and generate infrastructure code for the `c-FSP aspects` using FSP semantics. The current study explores the strengths of both semi-formal UML meta-level extensions and formal finite state machines for representing the context-dependent behavior of software services, and model transformation techniques are applied as a bridge to enforce correct separation of concerns between these two design abstractions. The main benefits of this approach are: improving the quality and productivity of service development; easing system maintenance and evolution; and increasing the portability of the service design for the pervasive services engineer.

This approach focuses on the application of model-driven development for engineering pervasive services at finite state machine level. An aspect in FSP can be identified as an independent finite state machine that executes concurrently and synchronizes with its base state machine. In general, an aspect in FSP needs to contain synchronization events (transitions) to coordinate with its base state machine and other aspects. Also, each aspect type (e.g. `context`, `trigger`, and `recovery`) contains its unique constructs which can be generated automatically using model transformation techniques. For example, a `trigger aspect` requires constructs to alert and send notifications while a `recovery aspect` needs constructs to recover from exception-handling situations. On the other hand, a `context aspect` has attribution, instance-of, ISA, and reference constructs from the notion of context applied here. In Figure 2, the models and activities of the development process are represented as ellipses and square boxes respectively. The development process is structured into three main flows of activities. `Flow 1` and `Flow 2` extensively apply model transformations where `Flow 1` uses a model-to-text JET transformation and `Flow 2` applies an effective pipeline of model-to-model and model-to-text JET transformations. Both `Flow 1` and `Flow 2` originate from the `c-FSP-UML profile`. `Flow 3` represents activities involved for rigorously verifying the context-dependent adaptive behavior and the core service behavior of the pervasive software services using formal model checking.

3   Evaluation Framework

This evaluation framework [4] mainly validates the main contributions or deliverables of this study against several key evaluation criteria. The main tools used in this study include the `Aspectual FSP Generation tool` created in this research, the LTSA
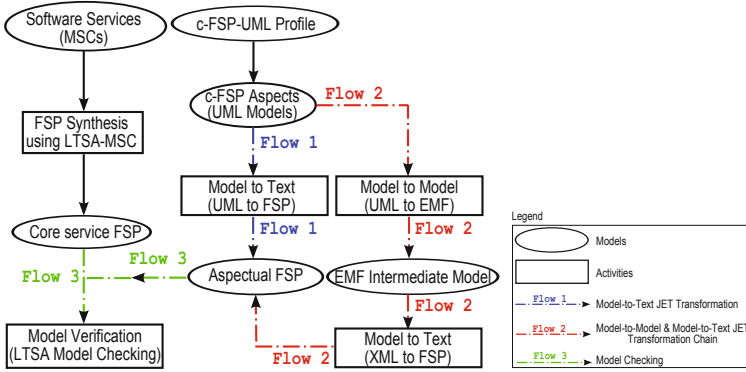
Fig. 2. Context-dependent adaptive behavior generation process

model checker and the LTSA-MSC tool. First, this research has developed a custom tool (`Aspectual FSP Generation tool`) applying an effective pipeline of model-to-model and model-to-text JET transformations using the IBM Rational Software Architect to automate the application of design patterns and generate infrastructure code for the *aspects*. Second, this research has performed rigorous specification and verification of concurrent models of pervasive software services and their compositions using the LTSA model checker to assure the correctness and quality of the pervasive services design. Third, in this study the interaction patterns defining the pervasive software services of the specification have been modeled using the LTSA-MSC tool, from which a core service model was extracted. The evaluation framework established in this paper intends to validate the aforementioned three main deliverables against key evaluation criteria.

The method of evaluation used here is based on key features comparison. The evaluation framework developed here does not produce additions to the research methodology but instead validates the methods and tools used in the research as a whole. The evaluation framework consists of two views: *vertical* and *horizontal*. Figure 3 illustrates the two views of the overall evaluation approach.

4 Vertical Evaluation

This section discusses the vertical view of the evaluation framework [4]. In the vertical view of the evaluation framework, several tools on aspect-oriented modeling (AOM) are compared with the `Aspectual FSP Generation tool` as a whole at the PIM and PSM levels of model-driven development. As the `Aspectual FSP Generation tool` covers several modeling layers of the MDA stack such as PIM and PSM levels, the evaluation process is essentially vertical in nature. The vertical evaluation essentially provides an analysis of several features of the `Aspectual FSP Generation tool` against several aspect-oriented modeling-based tools. Like the `Aspectual FSP Generation tool`, these tools have been developed using commercially available toolchains of similar area of application such as IBM Rational Software Modeler, Borland Together, Telelogic Modeller and Topcased [11]. This evaluation is based on the
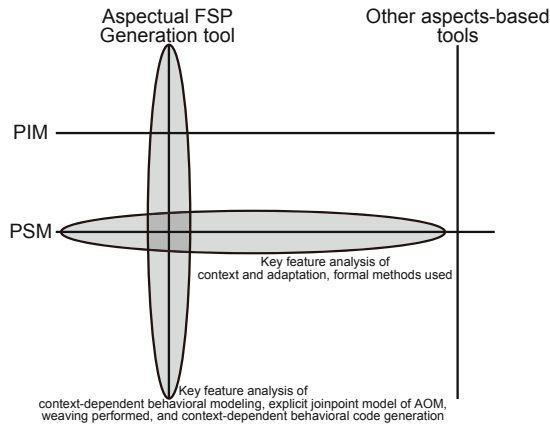
Fig. 3. Evaluation framework: vertical and horizontal views

following criteria: context-dependent behavioral modeling at the PIM level, explicit joinpoint model of AOM at the PIM level, weaving performed at the PIM level or PSM level, and context-dependent behavioral code generation from the PIM level to PSM level. A particular evaluation criterion can be fully satisfied (complete cover), partly satisfied (partial cover), or not supported at all (no cover). The results of this evaluation are presented in Table 1.

4.1 Aspectual FSP Generation Tool

The `Aspectual FSP Generation tool` created in this research [4,5,6,11] using IBM Rational Software Architect provides for context-dependent behavioral modeling at the PIM level, and context-dependent behavioral code generation from the PIM level to the PSM level of model-driven development. This tool effectively applies model-driven development in pervasive services engineering at the state machine level. Context-dependent behavior at the service interface level has been modeled using a custom UML profile called the `c-FSP-UML profile`, and aspect-oriented UML class models called the `c-FSP aspects`. The profile supports an explicit joinpoint model of AOM at the PIM level, and towards this the profile defines several stereotypes such as `Aspect`, `ContextAspect`, `TriggerAspect`, `RecoveryAspect`, `Advice` and `Pointcut`. The `Aspectual FSP Generation tool` can be employed to generate PSMs in formal behavioral specification level in FSP using an effective chain of model transformations. Model transformations are employed here to automate the application of design patterns and generate infrastructure code for the `c-FSP aspects` using FSP semantics. Also, using transformations the correct separation of concerns both at UML modeling and FSP behavioral specification levels is ensured. The main benefits of this approach are: improving the quality and productivity of service development; easing system maintenance and evolution; and increasing the portability of the service design. Weaving between an aspect and a base state machine is performed using an explicit weaving mechanism at the executable state machine level in FSP. The

context modeling and transformations features of the `Aspectual FSP Generation tool` have been explored using the reference scenario on intelligent transport.

4.2 Groher and Schulze's Approach

Groher and Schulze [12] present an approach for specifying crosscutting concerns using aspect-oriented modeling and discuss the seamless integration of those models to implementation. In their approach, UML has been customized for supporting aspect-oriented modeling using UML's standard extension mechanisms, such as stereotypes, tagged values and constraints. Their design notation for aspect-oriented modeling provides a base package for modeling the business logic, an aspect package for modeling the crosscutting concerns, and a connector (weaving rules) for linking the aspect and the base elements. The connector includes program execution points (pointcuts), and actions to be executed at those points (advices). The weaving in their approach is essentially performed at the PIM level and not at the PSM level as in the current study. The authors have implemented an AspectJ code generator using the CASE tool *Together* from Borland. In their tool aspect-oriented validation and code generation in AspectJ have been implemented as modules. The authors' work [12] is not based on pervasive services, which is a key difference to the current research. Also, the code generation is at the implementation level with AspectJ whereas in the current research it is at the software architectural level with FSP.

4.3 Whittle and Jayaraman's Approach

Whittle and Jayaraman [13] present a UML-based aspect-oriented modeling tool called MATA that applies graph transformations for specifying and composing aspect models. Their work is different to most other approaches on aspect-oriented modeling in three respects. First, there is no support for explicit joinpoints and composition is considered as a special form of model transformations. Second, the use of graph transformations for aspect composition, and third, support for statically analyzing aspect interactions using critical pair analysis, make their approach different to other approaches. The composition of a base model and an aspect model (weaving) is specified using a graph rule. A main difference in the authors' approach is that graph rules have been defined over the concrete syntax of the modeling language and not at the meta-level as in most known approaches on model transformations. The authors use a cell phone example to demonstrate the validity of their method and tool. Tool support for MATA has been built using IBM Rational Software Modeler. The tool uses graph transformation as its underlying theory for aspect composition. Similar to Groher and Schulze's approach, MATA is not based on pervasive services. The authors' work [13] does not support explicit joinpoints as provided in the current research. Also, no code generation of the models has been provided by them.

4.4 Cottenier *et al.* Approach

Cottenier *et al.* [14] present Motorola WEAVR, which provides aspect-oriented weaving for UML state charts that include action semantics. Motorola WEAVR is an industrial-strength aspect weaver for UML 2.0, which is implemented as an add-in to the

Telelogic TAU G2. The tool essentially performs four main functions. First, the tool's profile allows engineers to define aspects in UML 2.0. Second, it presents a joinpoint visualization engine for visualizing and validating the effects of the aspects. Third, the tool provides full aspect weaving at the modeling level. Finally, the tool's simulation engine allows the simulation of the aspect models without breaking their modular structure. The authors have provided several UML stereotype classes for identifying various constructs of an aspect at the PIM level. Motorola WEAVR introduces two fundamental language constructs to support aspect-oriented modeling. First, the authors provide constructs to specify the locations or joinpoints in the models where crosscutting behavior emerges. Second, the authors provide constructs to specify the actual behavior of the crosscutting concerns using the connector stereotype. Weaving process consists of two phases: advice instantiation and advice instance binding. As weaving is performed at the PIM level with aspects woven into executable UML models, it allows PSMs and source code to be generated automatically. Several examples on exception handling and recovery have been developed by the authors to demonstrate the validity of their approach. However, the authors' work [14] is not in the pervasive computing domain as in the current research. Also, their PSMs are not at the formal behavioral specification level as in our study.

4.5   Fuentes *et al.* Approach

Fuentes *et al.* [15] present an aspect-oriented executable modeling UML 2.0 profile called AOEM for designing pervasive applications. Using the AOEM profile, they demonstrate how aspect-oriented executable design models for context-aware pervasive applications can be constructed and executed. These models are run and tested using *Populo*, which is an eclipse plug-in created by the authors for interpreting executable UML models. The AOEM profile aims at addressing two challenges in pervasive applications. They are the crosscutting nature of context-awareness, and the complexity associated with reasoning about the correctness of the design model. In their approach, weaving of aspects to core components is performed at the PIM level. Special composition rules expressed as pointcuts in the AOEM profile describe how aspects are applied to the core components. An aspect-oriented model weaver, which is a type of compiler or preprocessor, has been developed for weaving. The weaver essentially creates the design model by injecting the crosscutting behavior of the aspects into the core modules that the aspects crosscut. The authors illustrate their approach using a location-aware intelligent transportation system. Although Fuentes *et al.* [15] provide for modeling of adaptive behavior at the PIM level, they do not provide any model transformations to generate PSMs. Also, in general, their context models are at the context-aware application and middleware levels and not at the state machine level as in the current study.

Like the `Aspectual FSP Generation tool`, [12], [14] and [15] support an explicit joinpoint model of aspect-oriented modeling at PIM level. Also, all the compared approaches support PIM or PSM level weaving of aspects. The vertical evaluation has demonstrated that the `Aspectual FSP Generation tool` has unique features on context-dependent behavioral modeling and context-dependent behavioral code generation.

Table 1. Comparison matrix for vertical evaluation

Evaluation Criteria	Groher & Schulze	Whittle & Jayaraman	Cottenier *et al.*	Fuentes *et al.*	Aspectual FSP Generation tool
PIM level support for context-dependent behavioral modeling				▨	■
PIM level support for explicit joinpoint model of AOM	■		■	■	■
PIM or PSM level support for weaving	■	■	■	■	■
PIM and PSM level support for context-dependent behavioral code generation					■

Complete cover of a criterion	■
Partial cover of a criterion	▨
No cover of a criterion	

Table 1 shows that the `Aspectual FSP Generation tool` satisfies all the criteria as opposed to the other tools which satisfy only some criteria.

5   Horizontal Evaluation

This section describes the horizontal dimension of the framework [4]. In contrast to vertical evaluation discussed above, horizontal evaluation validates only particular features at a specific level of abstraction of the MDA stack. The horizontal evaluation view is designed to validate several desired key features that are mainly required at the PSM level of the aspectual pervasive software services specification. These evaluation criteria cover two aspects of the study. They are the formal methods and tools employed in the study, and the context and adaptation dimensions of the customization approach used in the services.

5.1   Formal Methods and Tools Used

In this subsection, the formal methods and tools used in the current study at the PSM level of abstraction are evaluated. Clarke *et al.* [16] provide several criteria that formal methods-based approaches and tools need to support. According to Clarke *et al.* [16], although some of these criteria are ideals, it is still considered good to aim for them. As provided in Section 2.1, the research methodology of the current study contains three stages: service specification, architecture definition and architecture modularization. In the present study, formal methods and tools have been applied during the service specification and architecture definition stages of the research methodology, and finally for model checking the aspectual pervasive software services specification. The *LTSA-MSC tool* has been used for specifying the software services of the service specification and generating a behavioral model in FSP. Using this generated behavioral model, a core service model was extracted. The formal model checker, the *LTSA*, has been used to verify the aspectual pervasive software service specification against specified system requirements. This subsection evaluates the application of the aforementioned formal

methods and tools in the current research against the criteria provided by Clarke *et al.* [16].

- *Early Payback*. Early payback is one of the key benefits of the current study. This study is focused on the architectural level of the software life-cycle. This architecture-centric approach builds models of pervasive software services and their compositions and verifies their behavior against specified system properties. Building architectural models of pervasive software services allows the software engineers to validate the actual correctness of the services before the services are implemented later in the software life-cycle. Thus, it provides early payback or feedback to the service engineer on the validity of the services.
- *Incremental Gain for Incremental Effort*. In the study, the PSMs of the aspectual pervasive software services specification have been derived following the three incremental stages of the research methodology: service specification, architecture definition and architecture modularization. Each of these stages has its own deliverables such as an message sequence chart specification for software services, architecture model for the software services in FSP, and a modularized architecture with aspect-oriented models to represent context-dependent behavior at the service interface level. This demonstrates that the service engineer can receive the benefits of the methodology in an incremental manner.
- *Multiple Use*. The pervasive services engineering methodology established in this research in general covers the requirements and architecture design stages of the software life-cycle. Therefore, the benefits of this methodology can be seen in multiple stages of the software life-cycle. This design methodology effectively facilitates the transition from requirements-oriented scenario descriptions of pervasive software services to architecture-centric behavioral models of aspectual pervasive software services.
- *Integrated Use*. The formal methods and tools used in this research are widely known in both academia and industry. First, this research has modeled the pervasive software services using message sequence charts provided by the LTSA-MSC tool. Message sequence charts are one of the most widely used sequence chart notations for describing system behavior. Second, the model checking tool employed in this study (LTSA tool) is widely used for behavior modeling and analysis and is well supported with documentation [17].
- *Ease of Use*. This research applies three automated tools in the pervasive services engineering process: the LTSA-MSC tool to generate the architecture model in FSP which is later used to extract the core service model, the `Aspectual FSP Generation tool` to generate context-dependent behavior in FSP, and the LTSA tool for simulating, animating and model checking pervasive services. The use of automated tools such as the LTSA makes the engineering process much easier to understand and adopt for the service engineer.
- *Efficiency*. One of the limitations of the current study is efficiency in terms of time and space. This is mainly attributed to the formal model checking technique used for verifying the pervasive services specification. One of the main challenges associated with model checking technique is the state space explosion problem. Nevertheless, by using action hiding and minimization features of the LTSA model

checker the present study has proven sufficiently efficient in modeling and verifying the case study subset of this research.

- *Ease of learning*. The graphical interfaces provided in the LTSA-MSC tool and the `Aspectual FSP Generation tool` and the automated nature of these tools effectively reduce the need to know formal FSP to start realizing the benefits of this research. The use of basic and high-level message sequence charts in the service specification stage of the methodology are widely used and understood in both academia and industry. Also, the sequence chart notion and semantics applied in the LTSA-MSC tool are restricted to basic features. It does not include complex constructs such as message queues, gates, parametric messages, or dynamic creation and termination of instances.

- *Orientation Toward Error Detection*. The approach presented in this paper is oriented towards detecting errors in the aspectual pervasive software services specification using the formal model checking technique. Aspects can introduce an additional correctness problem in software specifications because of their crosscutting effect and obliviousness nature. Therefore, tool support if possible automatic, is highly desirable to ensure the correctness of the specification. This research employs formal model checking to find errors concerning safety, progress, and fluent linear temporal logic assertions in the service specification, which can be hidden behind the individual components and aspects, or in the woven model. In the study, two techniques - counterexamples and witness executions - have been employed to point out any errors in the specification, which can be used by the service engineer to improve the state models or the system properties for the aspectual pervasive software services.

- *Focused Analysis*. This research is explored using a modified subset of a real-world case study called the ParcelCall project. The case study subset focuses on exception handling that needs to be enforced when a transport item's context information violates acceptable threshold values. The reference scenario used in the research describes an `awareness monitoring and notification pervasive service`, which alerts with regards to any exceptional situations that may arise on transport items primarily to the vehicle driver of the transport unit. This is an example where the research is focused on analyzing only a particular aspect of the system and not the entire system. Also, at the PSM level only temperature and pressure context properties have been modeled and verified. Similarly, other context properties such as item identity, location and acceleration can also be supported.

- *Evolutionary Development*. The incremental and iterative nature of the activities performed in each stage of the methodology essentially facilitates evolutionary system development. This can be demonstrated by the fact that the engineering process, which is initiated as a scenario-based specification expressed as message sequence charts, has evolved into a modularized service architecture where complex context-dependent information has been separated from the core service behavior as aspect-oriented models.

5.2 Context and Adaptation of the Customization Approach

This subsection evaluates the customization approach used in the pervasive services of the current study. This evaluation focuses mainly on the PSM level of abstraction. The authors in [18,19] present a comprehensive and uniform evaluation framework, which can be used to compare customization capabilities of approaches originating from the *mobile computing* and the *personalization* domains. The notion of customization refers to the adaptation of an application's services towards the current context. Their framework has two orthogonal dimensions, which are context and adaptation, and the mapping between context and adaptation represented by the notion of customization. The authors [18,19] provide detailed criteria for both the context and adaptation dimensions of the framework. Their evaluation framework aims at providing three main benefits. First, it provides a structured and uniform view of the various aspects of customization. Second, the framework can be effectively used as a conceptual framework for evaluating existing customization approaches. Finally, the framework can be effective for developing any future customization approaches.

Next the *context* and *adaptation* dimensions of the customization approach used in the pervasive services are evaluated using the criteria provided in [18,19]. The results of this evaluation are summarized in two tables respectively: Table 2 and Table 3. This evaluation is part of the horizontal evaluation dimension of the framework, and mainly covers the PSM level of the MDA stack. However, several examples on the PIM level are also provided. This evaluation is presented in three logical sections as suggested in [18,19]. First, general details of the customization approach are provided covering issues on *origin*, *major focus*, *technology*, *architecture* and *implementation* (if applicable) of the customization approach. Second, the context dimension of the approach is described, and third, the adaptation dimension of the approach is discussed.

The current research *originates* from the pervasive computing domain in general, and specifically from the pervasive services domain. This research is at the software architectural level, and no *implementation* of services such as pervasive Web services, has been considered in the study. The *main focus* of the `awareness monitoring and notification pervasive service` is to alert on any exceptional situations that may arise on transport items primarily to the vehicle driver of the transport unit. The component configuration of the *architecture* defined for the pervasive software services specification is based on an event-control-action architecture pattern. The research approach has been applied to a modified subset of a real-world case study in intelligent transport called the ParcelCall project. As stated previously in this paper, three automated *tools* have been used in the research: LTSA-MSC tool, `Aspectual FSP Generation tool`, and the LTSA model checker. The customization of the architecture can be considered internal as the system is not aware of the customization in terms of knowing about context or adaptation.

The *context dimension* of the customization approach is described next (Table 2). The primary context parameters modeled at the PIM level of abstraction comprise location, temperature and pressure. However, at the PSM level only temperature and pressure primary context properties have been modeled *(C.P.)*. Nevertheless, at the PSM level other context properties such as item identity, location and acceleration can be supported as well *(C.E.)*. These context parameters as a whole constitute the physical context of the

Table 2. Current study's context characteristics

Scope of Context										Representation of Context	Acquisition of Context				Access of Context	
	Property (C.P.)						Extensibility (C.E.)	Chronology (C.C.)	Validity (C.V.)	Reusability (C.R.)	Abstraction (C.Ab.)	Automation (C.Au.)			Dynamicity (C.D.)	Mechanism (C.M.)
location	temperature	pressure	time	device	network	user	application	history	future			manual	semi-automatic	automatic	static	dynamic	push	pull

Explicitly supported	■
Not explicitly supported	▨
Not applicable	☐

study. Context modeling at the PIM level is provided by the `c-FSP-UML profile` and the `c-FSP aspects`. The profile essentially provides several stereotypes to represent the core service behavior, the context-dependent behavior, and the dependencies between the core service behavior and the context-dependent behavior at the service interface level. The use of stereotypes essentially supports the notion of extensibility *(C.E.)*. The object-oriented notions used in the profile such as generalization further support the notion of extensibility. At the PSM level, the notions of attribution, classification, generalization and encapsulation from the context definition have been modeled to structure and link the objects defined in the aspects *(C.E.)*. In the study, validity period *(C.V.)*, chronology *(C.C.)* or availability *(C.Av)* of context are not supported as the time dimension of the context properties have not been considered. The explicit support provided for attribution, instance-of and ISA notions at the PSM level, facilitates reusability of context *(C.R.)*. The approach provides a high-level inference mechanism to automatically derive higher-level logical context *(C.Ab)*. Both primary and logical context are modularized into aspects as atomic context aspects and composite context aspects respectively. Thus, the study supports an explicit separation between physical and logical context *(C.Ab)*. For example, low-level temperature readings from the `RFID tags` are inferred as low temperature or high temperature during the refinement step of the pervasive service. At the PIM level, the profile is maintained manually by the service engineer *(C.Au.)*. At the PSM level, both physical and logical context information are acquired automatically *(C.Au.)* and at run-time *(C.D.)*. The pervasive service engineer using the LTSA animator can select values for the temperature or pressure readings from a range of values at run-time. The mechanism *(C.M.)* used to acquire context and made accessible to the pervasive service can be considered push-based as context readings from the `RFID tags` are provided based on context changes and not on requests.

Table 3. Current study's adaptation characteristics

Kind of Adaptation				Subject of Adaptation			Process of Adaptation			
Operation (A.O.)	Extensibility (A.Ex.)	Effect (A.Ef.)	Complexity (A.C.)	Level (A.L.)	Element (A.El.)	Granularity (A.G.)	Tasks (A.T.)	Automation (A.A.)	Dynamicity (A.D.)	Incrementality (A.I.)
		add / remove / transform	simple / complex	content / hyperbase / presentation / others	text / audio / image / video / link / others	micro / macro		automatic / semi-automatic / manual	static / dynamic	

Explicitly supported	■
Not explicitly supported	▨
Not applicable	□

The second dimension of the customization approach is provided by the notion of *adaptation* (Table 3). In this study, there are two types of adaptation operations: triggering and recovery operations *(A.O.)*. At the PSM level, both these operations have been supported by the `Trigger` and `Recovery` c-FSP aspects *(A.O.)*. `Trigger` aspects, for example `Trigger Aspect Adverse Environment Status`, effectively send notifications or alert messages to the vehicle driver when a transport item's context information violates acceptable levels. `Recovery aspects`, for example `Recovery Aspect Adverse Environment Status`, model any recovery actions that need to be enforced after an exception situation is raised by a `Trigger aspect`. Both these aspects have been modeled as independent state machines at PSM level, which synchronize with their base state machines using explicit synchronization events. The adaptation operations provided by the aspects are associated with the core service model through weaving, and the behavior of these aspects can affect the core service behavior depending on the context information *(A.Ef.)*. This can be, for example, executing or modifying a service based on context information *(A.Ef.)*. At the PIM level, adaptation operations are specified using the stereotypes: `TriggerAspect` and `RecoveryAspect`. As the adaptation process contains a series of stages, it can be considered a complex process *(A.C.)*. Also, a distinct separation can be identified between the different tasks of the adaptation process *(A.T.)*. When an item's context information is violated, first, the pervasive service alerts the vehicle driver by sending an SMS. Second, the service can perform any appropriate recovery actions to remedy the situation, such as control the refrigerator's temperature *(A.C.)*. Third, the service adaptation can be extended as follows *(A.Ex.)*. If the environment status of items is critical the service can alert the `goods tracing server` and eventually the `customer` being affected through the `goods information server`

(A.Ex.). In the current study, context information is represented at the service inter-face level *(A.L.)* that essentially consists of operation invocations and the exchange of respective input/output parameters. The core service elements represented at the mod-eling level include states, transitions, processes, and services *(A.El.)*. Any adaptation operation, which can be a triggering operation or a recovery operation, is bound to the transitions of the core service model. Therefore, the adaptation level and adaptation elements of this study are the service interface level *(A.L.)* and transitions *(A.El.)* re-spectively. As a result, different Web application levels such as content, hyperbase and presentation are not applicable in this study nor the Web adaptation elements of text, audio, image, video and link provided in [18,19]. The adaptation granularity *(A.G.)* can be considered micro considering the number of elements affected by the adaptation process in the study. Also, it is performed automatically *(A.A.)* and at run-time *(A.D.)*. The adaptation process can be considered incremental *(A.I.)* as recovery operations are dependent on triggering operations at both PIM and PSM levels.

6   Conclusions

To summarize this paper has discussed the *evaluation framework* developed to validate the main methods and tools employed in this study for engineering pervasive software services. The method of evaluation is based on key features comparison. The evaluation framework consists of two dimensions or views: vertical and horizontal. The vertical evaluation of the research compared several research tools to the `Aspectual FSP Generation tool` developed here. The tools were compared across the PIM and PSM levels of the MDA stack. This evaluation was based on several criteria: context-dependent behavioral modeling at the PIM level, explicit joinpoint model of AOM at the PIM level, weaving performed at PIM or PSM levels, and context-dependent behavioral code generation from the PIM level to the PSM level. The horizontal evaluation view was designed to validate several desired key features required mainly at the PSM level (i.e. FSP) of the aspectual pervasive software services specification. These evaluation criteria mainly cover two aspects. They are the formal methods and tools employed in the study and the context and adaptation dimensions of the customization approach used in the pervasive services. The results of the evaluation are assuring. The vertical evalu-ation has demonstrated that the `Aspectual FSP Generation tool` has unique features in context-dependent behavioral modeling and context-dependent behavioral code generation. The horizontal evaluation of the approach has shown that the formal methods and tools employed in the research, and the customization approach used in the services are indeed effective towards the overall objectives of this research.

References

1. Hegering, H.-G., Küpper, A., Linnhoff-Popien, C., Reiser, H.: Management Challenges of Context-Aware Services in Ubiquitous Environments. In: Brunner, M., Keller, A. (eds.) DSOM 2003. LNCS, vol. 2867, pp. 246–259. Springer, Heidelberg (2003)

2. Mandato, D., Kovacs, E., Hohl, F., Amir-Alikhani, H.: CAMP: a Context-Aware Mobile Portal. IEEE Communications Magazine 40(1), 90–97 (2002)
3. Mostefaoui, S.K., Hirsbrunner, B.: Context-Aware Service Provisioning. In: IEEE/ACS International Conference on Pervasive Services (ICPS 2004), pp. 71–80. IEEE (2004)
4. Abeywickrama, D.B.: Pervasive Services Engineering for SOAs. Ph.D Thesis, Faculty of IT, Clayton Campus, Monash University, Australia (2010)
5. Abeywickrama, D.B., Ramakrishnan, S.: Model-Driven Development of Aspectual Pervasive Software Services. In: 14th IEEE International Enterprise Distributed Object Computing Conference Workshops, pp. 49–59. IEEE, Vitoria (2010)
6. Abeywickrama, D.B., Ramakrishnan, S.: Towards Engineering Models of Aspectual Pervasive Software Services. In: 3rd Workshop on Software Engineering for Pervasive Services (SEPS 2008), pp. 3–8. ACM, Sorrento (2008)
7. Abeywickrama, D.B., Ramakrishnan, S.: A Model-Based Approach for Engineering Pervasive Services in SOAs. In: 5th International Conference on Pervasive Services (ICPS 2008), pp. 57–60. ACM, Sorrento (2008)
8. Aspect-Oriented Modeling, http://www.aspect-modeling.org/ (last accessed on July 20, 2011)
9. Davie, A.: Intelligent Tagging for Transport and Logistics: The ParcelCall Approach. Electronics & Communication Engineering Journal 14(3), 122–128 (2002)
10. Analyti, A., Theodorakis, M., Spyratos, N., Constantopoulos, P.: Contextualization as an Independent Abstraction Mechanism for Conceptual Modeling. Information Systems Journal 32(1), 24–60 (2007)
11. VIsualize all moDel drivEn programming (VIDE), WP 11: Deliverable number D11.3, Supported by the European Commission within Sixth Framework Programme. Polish-Japanese Institute of Information Technology, http://www.vide-ist.eu/download/VIDE_D11.3.pdf (last accessed on July 20, 2011)
12. Groher, I., Schulze, S.: Generating Aspect Code from UML Models. In: 3rd International Workshop on Aspect-Oriented Modeling Co-located with 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), Boston, USA (2003)
13. Whittle, J., Jayaraman, P.: MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 16–27. Springer, Heidelberg (2008)
14. Cottenier, T., van den Berg, A., Elrad, T.: Motorola WEAVR: Aspect Orientation and Model-Driven Engineering. Journal of Object Technology 6(7), 51–88 (2007)
15. Fuentes, L., Gamez, N., Sanchez, P.: Aspect-Oriented Executable UML Models for Context-Aware Pervasive Applications. In: 2008 5th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, pp. 34–43. IEEE, Budapest (2008)
16. Clarke, E.M., Wing, J.M., Alur, R.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys 28(4), 626–643 (1996)
17. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs, 2nd edn. John Wiley and Sons (2006)
18. Kappel, G., Pröll, B., Retschitzegger, W., Schwinger, W.: Customisation for Ubiquitous Web Applications: A Comparison of Approaches. International Journal of Web Engineering and Technology 1(1), 79–111 (2003)
19. Schwinger, W., Grün, C., Pröll, B., Retschitzegger, W., Schauerhuber, A.: Context-Awareness in Mobile Tourism Guides - A Comprehensive Survey. Technical report, Johannes Kepler University, Linz, Austria (2005)

ABC Architecture: A New Approach to Build Reusable and Adaptable Business Tier Components Based on Static Business Interfaces

Oscar M. Pereira[1], Rui L. Aguiar[1], and Maribel Yasmina Santos[2]

[1] Instituto de Telecomunicações, University of Aveiro, 3810-193 Aveiro, Portugal
{omp,ruilaa}@ua.pt
[2] Centro Algoritmi, University of Minho, 4800-058 Guimarães, Portugal
maribel@dsi.uminho.pt

Abstract. Currently, programmers of database applications use standard API and frameworks as artifacts to develop business tiers aimed at integrating the object-oriented and the relational paradigms. These artifacts provide programmers with the necessary services to develop business tiers. In this paper we propose a new architecture based on general Call Level Interfaces from which reusable and Adaptable Business tier Components (ABC) may be developed. Each individual ABC component is able to manage SQL statements of any complexity, deployed at run-time, and also to provide tailored services to each SQL statement. To accomplish this goal, the only requirement is that the schema of each deployed SQL statement must be in conformance with one of the pre-defined static schemas (interfaces) of the recipient ABC component. The main contributions of this paper are threefold: 1) to present the new architecture based on general Call Level Interfaces on which ABC components are based, 2) to show that the source code of ABC components may be automatically built by a tool and 3) to present a concrete example of ABC based on JDBC. The main outcome of this paper is the evidence that the presented architecture is an effective approach to build reusable and adaptable business tiers components to bridge the object-oriented and the relational paradigms.

Keywords: Component-based software, Adaptability, Business tiers, Impedance mismatch.

1 Introduction

Good programming practices advise the development of database applications relying on a multi-tier architecture. The three tier architecture is the most widespread one comprising the application tier, the database tier and the middle tier known as the business tier. The business tier may provide a clear separation (technological, business and administrative/administration) between host databases and client applications. When database tiers and application tiers rely on different paradigms, as the relational and object-oriented, respectively, business tiers are responsible for relieving programmers of client applications from several critical issues being

impedance mismatch [1] the most noticeable one. Impedance mismatch is an outcome of the diverse foundation of both paradigms raising a major hindrance for their integration, being an open issue for more than 50 years [2]. Despite their advantages, business tiers present some weaknesses. Among them we emphasize their inertia to evolve in consequence of maintenance needs. These needs may have their origin in the need for new queries, the need to update existent queries or changes in the database schema. Inertia may reach increased relevancy if SQL statements are wrapped into classes with improved usability to ease their usage by programmers of client applications. In this case, maintenance activities will not only comprise lower-level issues as writing or re-writing the SQL queries but will also comprise the development or maintenance of the involved wrappers to keep their usability (examples: getter and setter methods). The difficulties to build and maintain business tier components may have a shelter on the Component-Based Software Engineering (CBSE) [3] subject. CBSE is widely recognized as a sub-discipline of Software Engineering to build complex systems. The main goals of CBSE are threefold: 1) to provide guidelines for the development of systems as assemblies of components; 2) to provide guidelines for the development of components as reusable artifacts and finally 3) to provide guidelines for the maintenance of systems through the adaption and replacement of their constituent components. Using commercial off-the-shelf (COTS) software components to build software systems may be seen as a goal for many system architects. Unfortunately, components reutilization may raise several technological difficulties and, not less important, may easily gather voices against its adoption. In fact, despite the relevancy of the CBSE postulates, several issues are difficult to tackle such as the replacement and adaptation of components. Component replacement has some disadvantages conveying an impact on the overall system. Some of the disadvantages are [4]: 1) the state of the replaced component may be lost; 2) component or even system availability may be affected; 3) performance may decay during the replacement process – additional power computation is required. In order to avoid the replacement of components, components must be able to adapt dynamically at run-time, which is one of the crucial aspects of CBSE [5]. The adaptation of components should comprise not only the configuration process but mainly the replacement of old services and also the definition of new services in a seamlessly way.

In this paper we are focused on an architecture for reusable business tier components where client applications are developed in the object-oriented paradigm and the host database relies on the relational paradigm. The components based on the presented architecture are herein known as Adaptable Business Components (ABC). The ABC architecture pretends to achieve the following three main goals: to comply with full expressiveness of SQL, to provide an enhanced usability from client applications point of view and to provide supervised adaptability to SQL statements deployed in run-time. Full SQL expressiveness is achieved by using Call Level Interfaces (CLI) [6] as a low-level API to communicate with the host database. Call Level Interfaces will be addressed with some detail in Section 4. Usability is assured by the implemented interfaces to communicate with client applications. These interfaces are based on the schema of the SQL statements and are also type-safe. This issue will be addressed with more detail in Section 5. Supervised adaptability is assured by the ability to dynamically, in run-time, as a server component, receive

messages from *"authorized"* entities to accept new, remove existent and update existent SQL statements of any complexity and, on behalf of client applications, manage their execution. The results of their execution are at the disposal of client applications through the aforementioned interfaces. This issue will be addressed with more detail in Section 5 and Section 6. Fig. 1 presents a general view of the interaction between ABC components and client applications (CA). Authorized entities may create and update a pool of SQL statements in run-time and delegate their management to the ABC component. Then client applications may ask the execution of SQL statements through the interfaces provided by ABC components.
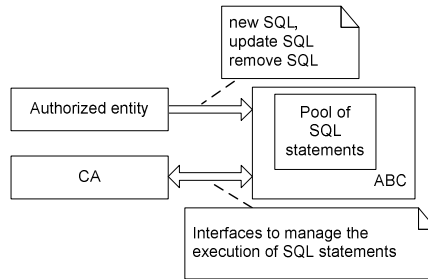
Fig. 1. General view of CA and ABC interactions

It is expected that the outcome of this paper will contribute to open a new approach to the development of business tier components. The paper is structured as follows. Section 2 presents the motivation; Section 3 presents the related work; Section 4 concisely presents the Call Level Interfaces; Section 5 presents the ABC architecture; Section 6 presents the ABC Life-Cycle and Section 7 present the final conclusion and future work.

2   Motivation

Database applications of some complexity may comprise hundreds of SQL statements to deal with business requirements. Very often they cannot be inferred from any data model that may eventually be available (ex: O/RM). This leads to situations where the development and maintenance processes of business tiers are very tedious and exhaustive. Programmers are pushed to write similar source code for each SQL statement, mainly for Select statements with a long select-list. There should exist a methodology to relieve programmers from these tedious, exhaustive and error-prone processes.

SQL statements may be classified into two orthogonal dimensions: by complexity and by schema. Complexity says if a SQL statement is simple or complex. The schema characterizes each SQL statement in terms of the schema of its parameters and the schema of the returned relation (only for Select statements). A SQL statement may be simple and have a simple or a complex schema or a SQL statement may be complex and have a simple or a complex schema. Moreover, several SQL statements, simple or complex, may share the same schema. The two following queries share a

simple schema. The first query is very simple and the second is not so simple. The code to execute the query and to read the returned data is the same for both queries. This evidence raises the following question: if several SQL statements may share the same schema why not make use of it to optimize the source code editing and maintenance processes? The first question to be put is: "*Have we been spreading boilerplate code in business tiers?*". Another relevant issue is the access

```
-- a simple query
Select p.id,p.fName,p.lName
  From pilot p

-- a more complex query
select p.id,p.fName,p.lName
  From pilot p,circuit c,classif f
  Where p.id=f.id and f.date=c.date and f.position between 1 and 3
  Group by p.id,p.fName,p.lName
  Having count(f.position)= (select count(*) from ...)
Union
Select ...
   ...
Order by ...
```

and manipulation of data kept in databases. While the issue previously discussed was essentially technological, the access and manipulation of data concerns the soul of companies. Very often data are the most important asset in companies conveying an unavoidable need to completely control and protect them.

In this paper we present an architecture for ABC components aimed at coping with two important features: 1) Re-use of source code to manage different SQL statements, simple or complex, defined after ABC deployment; 2) To follow the separation of concerns regarding the use of ABC components by programmers of client applications from all other issues related to the development, configuration and administration processes.

3   Related Work

In [7] it is presented a model-typed interfaces concept relying on generic interface parameters. These parameters are characterized as Model-defined Types whose schema is defined by a Data Model. The authors claim that by this way complex data structures (based on Data Models) may be transferred between components in a single method invocation avoiding successive calls to accomplish the same task. This methodology is very useful when two conditions occur simultaneously: 1) the involved components do not share the same working address space and 2) the component playing the client role has full control and knowledge about the amount of data being transferred. In the work proposed is this paper, ABC components share the same address space as client applications and the access to the returned data (from Select statements) is to be implemented in an attribute by attribute and a row by row basis. This work could profit from [7] if or when ABC components and client application run in different address spaces.

Data Transfer Objects [8] is a design pattern used whenever an entity gathers a group of attributes that must be accessed in a swift way. Accessing those attributes

one by one through a remote interface raises several disadvantages such as the increase of the network traffic, latency is increased, performance is negatively affected, demand on server and client processing is increased. Data Transfer Objects are tailored to address these situations. They are organized in serializable classes gathering the related attributes and forming a composite value. An entire instance of the serialized object is transferred from the server to the client. This approach is quite similar to the previous conveying the same disadvantages.

O/RM tools (Hibernate [9], TopLink [10], LINQ [11]) are powerful tools to integrate object-oriented applications with relational databases. Their extended functionalities are mostly used to build persistent business tiers relying on object to relational mapping techniques. They may also support native queries, proprietary SQL language, language extensions and other relevant tools to ease programmers' work. Their scope is too wide and deeply diverges from the scope of this work. Anyway, if required, ABC components may be developed with and above O/RM frameworks. Their services have to be coordinated, integrated and wrapped by the architecture herein presented.

4   Call Level Interfaces

One of the key requirements of ABC component is the ability to execute SQL statements of any complexity. Before this requirement, the option for an API to access the host databases is a key issue. The choice felt upon low-level APIs being Call Level Interfaces an important candidate. Call Level Interfaces are considered important options whenever performance and SQL expressiveness are considered key issues [2]. Call Level Interfaces provide mechanisms to encode Select, Insert, Update and Delete SQL expressions inside strings, easily incorporating the power and the full expressiveness of SQL. JDBC [12] and ODBC [13] are two of the most relevant Call Level Interfaces. Statements are executed against the host database and the possible results they produce (only for Select statements) are locally managed by local memory structures (LMS) – (ResultSet for JDBC and RecordSet for ODBC). LMS provide two orthogonal functionalities: *scrollability* and *updatability*. Scrollability defines the ability to scroll over the LMS. There are two possibilities: *forward-only* – in this case cursors may only move forward one row at a time; *scrollable* – in this case cursors may move in any direction and jump several rows at a time. Updatability defines the capacity to change the in-memory data of LMS and therefore the content of the host database. There are two possibilities: *read-only* – the content of the LMS is read-only and, no changes are allowed; *updatable* – changes may be performed over the in-memory data of LMS (insert new rows, update current rows and delete rows).

5   ABC Architecture

The main goal of this paper is to present an architecture for general a ABC, with the ability to manage and execute a pool of SQL statements on behalf of client applications. The pool of SQL statements is dynamically updated in runtime by an external authorized entity. From client applications point of view, ABC components

always play the role of server components. From the host database point of view, ABC components always play the role of client components. The static architecture of ABC components comprises three main blocks: Business Manager (BM), Business Entities (BE) and the Database Driver (DB Driver), see Fig. 2.
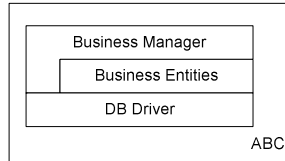
Fig. 2. ABC main blocks

DB Drivers are standard API and they are responsible for providing internal services to ABC components in order to ease their communication with the host DBMS. The choice for the specific DB Driver depends on several issues, as the host DBMS and the host programming language of the client application.

Business Entities (BE) are software artifacts (classes) responsible for the implementation of contracts (interface) between ABC components and client applications. Each Business Entity implements one contract which is specified by an interface known as Business Interface (BI). The correspondent dynamic artifacts are the Business Workers (BW). Fig. 3 presents the basic relation between BI, BE and BW. BW are active entities, in other words, are running instances of Business Entities. A Business Worker accepts any SQL statement, at run-time, whose schema is in accordance to the Business Interface implemented by its source Business Entity. Thus, Business Entities address the key issue of *reuse* of computation [14]. SQL statements are defined through the *setSQL* allowing that the definition and updating processes be carried out after Business Workers instantiation.
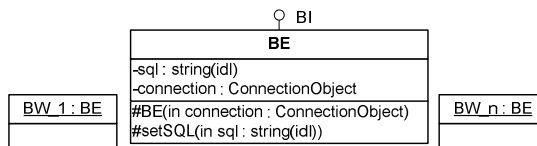
Fig. 3. Relationships between BI, BE and BW

These processes are managed by the Business Manager in a transparent way for client applications. Business Interface is a contract that a Business Entity is committed to implement. Business Interfaces define how client applications and Business Entities may communicate. The schema of a Business Interface is directly dependent on the queries to be processed and, in case of Select statements, on the functionalities to be made available (scrollability and updatability). In order to comprise all LMS functionalities, three types of Business Interfaces were defined: 1) Alter Business Interface (A-BI) – for Insert, Update and Delete SQL statements; 2) Select Active Business Interface (SA-BI) – for Select SQL statements that create

updatable LMS and 3) Select Passive Business Interface (SP-BI) – for Select SQL statements that create read-only LMS. Select Active Business Interfaces and Select Passive Business Interfaces implement one of the two scrollability facets: forward-only or scrollable. Before delving into the Business Interfaces details, let's consider a table with the following schema:

```
Table(SqlDT1 att_1,
      SqlDT2 att_2,
      ...,
      SqlDTn att_n)
```

where SqlDTn is the SQL data type of the attribute *att_n*. The correspondent data type of SqlDTn in the host programming language is represented by DTn. This table will be used as the basis for Business Interfaces specification. Each Business Interface may be considered as aggregations of super-interfaces. Therefore, we will begin the description of Business Interfaces by their elementary super-interfaces, see Fig. 4.

«interface» **ISet**	«interface» **IResult**	«interface» **IForwardOnly**		
+sAtt_1(in value : DT1) +...() +sAtt_n(in value : DTn) +beginUpdate() +endUpdate() +beginInsert() +endInsert() +delete()	+getResult(in args) : unsigned long(idl)	+moveNext() : boolean(idl)		
	«interface» **IScrollable**		«interface» **IGet**	«interface» **IExecute**
	+moveNext() : boolean(idl) +moveAbsolute(in pos : long(idl)) : boolean(idl) +moveFirst() +moveLast()		+gAtt_1() : DT1 +...() +gAtt_n() : DTn +gAtt_2() : DT2	+execute(in args)

Fig. 4. Super-interfaces of BI

Super-interface *IExecute* is shared by all Business Interfaces. It comprises only one method. This method is invoked by client applications to trigger the execution of the associated SQL statement. The method *execute* may be invoked as often as necessary to re-execute the SQL statement. The argument *args* comprises all the arguments to be used in conditions inside the SQL statement and also as values to be inserted or updated on tables of the host database. As an example, the next SQL statement leads to the next method signature.

```
// SQL statement          // method signature
Update Table              void execute(DT1 v_1,
  Set (att_1=@v_1,                     DTn v_2,
       att_n=@v_2)                     DT2 v_3)
  Where (att_2=@v_3);
```

Super-interface *IResult* comprises a single method to return the execution result of Insert, Update and Delete statements. Basically, its returns the number of rows affected by the statement execution.

Super-interface *IGet* gathers all necessary methods to read all attributes of one row from the in-memory data of LMS. The *IGet* interface shown in Fig. 4 may be used with the following query: *Select * from Table*. The method signatures are based on the schema of the Select statement and are also type-safe. These features improve ABC component usability when compared with the standard CLI API. Users of ABC

components are before signatures type-safe and schema oriented easing both the understanding of their meaning and the associated data-type.

Super-interface *ISet* gathers all necessary methods to update, insert and delete data in the LMS. It is only used with updatable LMS. The *ISet* interface shown in Fig. 4 may be used with the statement *Select * from Table*. The signatures of setter methods are based on the schema of the Select statement and are also type-safe as happened with the IGet interface. The remaining methods are used to implement the protocols to update, to insert and to delete rows.

Super-interface *IFowardOnly* comprises all methods associated to the scrolling policy of forward-only LMS. Fig. 4 only presents the main method which allows the cursor to move one row forward at a time.

Super-interface *IScrollable* gathers all methods associated to the scrolling policy of scrollable LMS. Fig. 4 presents only four of the main methods.

All super-interfaces of Business Interfaces have been individually presented. Fig. 5 presents the three general Business Interfaces: A-BI, SP-BI and SA-BI. A-BI comprises two interfaces (IExecute, IResult), SP-BI comprises three interfaces (IExecute, IGet and, depending on the LMS functionality, IForwardOnly or IScrollable). SA-BI comprises four interfaces (IExecute, IGet, ISet and, depending on the LMS functionality, IForward or IScrollable).
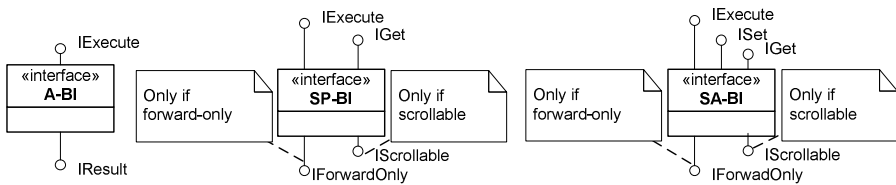

Fig. 5. A-BI, SP-BI and SA-BI sub-interfaces

Business Workers are running instances of Business Entities. Each Business Worker is identified by its type (parent Business Entity) and the SQL statement to be executed. Each SQL statement is uniquely identified by a token. There cannot exist two tokens with the same value in the same ABC component instance. Business Workers instantiated from the same Business Entity are called sibling Business Workers. Business Workers running the same SQL statement are called true sibling Business Workers and Business Workers running different SQL statements are called false sibling Business Workers. Examples of two SQL statements managed by the same Business Entity and, therefore, running on false sibling Business Workers could be:

```
Select *                Select t1.*
  From Table               From Table t1, Table t2
  Where att_1=@v_1         Where t1.att_1=@v_1 and
                                t1.att_2=t2.att_2 and
                                t2.att_3=1
```

Despite some restrictions, each Business Entity may support an unlimited set of SQL statements. The restrictions are only centered on the implemented Business Interface. Particularly the interface IExecute, *execute(args)*, may eventually convey a significant weakness on the Business Entity adaptability. This weakness is felt at the level of the

SQL statements *where* and *having* clauses. Anyway, if required, this weakness may be avoided by using the following signature *void execute()*. From now on, SQL statement parameters, if required, must be set by the client application as shown in the next example. This strategy may improve Business Entities' flexibility but forbids the use of parameterized queries as pre-parsed queries (prepared statements) this way provoking decrease in performance. Another important drawback of this approach is the decrease of ABC usability: IExecute may no longer be used to help programmers on setting up the parameters of SQL statements.

```
Select *
  From User
  Where Grade>10 and Grade<16 and
        Substring(FName,1,1) like 'A'
```

Business Manager is the entry point of all ABC components. ABC's administrators and programmers of client applications access ABC components functionalities through their entry static methods. Business Manager has a static method for each Business Entity that creates a singleton instance of its factory class, as shown in Fig. 6. Business Entities factories implement two interfaces: one is only at administrator's disposal to update the pool of SQL statements and is known as IAdm; the other interface is at common programmers' disposal to create Business Workers and is
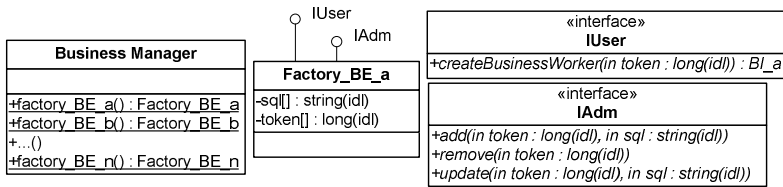
Fig. 6. Class diagrams of: business manager, factories and, IAdm and IUser

known as IUser. Each SQL statement to be used by any Business Worker is uniquely identified by a token and is defined by administrators in the *add* method. The class diagrams for IAdm and IUser interfaces are also shown in Fig. 6. The next block of code depicts examples of source code to add SQL statements to the pool and source code to create and use a Business Worker.

```
// add SQL statement          // create Business Worker
IAdm a=Manager.factory_BE_a();  IUser u=Manager.factory_BE_a();
a.add(tk_a1,sql_a1);            BI_ai a=u.createBusinessWorker(tk_a1);
a.add(tk_a2,sql_a2);
```

6   ABC Life Cycle

ABC components comprise two types of software sources: outsource (software from other suppliers – DB Driver) and insource (software specifically developed to ABC components – Business Manager, Business Entities and Business Interfaces). The catalog of ABC components is defined within the context of *idealized component life cycle* [15]. The life cycle is based on the *development for reuse* and *development*

with re-use processes in agreement with CBSE principles and considers three phases: *design*, *deployment* and *runtime*.

6.1   Design Phase

The design phase is focused on the development of ABC components. Developers of ABC components may follow three distinct approaches: global approach, the activity approach or the entity approach. The entity approach is based on the development of ABC components with only one business entity. The activity approach is based on the development of ABC components by each activity such as accounting, clients, suppliers, data warehouse and OLAP. The global approach is based on the development of a single ABC component for all activities. It is also possible to follow any combination of the three approaches. The decision is up to the system administrator. Regardless the chosen approach, each ABC component may be accessed by several actors where each one plays a specific role conditioned by the queries he may execute. Different instances of the same ABC component may run a different set of SQL statements for each Business Entity this way promoting its reuse by different actors. Moreover, every software subsystem (SS), such as warehouse management, gathers several activities such as clients, suppliers and orders this way promoting the component reutilization. There is a wide range of possibilities for component reuse: by activity, by actor or any by other combination. Nevertheless, Business Interfaces cannot be modified, added or removed after the design phase. They materialize the contract between each individual ABC component and the final client applications. Client applications trust ABC components to manage SQL statements since each SQL statement is in conformance with one of the implemented Business Interfaces. Any change in the contract after the design phase of ABC components compels the re-opening of the design phase. After the design phase SQL statements may be created as needed (this is the degree of freedom they have) but each SQL statement must be in conformance with one of the available Business Interfaces (this is the restriction they must obey). The insource code for each ABC component may be automatically built by a tool as the one shown in Fig. 7. Only the GUI used to create Select Business Interfaces is shown. We may see 3 Business Interfaces in the pool (BI_Course, BI_Subject, BI_Degree) and a new one is being edited (BI_Student) and ready to be inserted in the pool. This Business Interface supports, for example, the next SQL statement: *Select id, firstName, lastName,crdId, grade from Student where id=@id*. Other additional features defined at the design phase for Select Business Interfaces are its scrollability and its updatability. Alternatively, the insource source may be derived from a general model herein known as the Business Component Model (BCM). In order to automatically generate the insource code the Business Component Model requires the following information:

▪ For each Business Interface: 1) Its type: alter, select passive or select active; 2) the Business Interface schema; 3) the scrollability policy.
▪ The source code programming language.
▪ The host database management system.
▪ The DB Driver to be used.

These two approaches, a tool or the BCM, relieve programmers from writing any source code and therefore to avoid the deployment of ABC components with

undesirable errors. The final stage of the design phase is attained when ABC components are compiled and packed as components ready to be deployed.

As a summary, the design phase embodies the process of development for reuse and is focused on the definition of Business Interfaces to be implemented by reusable ABC components. The concrete SQL statements to be deployed in each instance of an ABC component are not defined in the design phase but in a later phase.
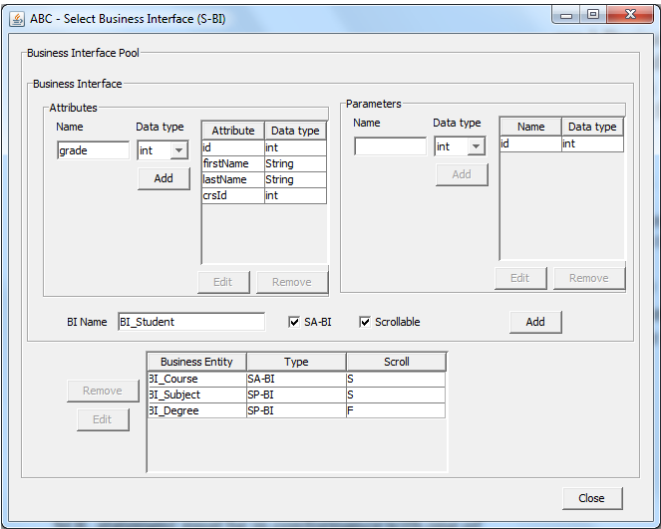
Fig. 7. Widget to create S-BI

6.2   Deployment Phase

In the deployment phase, developers use ABC binary components to develop each subsystem of their database applications. Developers may play the role of an administrator developer or the role of a common developer. The former role is used to write source code to update the pool of SQL statements to be made available. The latter role is used to write source code for subsystems. Database applications may incorporate one or more subsystems, each subsystem may incorporate one or more activities and each activity may support one or more actors. System administrators must define the strategy for the ABC components reutilization in the deployment phase. Common developers may not know anything about the SQL statements made available in each ABC component. All they need to know is the location of the required SQL statement in terms of ABC component instance, the Business Entity and its token. As summary, the deployment phase embodies the process of *development with reuse*.

6.3   Runtime Phase

In this phase all components are running. Fig. 8 concisely presents a possible running scenario with two subsystems, SS_a and SS_b. Fig. 8 a) shows two instances of SS_a (SS_a1, SS_a2). Both share the same ABC components, ABC_y and ABC_w.

Eventually, if SS_a1 and SS_a2 correspond to two different actors, the sets of SQL statements made available in each component may be different. Fig. 8 b) shows one instance of SS_b (SS_b1) which is a different component from SS_a. It comprises three ABC components, ABC_y ABC_w, ABC_z, two of them shared with SS_a (ABC_y and ABC_w), probably with their own sets of SQL statements. Remenber that SQL statements are deployed at run-time through the IAdm interface which is not shown in this figure. The administrator role may or may not be protected by some security policy such as authentication and/or authorization to grant access to the configuration process. This topic is out of scope of this work.
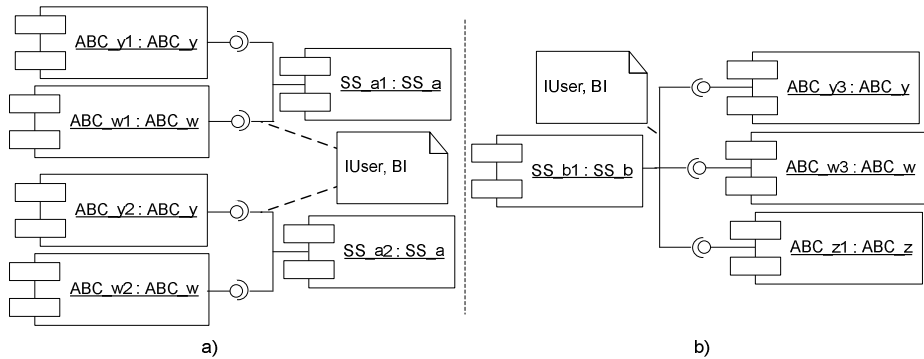
Fig. 8. Client application and ABC deployment

6.4   Seamless Operation

SQL statements updating process is executed in a seamless way. This means that the process to insert, update or delete SQL statements from the pool of ABC components may be executed without any restrictions. Actually, ABC components assume a passive attitude. They do not provide any service to inform client applications from any relevant or critical occurrences. These occurrences should be coordinated between client applications and the component that plays the administrator role. Inserting new SQL statements does not raise any critical question. Subsystems are not allowed to use what still does not exist. The identification token should only be made available after inserting the SQL statement into the pool. Updating and removing SQL statements that are being used by one or more Business Workers are the critical situations. When a SQL statement is being used and it is updated or removed, Business Workers keep their states unchanged. This assures that client applications may continue their work. Business Workers' state will only be updated when client applications re-invoke the *execute* method. Then, Business Workers will re-execute the most recent SQL statement. The *execute()* method should not be invoked if the SQL statement has been removed from the pool. To prevent any undesirable situation, it is advisable that client applications become aware of the actions taken by the administrator in order to proceed with the most convenient measures. Business Manager does not interfere or change the state of any Business Worker. The state of

Business Workers are always under sub-systems' control. Therefore, each application should define its own protocol between administrator and application components.

6.5   ABC Example Based on JDBC API

This section presents a concrete example of an ABC component based on JDBC API. The example is based on the Business Interface BI_Student presented in Section 6.1, which is type SA-BI. The following scenario has been implemented: 1) the administrator adds two queries; 2) a user creates one Business Worker using one of the queries, iterates over the LMS and, under certain condition, updates and insert some rows into the LMS. The example is based on Java, SQL Server 2008 and JDBC for SQL Server (sqljdbc4.jar). Code may not execute properly since we only show the relevant parts for the points under discussion.

Fig. 9 presents a possible use of ABC from administrators' point of view: two queries are defined, an instance of the Business Entity factory is created and, finally, the two queries are added to Business Entity BE_Student. From now on, users may instantiate Business Workers from BE_Student and choose the desired queries to be executed.

```
// definition of queries
String sql1="select id,firstName,lastName,crsId,grade " +
            "from Student where id=?";
String sql2 ="select id,firstName,lastName,crsId,grade " +
            "from Student where id>?";
// more queries
IAdm admStd = BusinessManager.factory_BE_Student();
admStd.add(token1, sql1);
admStd.add(token2, sql2);
// add more queries
```

Fig. 9. Usage of ABC from administrators' point of view

```
IUser userStd = BusinessManager.factory_BE_Student();
BI_Student std2 = userStd.craeteBusinessWorker(token2);
std2.execute( 10 );
while ( std2.moveNext() ) {
    firstName = std2.gFirstname();
    // ... read more attributes
    if ( condition1 ) {
        // update row
        std2.beginUpdate();
        std2.sFirstName(newName);
        // ... update more attributes
        // commmit update
        std2.endUpdate();
    } else if ( condition2 ) {
        std2.beginInsert();
        std2.sFirstName(newName);
        // .. insert more attributes
        // commit insert
        std2.endInsert();
    }
    // ... more code
}
```

Fig. 10. Usage of ABC from users' point of view

Fig. 10 presents a possible use of ABC from users' point of view. A BE_Student worker has been instantiated for the query identified with *token2*. Then the program scrolls the ResultSet (JDBC LMS implementation) and, when *condition1* is true the current row is updated, and, when *condition2* is true a new row is inserted.

Fig. 11 depicts a partial view of BE_Student source code. Java objects used in this example: *Connection* – to connect to the host database, *PreparedStatement (ps)* – to execute SQL statements and *ResultSet (rs)* to manage LMS. *beginUpdate* is an empty method because JDBC has no explicit method to start updating a row. The JDBC updating protocol is automatically started when a first update action is performed in any attribute.

```java
public BE_Student(Connection conn) {
    this.conn=conn;
}
void setSql(String sql) throws SQLException {
    ps.close();
    ps = conn.prepareStatement(sql);
}
public void execute(int id) throws SQLException {
    ps.setInt(1, id);
    rs=ps.executeQuery();
}
public boolean moveNext() throws SQLException {
    return rs.next();
}
public String gFirstname() throws SQLException {
    return rs.getString(2);
}
public void sFirstName(String firstName) throws SQLException {
    rs.updateString(2,firstName);
}
public void beginUpdate() throws SQLException {
}
public void endUpdate() throws SQLException {
    rs.updateRow();
}
public void beginInsert() throws SQLException {
    rs.moveToInsertRow();
}
public void endInsert() throws SQLException {
    rs.insertRow();
    rs.moveToCurrentRow();
}
```

Fig. 11. Partial view of BE_Student source code

7   Conclusions

In this paper an architecture based on general CLI for reusable and adaptable business components was presented. It is focused on bridging object-oriented applications and relational databases. ABC components are in line with the context of CBSE supporting the process of development for reuse and development with reuse. The reutilization intensity is determined by the chosen approach for the development of ABC components (global, activity, entity or mixed) and also by the intensity of *reuse* of computation. The ABC component main architecture relies on Business Entities

created during ABC design phase. Business Entities define the contracts, based on Business Interfaces, between ABC components and client applications. Each Business Entity is able to manage any set of SQL statements that conform to its associated Business Interface. Moreover, SQL statements may be deployed to each running instance of ABC component in an unbalanced way. This means that the same Business Entity may have different sets of SQL statements in two different running instances of the same ABC component. The SQL statements updating process is accomplished at run-time, at any moment and as often as necessary by authorized entities.

All Business Interfaces were designed to improve, from programmers' point of view, the usability of ABC components when compared to traditional Call Level Interfaces. The most significant improvement has been achieved around *getter* and the *setter* methods. Their signatures are type-safe and syntactically based on the schema of SQL statements.

The development of ABC components is completely decoupled from the development of client applications. Moreover, the process of definition and deployment of SQL statements on each running instance of ABC component may be completely controlled by authorized entities. These two issues allow the separation of concerns through the definition of two main actors: administrator programmers and common programmers.

Source code of ABC components may be automatically built by a tool, relieving programmers from their manual development and their maintenance processes.

As an outcome of this work, it is expected that this work may open new approaches to the development of business components for database applications. Future work may be divided in two stages: short term and long term.

Short term: Assess and compare ABC performance with a solution based on a standard JDBC. In spite of not being equivalent or even comparable solutions it is advisable to have an idea about the induced overhead in order to promote, if necessary, a finer performance tuning.

Long term: Investigate the possibility of creating new architectures for other reusable and adaptable business tier components. These new architectures should address different requirements such as components with a single wide range Business Interface. This wide Business Interface should eventually support and integrate all known individual Business Interfaces.

References

1. David, M.: Representing database programs as objects. In: Bancilhon, F., Buneman, P. (eds.) Advances in Database Programming Languages, pp. 377–386. ACM, N.Y (1990)
2. Cook, W., Ibrahim, A.: Integrating programming languages and databases: what is the problem? (May 2011) http://www.odbms.org/experts.aspx#article10
3. Heineman, G.T., Councill, W.T.: Component-Based Software Engineering: Putting the Pieces Together, 1st edn. Addison-Wesley (2001)
4. Costa, C., Pérez, J., Carsí, J.Á.: Dynamic Adaptation of Aspect-Oriented Components. In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 49–65. Springer, Heidelberg (2007)

5. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software 74(1), 45–54 (2005)
6. ISO. ISO/IEC 9075-3:2003 (2003),
   http://www.iso.org/iso/catalogue_detail.htm?csnumber=34134
   (2010/May 2011)
7. Schmoelzer, G., et al.: Model-typed Component Interfaces. In: 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2006 (2006)
8. Flower, M.: Patterns of Enterprise Application Architecture. Addison-Wesley (2002)
9. Christian, B., Gavin, K.: Hibernate in Action. Manning Publications Co. (2004)
10. Oracle TopLink (May 2011),
    http://www.oracle.com/technetwork/middleware/toplink/overvie
    w/index.html
11. Kulkarni, D., et al.: LINQ to SQL: .NET Language-Integrated Query for Relational Data. Microsoft
12. Microsystems, S.: JDBC Overview (May 2011),
    http://www.oracle.com/technetwork/java/overview-141217.html
13. Microsoft. Microsoft Open Database Connectivity (May 2011),
    http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx
14. Elizondo, P.V., Lau, K.-K.: A catalogue of component connectors to support development with reuse. Journal of Systems and Software 83(7), 1165–1178 (2010)
15. Kung-Kiu, L., Zheng, W.: Software Component Models. IEEE Transactions on Software Engineering 33(10), 709–724 (2007)

Improving Quality of Business Process Models

Laura Sánchez-González, Francisco Ruiz, Félix García, and Mario Piattini

Alarcos Research Group, TSI Department, University of Castilla La Mancha
Paseo de la Universidad, nº4, 13071, Ciudad Real, Spain
{Laura.Sanchez,Francisco.RuizG,Felix.Garcia,
Mario.Piattini}@uclm.es

Abstract. Business process improvement is a key aspect for organizational improvement. We focus the business process improvement in the first stage of process lifecycle, design stage, because it is a means to avoid the propagation of errors to later stages, in which their detection and correction may be more difficult. Since business process improvement is centered in business process models, a proposal of certain steps based on measurement activities on conceptual models (measurement, evaluation and redesign) is described. The application of these steps in business process models produces an increase of the quality of them. Quality is defined as the level of understandability and modifiability, subcharacteristics of the usability and maintainability in ISO 9126. The steps for model improvement have been applied to a real hospital business process model. The model was modified by following expert opinions and modeling guidelines, thus leading to the attainment of a higher-quality model. Our findings clearly support the practical utility of measurement activities for business process model improvement.

Keywords: Business process, Measurement, Continuous improvement, BPMN.

1   Introduction

In recent years, business process (BP) modelling and improvement has become an important means of ensuring changes in an organization's structure and functioning, thus leading to the creation of a more competitive and successful enterprise [1]. BP influences product quality and customer satisfaction, which are fundamental aspects in a market environment, and enterprises are therefore forced to improve their processes in order to improve products and services [2].

The first step towards improving business processes is to collect any data regarding their design, deadlocks, bottlenecks, etc. Measurement is a good means of collecting this kind of data, and serves at least the following three purposes: understanding, control and improvement [3]. The use of measurement information therefore makes it possible for organizations to learn from the past in order to improve their performance and achieve better predictability over time.

A business process is a complex entity with a characteristic lifecycle. In our work we consider the approach defined by Weske [4], who organizes the lifecycle in a

cyclic structure with logic dependences between the design and analysis, configuration, enactment, and evaluation stages. We focus on the first stage, *design and analysis*, in which the principal activity is that of process modelling. The main purpose of design and analysis is to capture the business schema and general procedures [5]. The conceptual models produced in this stage are first required to be intuitive and easily understandable in order to facilitate communication among stakeholders. Measuring and improving BP models has several advantages, principally that of avoiding the propagation of errors or bad-structures to later lifecycle stages, in which corrections and modifications may involve a high economic cost and effort [6].

Measures for conceptual models deal with the static properties of BP and are defined upon the BP model at the time of the design. Several initiatives concerning the measurement model have recently been published, owing to the advantages of improving business processes in this stage. Most of the measures published to date have been collected in [7]. This work shows that there is no consensus among researchers as to which measurable concepts it is most interesting to measure (complexity, structuredness, cohesion, coupling, etc). It also highlights that most of the proposals have not been empirically validated. This lack of validation particularly emphasises the need for research in this area. The work presented herein contributes to the maturity of BP measurement through the collection of measures and the demonstration of their practical utility in an experience report.

The principal idea behind our proposal is to apply measurement during the early stages of the lifecycle, the *design and analysis stage*, in order to obtain feedback controlled by measures and thereby achieve a higher-quality implementation of the process, with a lower value of complexity, therefore making it easier to maintain [8]. The measurement process is divided into three activities: applying measures, evaluating measurement results and redesigning the model. The pragmatic idea of these activities is to discover unsafe design, hazardous structures or unexpected. Finally, one critical aspect of the improvement activities is to demonstrate that they are potentially useful in practice. We therefore present an experience report of the application of improvement activities to a real hospital business process.

The remainder of the paper is as follows. In Section 2 we describe the improvement activities in which measures were applied, evaluate the measurement results and redesign the model. In Section 3 we present an experience report of the application of these activities to a real business process, specifically a hospital business process. In Section 4, we describe some implications and limitations of this research. Finally, Section 5 shows our conclusions and presents topics for future research.

2   Business Process Model Improvement

In this article, we propose certain activities for business process model improvement. The principal idea is to collect as much information as possible about the static properties of the business process. The activities are: applying measures collected in

previous works, evaluating measurement results against threshold values and redesigning the model. These three activities can be executed in a cyclic manner, signifying that multiple iterations can be run to obtain a high-quality model. This idea is depicted in greater detail in Figure 1.

In Figure 1 the lifecycle stages are represented as a square and the improvement activities as ellipses. The *design and analysis stage* initially produces a conceptual model. This model serves as input for the improvement activities. The improvement of the model can be carried out in several iterations of the 3 activities (*measurement, evaluation and redesign*). These activities can be introduced in the BP lifecycle as an extended stage, which can enrich the final product. After the *configuration stage*, the execution model is enacted through the generation of log files, which describe all the steps followed to achieve the business goals. These log files can be measured (processed) in order to discover certain important aspects such as execution time, deadlocks, etc. The measurement initiatives for improvement in the execution stage are described in [9]. The evaluation of these execution reports implies the generation of new business requirements which had not previously been considered.
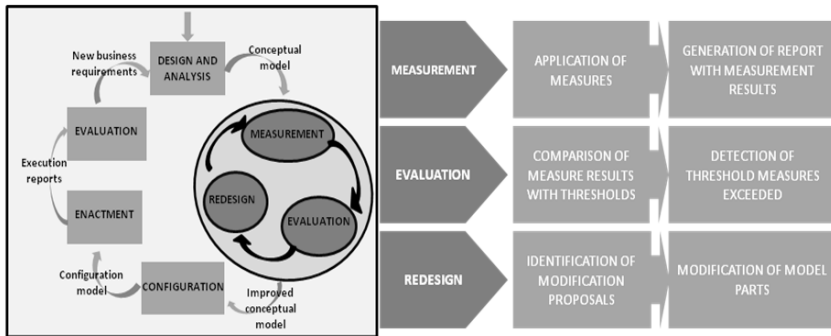
Fig. 1. Improvement activities in BP lifecycle

2.1   Measurement of Business Process Conceptual Models

In recent years, the number of measurement approaches for conceptual models has grown considerably owing to the advantage of improving business processes in the early phases. BP model measures are used to quantify structural aspects of models, which signifies measuring their internal quality. This internal quality is understood as the model's total number of characteristics from an internal view, and this is measured and evaluated against the internal quality requirements [10]. Internal quality (quality in general) can be seen from different points of view, and should therefore be quantified with more than one measure in order to obtain as much information as possible with regard to the model. For example, model complexity cannot be measured solely with the Control-flow Complexity (CFC) measure, because this measure only takes into account decision node elements.

As we mentioned, various measures are found in literature [7], and Table 1 specifically shows references to their measurement initiatives and provides a brief description of them.

However, it is also important to consider external quality in conceptual models. External quality refers to the total number of characteristics in the model from an external view [10], such as how understandable the models are, how difficult it is to modify them, etc. Several authors tried to define a group of quality characteristics for conceptual models [11-13]. However, there is no a consensus about what makes a good model and the list of quality characteristics is not still defined. In this paper, we highlighted the quality characteristics of understandability and modifiability because one of the most important purposes of business process models is the communication between stakeholders. Since a business process models is a communication vehicle, the correct understandability of them is a key aspect in business process development. From the point of view of a top-down quality SEQUAL framework [14], understanding is an enabler of pragmatic quality, which relates to model and modelling and its ability to enable learning and action. On the other hand, a business process models should be easy to modify, for example for adding new business requirements, so it is considered modifiability as another important quality characteristic in this context. In order to clarify this idea, Figure 2 shows the relationship between internal and external quality and some examples of measurable attributes. Both dimensions (internal and external quality) are related though the cognitive complexity, as it was indicated in [15]: "cognitive complexity is the mental burden of the persons who have to deal with the model, so high cognitive complexity of a model causes it to display undesirable external qualities. The external quality attributes are therefore indicators of the cognitive complexity".

Most authors have carried out experiments focused on the relationship between measures and external quality attributes: understandability and modifiability. These belong to the more general concepts of usability and maintainability respectively [10].

Table 1. Proposals of measures for business process models

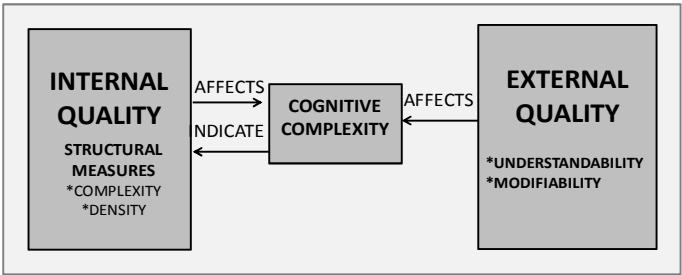| Measure	Description
Coupling, cohesion and connectivity level [16, 17]	Cohesion and coupling between activities and cross connectivity in the relationship between nodes and directed arcs.
Structural complexity [18]	Measures related to the number of different elements of BPMN models.
Error probability [8]	Number of nodes, diameter, gateway mismatch, depth, density, average and max connector degree, cyclicity, sequentiality and separability.
Control flow complexity [2]	Related to the number of OR-split, AND-split and XOR-split
Entropy [19]	Uncertainty or variability of workflow process models
Structuredness [20]	Number of unstructured parts
Complexity [21]	Activity, control-flow, data-flow and resource complexity
Goodness [22]	Goodness of models regarding execution logs

Fig. 2. Internal and external quality in conceptual models

Table 2. Empirically validated measures and their relationship with understandability and modifiability

Measure	Description	U*	M*
Measures of Rolón [18]			
TNSF	Total Number of sequence flows	X	
TNE	Total Number of events	X	
TNG	Total Number of gateways	X	
NSFE	Number of sequence flows from events	X	
NMF	Number of message flows	X	
NSFG	Number of sequence flows from gateways	X	X
CLP	Connectivity level between participants	X	
NDOOut	Number of data objects which are outputs of activities	X	
NDOIn	Number of data objects which are inputs of activities	X	
CLA	Connectivity level between activities		X
Measures of Cardoso [2]			
CFC	Control flow complexity. Sum over all gateways weighted by their potential combinations of states after the split	X	X
Measures of Mendling [8]			
Number of nodes	Number of activities and routing elements in a process model	X	
Gateway mismatch	Sum of gateway pairs that do not match each other, e.g. when an AND-split is followed by an OR-join	X	X
Depth	Maximum nesting of structured blocks in a process model	X	
Connectivity coefficient	Ratio of total number of arcs in a process model to its total number of nodes	X	
Density	Ratio of total number of arcs in a process model to the theoretically maximum number of arcs		X
Sequentiality	Degree to which the model is constructed from pure sequences of tasks	X	X

To the best of our knowledge, very few articles concerning the relationship between measures for internal quality and measures for external quality have been published to date, although some research has been published in [23-25], and these works obtained a subgroup of measures which can be considered as good indicators

for understandability and modifiability. This subgroup of measures is shown in Table 2. The application of this subgroup of measures is produced in a pair *(measure, result)*, which should be reported in a document in order to be used in next activity: evaluation.

Table 3. Thresholds for business process model measures

	1: very inefficient	2: fairly inefficient	3: fairly efficientt	4: very efficient
Understandability				
Nºnodes	65	50	37	31
GatewayMismatch	29	16	6	1
Depth	4	2	1	1
Coefficient of connectivity	1,7	1,1	0,6	0,4
Sequentiality	0,1	0,35	0,6	0,7
TNSF	72	49	34	20
TNE	20	12	7	2
TNG	17	10	5	0
NSFE	28	13	4	0
NMF	27	15	7	1
NSFG	40	22	11	0
CLP	7,5	4,23	2,2	0,2
NDOIN	31	44	4	0
NDOOUT	23	11	3	0
CFCxor	30	17	8	1
CFCor	9	4	1	0
CFCand	4	2	0	0
Modifiability				
GatewayMismatch	46	22	4	1
Densitiy	0,6	0,22	0,001	0
Sequentiality	0	0,18	0,6	0,86
NSFG	25	13	9	0
CLA	0,53	0,875	1,1	1,3
CFCxor	27	16	8	1
CFCor	9	4	1	0
CFCand	6	2,3	0	0

2.2 Evaluation of Measurement Results

The evaluation of measurement results involves providing an objective assessment of them. Numerical results only offer information in terms of comparison between models rather than an independent interpretation. For example, given two process models, it is possible to discover not only which of them is best in the relative terms of a specific measure, but whether the values are acceptable or not. It is therefore necessary to consider the threshold or limit values in order to indicate for what specific value the measure's quality begins to decline.

Various proposals for the extraction of threshold values exist in literature, principally in the Software Engineering field. Some proposals for thresholds are derived from experience [26-28], but the lack of scientific support has led to disputes about their values. Some authors, on the other hand, have used statistical techniques to obtain thresholds. For example, Shatnawi [29] extracted thresholds for Object Oriented (OO) code measures in order to study the relationship between OO and error-severity categories. This author also validated the Bender method [30] and found that there are effective thresholds for the measures analyzed.

With regard to business process measurement, we have attempted to extract threshold values for some measures in previous works. This is the case of *Control-flow complexity* measure, *structural complexity* and *error probability* measures, which were used to apply the Bender method in order to extract thresholds. These works were published in [31, 32]. Table 3 shows extracted thresholds for some empirically validated measures. This table divides the domain of the measure into 4 different groups, depending on the level of efficiency: "very efficient", "fairly efficient", "fairly inefficient" and "very inefficient".

2.3   Redesign of Business Process Models

In this section, we focus on modifying some parts of the model in order to improve its general quality. Those parts that are candidates for alteration have been identified through the use of measures. For example, let us imagine that we are analyzing the results of the CFC measure in a specific model, and we obtain a numerical value which is higher than the threshold: "If CFC is higher than 44, the model is difficult to understand". These results indicate that the number of decision nodes must be reduced in the model, since it may be difficult for stakeholders to understand.

Nevertheless, modifying the model using only the information collected from measures and thresholds can be quite difficult. Some guidelines therefore exist to assist modellers in this task. In literature, it is possible to discover various guidelines for inexpert modellers, whose purpose is to obtain higher-quality models that can ensure a more reliable execution. Mendling et al. [33] proposed seven pieces of advice for modellers (denominated as 7PMG) which are built on strong empirical insight. This advice is related to the maximum number of nodes before decomposition, number of events, OR-routing elements, routing paths per element or the use of a verb-object activity label. On the other hand, Becker et al. [34] define certain guidelines of modelling (GoM), which are specifically six general techniques for adjusting models to the perspectives of different types of user and purposes. To illustrate the used of these guidelines, let us imagine the following example. If the measure "total number of events" is higher than 20 (very inefficient), 7PMG advises that the use of "one start and one end event" is the best way to reduce the measure value.

Redesign therefore involves changing those specific parts of the model with low quality detected by measures. Modelling guidelines can also help to ensure the quality of the model but a previous measurement effort is necessary to identify any potential problems.

3 Experience Report: Hospital Process

In order to demonstrate the practical utility of this proposal, we describe an experience report which was developed in the General Hospital of Ciudad Real (GHCR) in Spain. First, a specific work group was created, consisting of specialists in modelling tasks (Software Engineers) and health professionals at the hospital:

a) Those responsible for processes: the assistant director of nursing and the person responsible for hospital's admissions units.

b) Collaborators: head of human resources and finances, head of computer services and head of out-patients' healthcare.

The work group then modelled various processes which had previously been selected by the hospital's managerial and quality staff, although in this paper we shall focus on the "Incorporation of a new employee" (INE) process, which includes the training plan, information and suitability of those people involved in the hospital in order to facilitate their integration into the new job. The process model is shown in Figure 3.

Fig. 2. BPMN model for the Incorporation of the New Employee (INE) hospital process

This process was selected as a low-complexity process, although the services provided are very important. It is a purely administrative process (it is not related to patient care), but moves a large number of users (in 2007, the hospital staff consisted of 2.600 workers, and 6989 new contacts were made with regard to substitutions and new incorporations). This process involves different professional categories: doctors, pharmacists, nurses, psychologists, administrative and technical staff and others. Specific process characteristics were the following:

a) Mission: to promote the organization of the INE process, which includes a plan for training, information and adaptation of the people involved to the hospital requirements in order to facilitate their integration into the new job.

b) Limits: the INE process starts when the professional comes to the hospital and finishes when he/she is incorporated into the new job.

c) Clients: new professionals

d) People responsible: those responsible for nursing, medical aspects and management.

e) Participants: new professionals in hospital, human resources, computer services, lingerie, pharmacy, prevention services, nursing and management service.

f) Suppliers: human resources, provisions, maintenance, training and information systems.

The results of the application of the improvement activities are described in the following sub-sections:

3.1   Applying Improvement Activities

The design of the INE process model is represented in BPMN [35] (Figure 3), the *de facto* standard for BP modelling. This conceptual model was a candidate for improvement. We therefore applied the three measurement activities previously presented.

A) **Measurement.** We applied most of the measures published to date, particularly those measures which had been empirically validated. It was not possible to apply all of them owing to the absence of certain elements in this specific model. The results obtained are shown in Table 4 (pair measure/result).

B) **Evaluation.** After obtaining the measurement results, we evaluated them by following the threshold values shown in Table 3. The conclusions were as follows:

▪ Number of nodes is 59, so the model is fairly inefficient in understandability tasks

▪ Density is 0.02, so the model is fairly efficient in modifiability tasks

▪ Sequentiality is 0.396, so the model is fairly inefficient in understandability and modifiability tasks

▪ Connectivity coefficient is 1.54, so the model is very inefficient in understandability tasks

▪ Mismatch connector is 16, so the model is fairly inefficient in understandability and modifiability tasks

▪ Control flow complexity is 22, so the model is fairly inefficient in understandability and modifiability tasks

▪ CLA is 0.61, so the model is very inefficient in modifiability tasks

▪ CLP is 3, so the model is fairly efficient in understandability tasks

▪ TNE is 5, so the model is fairly efficient in understandability tasks

▪ TNSF is 73, so the model is very inefficient in understandability tasks

▪ NMF is 18, so the model is fairly inefficient in understandability tasks

After the evaluation, we detected some potential parts for alteration. For example, number of nodes was a very high value, and could have compromised the

understandability of the model. The same applies to connectivity coefficient, control-flow complexity, CLA and TNSF, which obtained the worst results of the measurement activity. On the other hand, density, CLP and TNE obtained acceptable results and did not need to be analyzed for further improvement initiatives. These results guided us in our definition of some proposals for redesign.

Table 4. Measurement results for the INE process

Measure	Result	Understandability	Modifiability
Nº of nodes	59	Fairly inefficient	-
Density	0,02	-	Fairly efficient
Sequentiality	0,396	Fairly inefficient	Fairly inefficient
Connectivity coefficient	1,54	Very efficient	-
Mismatch connector	16	Fairly inefficient	Fairly inefficient
Control flow complexity	22	Fairly inefficient	Fairly inefficient
CLA	0,61	-	Very inefficient
CLP	3	Fairly efficient	-
TNE	5	Fairly efficient	-
NSF	73	Very inefficient	-
NMF	18	Fairly inefficient	-

C) **Redesign.** After the selection of those parts of the INE model that are potential elements for modification, the redesign activity is carried out. This is the most critical activity, since it depends on the successful implementation of improvement activities.

Redesign was classified into two different groups: changes proposed by specialists in modelling tasks following guidelines of modelling and changes proposed by health professionals.

Changes Proposed by Health Professionals:
Professionals at the hospital proposed certain modifications which implied some differences in the way in which some parts of the model were designed.

The work group created to model tasks proposed changes which produced several semantically equivalent models. The Dephy method [36], was used to allow the work group to select the most suitable changes. Each of these changes produces a different version to the original, specifically 4 different versions are generated:

A) The "belongs to a nursing unit with medical dispenser" decision node was eliminated in the immediately superior lane.

B) Some activities were added: "complete pharmacy report" "send registration request to pharmacy services", "receive registration in computer services", "inform the employee" in the immediately superior lane.

C) The "belongs to a nursing unit" decision node was eliminated and another decision node was added in order to distinguish two categories: planned or urgent in specific superior lane.

D) Combination of version B and C.

The work group's opinion and a first application of the measures revealed that version D is the best option, and we selected it as the candidate for the improved conceptual model. The results of these changes are depicted in Figure 4. This change obtained better results with regard to measures in comparison to the original model. Table 5 shows the measures analyzed and the results obtained. The measurement values for the original model are shown in brackets for the purpose of comparison. This comparison shows an evident improvement in the model quality.

Table 5. Measure values of the improved model generated by health professionals

Measure	Result
Mismatch connector	16 (15)
Control flow complexity	22 (21)
CLA	0,61(0,64)
NSF	73 (71)
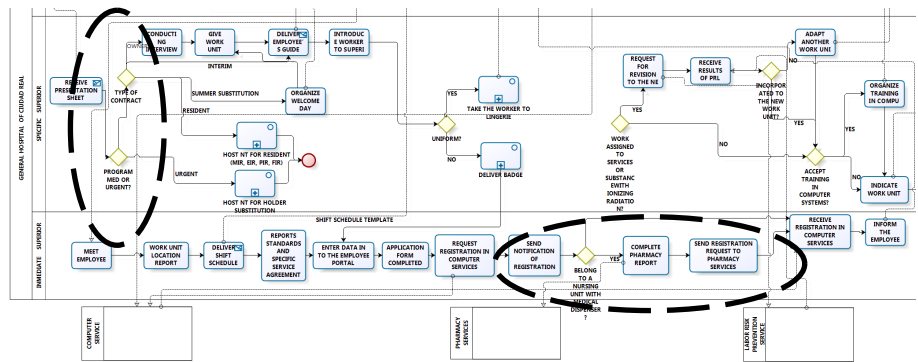
Fig. 3. Model of INE process applying changes proposed by health professionals

Changes Proposed Following Guidelines of Modeling:
On the other hand, the changes proposed by modelling experts was based on the *guidelines for modellers* published in [33]. The following modifications were therefore applied to the INE process model:
1. To reduce number of nodes:
   a. Decompose a model with more than 50 elements.
   b. Use one start and one end event.
2. To reduce TNF:
   a. The elimination of some nodes reduces the number of sequence flows.
3. To reduce NMF:
   a. The grouping of activities in a subprocess reduces the number of messages.
4. To reduce control-flow complexity and mismatch connector:
   a. Avoid OR routing elements.
5. A further improvement that is not taken into account in the measures is "use verb object activity labels".

The proposed changes to the model are depicted in Figure 5, and the measures' improved results are described in Table 6.

Table 6. Measure values of the improved model generated by IT expert

Measure	Result
Nº of nodes	48(59)
Sequentiality	0,47(0,396)
Mismatch connector	10(16)
Control flow complexity	19(22)
TNE	3(5)
NSF	63(73)
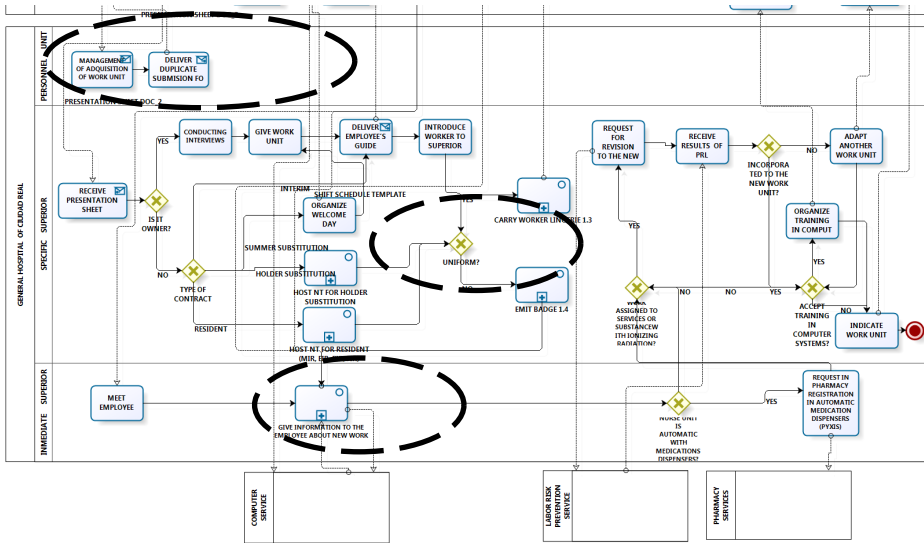
Fig. 5. Version of INE process, including changes proposed following guidelines of modeling

E) Selection of the Improved Business Process Design. The application of measures in both alternatives allowed us to discover that the most acceptable design is that obtained by professionals in modelling. Specifically, 35% of the measures analyzed improved their values when following guidelines for modelling, as opposed to 23% of the measures obtained when following the advice of professionals in the health sector. This signifies that the conceptual model depicted in Figure 5 obtained better measurement results, thus suggesting that the model is a good choice and can increase the probability of obtaining a correct process enactment.

4   Implications and Limitations

In this section we highlight some of the implications and limitations of our research. In the previous section, we described the process used to improve conceptual models. In the first part, some measures were applied to an INE process model, obtaining

certain measurement results. One limitation is related to applied measures. Although more measurement initiatives have been published, it is not possible to apply them because of their lack of empirical validation. This is an important disadvantage in business process measurement and may have limited our research.

On the other hand, measurement values were assessed by following thresholds in order to guide us in redesigning tasks. In a real situation (Incorporation of a new employee) we had two different initiatives for redesigning. One of them was based on the opinion of health experts. After seeing some business issues as a conceptual model, represented in BPMN, they discovered that some parts can be realised in a different way with the same results. These changes to the original model were made, and some improvements were made to the measures (i.e. Control flow complexity was 21 rather than 22 in the original model). Nevertheless, some improvement initiatives can be also be made by following theoretical guidelines, with which even better results are obtained (nº of nodes, sequentiality, mismatch connector, control flow complexity, TNE and NSF). These results reveal that theoretical guidelines produce better modification proposals than changes based on experience. Despite this result, we believe that the changes proposed by guidelines should not be applied in isolation, but should be accompanied by the opinions of domain experts. If the BP is modified by domain experts in a controlled manner, it will be possible to avoid the rejection of changes in the lifecycle enactment stage.

5   Conclusions and Future Work

We conclude this article by summarizing its contributions and by providing an overview of future research. We have discussed the importance of measuring business processes, specifically in the design and analysis stage, because it is known that improving conceptual models in the first stage implies several advantages in the case of avoiding the propagation of errors to later stages, in which their elimination might be more difficult and expensive. This finding has a strong implication for the way in which business process improvement is confronted. A high-quality conceptual model can therefore ensure an acceptable execution.

The experience report allows us to demonstrate the practical utility of measurement activities, obtaining a higher-quality model. The application of measurement to conceptual models detected some potential parts for alteration (number of nodes, reducing sequence and message flow, reducing decision nodes, or reducing number of events). Guidelines of modelling also assisted us in making these modifications. Finally, we obtained an improved quality model which can ensure a better execution.

As a future work, we wish to provide more empirically validated measures in order to make the measurement process more reliable. We also intend to design more guidelines for inexpert modellers. Finally, our idea is to apply measurement activities in other real business processes at the hospital and in other real organizations in order to ensure their practical utility.

References

1. Damij, N., et al.: A methodology for business process improvement and IS development. Information and Software Technology 50(11), 1127–1141 (2008)
2. Cardoso, J.: Process control-flow complexity metric: An empirical validation. In: SCC 2006: Proceedings of the IEEE International Conference on Services Computing, pp. 167–173 (2006)
3. Park, R.E., Goethert, W.B., Florac, W.A.: Goal-Driven software Measurement: A Guidebook. Handbook CMU/SEI-96-HB-002 (1996)
4. Weske, M.: Business Process Management: Concepts, Languages, Architectures, 1st edn. Springer, Heidelberg (2007)
5. Sparks, G.: An Introduction to UML, The Business Process Model. Enterprise Architect (2000)
6. Wand, Y., Weber, C.: Research commentary: Information systems and conceptual modeling–a research agenda. Info. Sys. Research 13(4), 363–376 (2002)
7. Sánchez-González, L., et al.: Measurement in Business Processes: a Systematic Review. Business Process Management Journal 16(1), 114–134 (2010)
8. Mendling, J.: Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness. Springer Publishing Company, Incorporated (2008)
9. Delgado, A., Ruiz, F., García-Rodríguez de Guzmán, I., Piattini, M.: MINERVA: Model drIveN and sErvice oRiented Framework for the Continuous Business Process improVement and relAted Tools. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave 2009. LNCS, vol. 6275, pp. 456–466. Springer, Heidelberg (2010)
10. ISO/IEC, 9126-1, Software engineering - product quality - Part 1: Quality Model (2001)
11. Moody, D.: Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. Data and Knowledge Engineering 55, 243 (2005)
12. Lindland, O.I., Sindre, G., Solvberg, A.: Understanding Quality in Conceptual Modeling. IEEE Software 11(2), 42–49 (1994)
13. Shanks, G., Tansley, E., Weber, R.: Using ontology to validate conceptual models. Commun. ACM 46(10), 85–89 (2003)
14. Krogstie, J., et al.: Process models representing knowledge for action: a revised quality framework. Eur. J. Inf. Syst. 15(1), 91–102 (2006)
15. Briand, L.C., Wüst, J., Ikonomovski, S., Lounis, H.: A Comprehensive Investigation of Quality Factors in Object-Oriented Designs. An Industrial Case Study. Technical Report ISERN-98-29 (1998)
16. Vanderfeesten, I., et al.: Quality Metrics for Business Process Models. In: BPM and Workflow Handbook (2007)

17. Vanderfeesten, I., Reijers, H.A., Mendling, J., van der Aalst, W.M.P., Cardoso, J.: On a Quest for Good Process Models: The Cross-Connectivity Metric. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 480–494. Springer, Heidelberg (2008)
18. Rolón, E., García, F., Ruiz, F.: Evaluation Measures for Business Process Models. In: Simposioum in Applied Computing, SAC 2006 (2006)
19. Jung, J.Y.: Measuring entropy in business process models. International Conference on Innovative Computing. Information and Control, 246–252 (2008)
20. Laue, R., Mendling, J.: Structuredness and its Significance for Correctness of Process Models. Information Systems and E-Business Management (2009)
21. Meimandi Parizi, R., Ghani, A.A.A.: An Ensemble of Complexity Metrics for BPEL Web Processes. In: Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, pp. 753–758 (2008)
22. Huan, Z., Kumar, A.: New quality metrics for evaluating process models. In: Business Process Intelligence Workshop (2008)
23. Rolon, E., et al.: Prediction Models for BPMN Usability and Maintainability. In: BPMN 2009 - 1st International Workshop on BPMN, pp. 383–390 (2009)
24. Sánchez González, L., et al.: Assessent and Prediction of Business Process Model Quality. In: CoopIS 2010 - 18th International Conference on Cooperative Information Systems, pp. 78–95 (2010)
25. Rolón, E., et al.: Analysis and Validation of Control-Flow Complexity Measures with BPMN Process Models. In: The 10th Workshop on Business Process Modeling, Development, and Support (2009)
26. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering SE-2(4), 308–320 (1976)
27. Nejmeh, B.A.: NPATH: a Measure of Execution Path Complexity and its Applications. ACM 31(2), 188–200 (1988)
28. Coleman, D., Lowther, B., Oman, P.: The Application of Software Maintainability Models in Industrial Software Systems. Journal of Systems and Software 29(1), 3–16 (1995)
29. Shatnawi, R.: A Quantitative Investigation of the Acceptable Risk levels of Object-Oriented Metrics in Open-Source Systems. IEEE Transactions on Software Engineering 36(2), 216–225 (2010)
30. Bender, R.: Quantitative Risk Assessment in Epidemiological Studies Investigatin Threshold Effects. Biometrical Journal 41(3), 305–319 (1999)
31. Sánchez-González, L., et al.: Towards Thresholds of Control Flow Complexity Measures for BPMN Models. In: 26th Symposium On Applied Computing. SAC, vol. 10 (in press, 2011)
32. Sánchez-González, L., et al.: Quality Assessment of Business Process Models Based on Thresholds. In: CoopIS 2010 - 18th International Conference on Cooperative Information Systems, pp. 78–95 (2010)
33. Mendling, J., Reijers, H.A., Van der Aalst, W.: Seven Process Modeling Guidelines (7PMG). Information and Software Technology 52(2), 127–136 (2010)
34. Becker, J., Rosemann, M., von Uthmann, C.: Guidelines of Business Process Modeling. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, pp. 30–49. Springer, Heidelberg (2000)
35. OMG. Business Process Modeling Notation (BPMN), Final Adopted Specification (2006), http://www.omg.org/bpm
36. Linstone, H.A., Turoff, M.: The Delphi Method: Techniques and Applications. Addison-Wesley (2002)

Team Radar: A Radar Metaphor
for Workspace Awareness

Cong Chen and Kang Zhang

Department of Computer Science, University of Texas at Dallas
800 West Campbell Road, Richardson, U.S.A.
{congchen,kzhang}@utdallas.edu

Abstract. In distributed software teams, awareness information is often lost due to communication restrictions. Researchers have attempted to retain team awareness by sharing change information across workspaces. The major challenge is how to convey information effectively while avoiding information overload. In this paper, we propose a radar metaphor for distributing and visualizing workspace awareness information. A prototype implementation, Team Radar, is presented and its design is also discussed.

Keywords: Collaboration, Workspace awareness, Visualization, Software con-figuration management.

1   Introduction

Software development is in general a collaborative activity. The complexity of the code itself and the complexity of the activities and process of producing it make such collaboration difficult [7]. One of the most frequently reported causes of these problems is the lack of awareness [16], which is typically defined as "an understanding of the activities of the others, which provides a context for one's own activities" [4].

In co-located teams, such information is maintained either through informal interactions among developers, such as monitoring each other's activities, informal conversations, pair programming sessions, and expert assistance [11], or through inspecting documents and source code, shared in software configuration management (SCM) systems [16].

When direct communication is restricted, e.g., the team is geographically distributed; people often struggle with coordination and collaboration because awareness information is lost. Moreover, studies show that loss of awareness even affects developers' willingness to collaborate and enthusiasm of work [12]. In such a setting, people have to take various alternative approaches to obtain awareness. One of the most common sources of awareness information is software repository, such as SCM repository. Developers traditionally used an SCM system to track and control changes of artifacts by imposing concurrency control and version control regulations. As it stores all relevant changes and events in the project, researchers now find SCM repository valuable to work as an organizational memory that can be accessed to find out what other developers have done [12].

SCM systems, however, fail to offer sufficient level of awareness, because their asynchronized propagating strategy isolates local changes until developers manually submit them. In order to alleviate this problem, and "break bad isolation while retaining good isolation" [20], a number of researchers have argued that the key to promote coordination among de-located teams is increasing the level of awareness and providing real-time information of ongoing changes [15].

We claim that awareness in SCM could be enhanced with additional communication mechanism that continuously exchanges information between workspaces. We could also enrich the team memory by supplying the existing software repository with additional awareness information, and promote mining software repositories (MSR) research to a fine-grained level.

The main challenge we are facing now is how to convey sufficient amount of information to team members while avoiding information overload. Our solution is a visualization that intuitively shows the most useful information to team members, and appeals to them as much as possible. We developed *Team Radar*, a workspace awareness supporting tool based on *Qt Creator* [18], an open source C++ IDE from Nokia. Team Radar monitors and captures changes in local workspaces and in SCM repository, extracts and analyzes the embedded awareness information, distributes it to other workspaces, and finally presents it in a visually attractive fashion.

The major innovation of our approach is that by applying afterimage technique and radar metaphor, we create a continuous and coherent team memory, which blends past with present, and more efficiently promote users' interests.

2   Related Work

There are a number of approaches in the community attempting to improve workspace awareness by enhancing existing SCM systems.

Palantir [21] is an SCM enhancement that takes awareness into account. Palantir informs a developer of which other developers change to which artifacts, calculates a severity measure of potential conflict, and graphically displays the information. Palantir does not intend to solve conflict problem by itself. It simply makes developers aware of potential conflict and relies on them to avoid it before it happens.

An important aspect of software project is its evolution. Gource [3] is a recent project on evolution visualization, which differs from previous work by clearly showing the structure of the code and the relationships between artifacts and authors. Gource takes a qualitative approach and uses animation to visualize the flowing history of a project mined from SCM repositories. It renders the project structure as a dynamic tree, rendered by a force-directed tree layout algorithm [10]. Nodes represent files, and are connected to the tree by edges. Currently contributing authors fly close to the files, sending out beams to indicate their relations.

Recent researches by Fitzpatrick et al. [6] reflect a move away from managing activities and workflow to providing visualizations of information that already exists in tools. Syde [11] follows this trend by integrating awareness information visualization tightly into existing IDEs. The author claims that despite of prolific applications of supporting workspace awareness, there is still no such a tool that provides enough fine-grained change information, and maintains a non-intrusive

approach. Scamp [15], built upon the communication infrastructure offered by Syde, extends Syde by delivering awareness information with three lightweight visualizations.

Our work is inspired by some of the previous approaches. We use an architecture similar to that of Palantir. The tree presentation of project structure comes from Gource. We employ an informal approach as Gource and Syde do. However, our approach differs from them in the following ways: our visualization is based on several new visual effects and metaphors, which stimulate users' imagination and engagement. We support both real-time monitoring and offline review, which is beneficial to both development and management.

3 Team Radar

Team Radar is an infrastructure aiming at enhancing workspace awareness. It is a client-server application. The client is a Qt Creator plug-in, which monitors local events, distributes them to all other workspaces, and finally renders them on a virtual radar screen. The server side acts as a communication center and a standalone team memory, which complements conventional SCM system's function of supporting awareness information.

3.1 Design Rationale

There are several important decisions we have made in designing Team Radar, which reflect the rationale and philosophy of our understanding of awareness support. When automating software development, some previous work tends to offer all-in-one solutions. That is why their tools automatically inform users of their inference and give exact instructions. While our philosophy is that since most failures in computer systems are caused by human mistakes [19], it is more appropriate to let human make the final judgment. We believe that informal awareness information helps formal processes to work [8]. Hence, Team Radar takes an informal and qualitative approach, and simply visualizes extracted awareness information without distracting developers from their main work.

Another important issue of designing an awareness supporting system is whether the system is intended for retrospective analysis of historical data, or it is used to analyze a project currently in progress. Of course, each approach has its own advantages. Most previous work, however, focuses more on either aspect of the project over the other. In our solution, we attempt to offer users a consistent and coherent team memory by unifying both past and present information in one visualization. Developers can use Team Radar to monitor coworkers' activities and coordinate collaboration, while managers may review and analyze the project by replaying the event scripts stored in Team Rader.

3.2 Architecture

Fig. 1 shows the architecture of Team Radar, which adopts the design of previous work [21]. The system is an extension of Qt Creator. The client is a Qt Creator

plug-in with five major modules. Qt Creator relies on signals to propagate events. The *Collector* is such a module that connects to its interested signals, and is notified when these signals are emitted. The *Viewer* is the visualization module that presents awareness information to users with animations. On the server side, typically on a separated site, the *Receiver* listens and accepts events from clients' Collectors, stores them into an extra *Repository*, and then asks the *Distributor* to broadcast them to other clients' Viewers. The Viewer can also retrieve the event scripts in the repository and replay them offline. Offline playback enables managers to inspect daily activities, review the process and analyze collaboration issues.
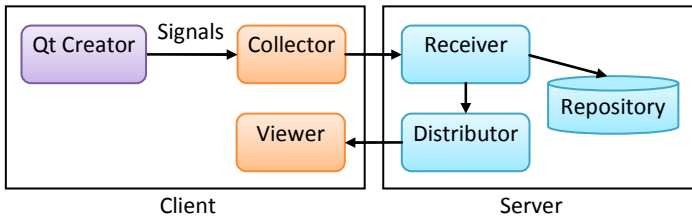
Fig. 1. Team Radar architecture

3.3   Capturing Local Events

Based on Gutwin's knowledge elements of awareness [9], workspace should track several types of awareness information, categorized by "how, when, who, where, and what" questions. In addition, a survey conducted in Microsoft shows that the majority of information needs are about discovering, meeting, and keeping track of people, not just code [2]. Hence, our work focuses more on tracing what developers are working or have worked on, rather than what specific changes they have made. In more detail, we address these aspects of collaboration:

• **Working Mode.** As a typical software development scenario, developers switch back and forth among several activities, or working modes in Qt Creator, including designing, coding, testing, debugging, reading documents, etc. Working mode could also label current progress of the project. No matter what process model the project follows, in different phases of the project, developers carry out each type of activity with various emphasis and intensity. In earlier phases, developers may take more time in designing and coding mode, while in later phases, more effort might be put to testing and debugging [16].

• **Current Changes.** It is important for developers to be aware of who else is working on the same artifacts or those artifacts closely related. Failing to acquire such information may lead to duplicated work, merge conflicts, and perhaps build failure [11]. Showing developers what artifacts others are changing gives them an early warning of potential conflict.

• **Past Changes.** In a software project, knowledge of others' activities, both past and present, has equal value for assisting the overall cohesion and effectiveness of the team. Observation of the evolution of a project helps to understand the history and

rationale behind the code. Knowing who has worked most often or most recently on a particular file aids to identify members' contribution and locate expert assistance [22]**.**

Though the significance of fine-grained information in tracing and coordinating activities is largely accepted, the granularity still needs to be tuned based on its particular application. In our case, we take an informal and qualitative approach, which do not require highly detailed information. Thus, Team Radar does not capture atomic changes, such as what character the developer has inputted, which line of code was edited, or any changes to the abstract syntax tree [17]. It simply captures some basic events in local workspaces, including client logging in and out, opening and closing project, editing file, and changing working mode. Editing file refers to any write operations to artifacts, because usually developers are not interested in others' read-only activities.

3.4   Visualization

Fig. 2 illustrates our animated visualization. Team Radar adopts a similar tree structure used by Gource [3] to present the structure of a project. The tree is dynamically generated by a force-directed layout algorithm [10]. Non-leaf nodes represent directories and are connected to the tree by edges. Leaf nodes denote files colored by their types. Each online developer is shown as an icon. When a developer is making changes to a file, his/her icon flies close to the corresponding tree node and indicates the artifact he/she is working on. When an icon moves, its afterimage stays, and a light trail shows its track. The tag beside developer's icon shows the developer's current working mode. All local events are stored in the central repository as event scripts, which drive the animation and allow user to retrieve and replay.

3.5   Metaphors

We believe that metaphor is a key factor to successful software visualization. In order to create a virtual environment that promotes the user's perception and engagement, as well as to increase information density, Team Radar adopts two metaphors in its visualization based on afterimage technique.

- *Afterimage*, or visual aftereffect, is an optical illusion that refers to an image continuing to stay in one's vision after the original image is removed. Neural biologists now generally agree that aftereffects are not mere by-products of "fatiguing neurons", but reflect neural strategies for optimizing perception [23]. There is also evidence that afterimage stimulates eyes to track motion smoothly [13]. Afterimage is a critical technique to implement our metaphors. We argue that afterimage technique, which embodies past and present information in our visualization, helps to stimulate the user's interests and engagement.

- *Radar* is an important component of battlefield awareness, a similar problem to workspace awareness, which refers to knowledge of everything occurring on the battlefield [5]. On a typical radar screen, positions of targets are displayed as moving blips, typically with light trails showing their courses and directions. Similarly, Team Radar alerts developers where others were and are working on. We use the radar metaphor to create a notion that monitoring software team is just like observing a

radar screen. In Team Radar, the tree layout mimics the polar coordinates of a radar system, icons simulate the blips of radar targets, and more interestingly, when an icon moves, its light trail shows the afterimage of the course.

• ***Memory*** metaphor refers to a common-sense that the older the memory is, the vaguer the image appears in the mind [1]. As mentioned above, when the icon flies to a new position, the afterimage of the icon and the light trail remains on the screen and blurs out through time, mimicking a passing memory. The afterimage eventually disappears, and how long this process takes is configurable, depending on how much past information the user intends to observe. Memory metaphor produces an illusory environment that allows users to traverse between past and present.
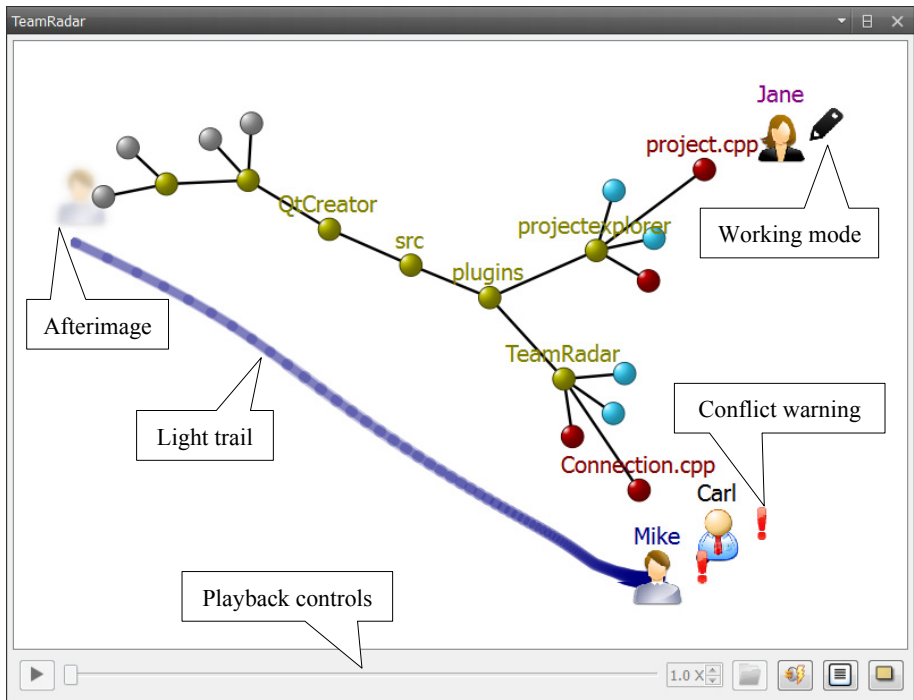
Fig. 2. Team Radar visualization. The developer's icon represents the location of a developer. Afterimages and light trials show the path the developer has gone through. Tags show the working mode of a developer. Conflict warning indicates a potential conflict when multiple developers are working on the same artifact.

3.6   Visualization Implementation

There are some challenges we are facing when implementing Team Radar. Major issues we concern are performance, scalability, and privacy.

The performance concern stems from the layout algorithm we choose. Though aesthetically appealing and flexible, the classic force-directed layout algorithm does not scale well, with the worse running time of $O(|V|^3)$, $|V|$ being the number of

vertices [10]. In our application, however, since the graph is a hierarchical tree, we utilize the local nature of the sub-trees and develop a simplified multi-scale force-directed layout algorithm [10], which takes into account only siblings in the same sub-tree and ancestors when relocating a node. Furthermore, Team Radar can cache the layout of the tree and load it faster the next time, which means the layout delay only bothers the user for the first time he joins the project. The following is the pseudo code of the improved layout algorithm.

```
force-directed-layout() {
  while(!converged()) {
    Queue queue;
    queue.enqueue(root);
    while(!queue.isEmpty()) {
      target = queue.dequeue();
      for all nodes that are ancestors of target) {
        pull(target, node);
        push(target, node);
      }
      for(all nodes that are siblings of target) {
        push(target, node);
      }
      queue.enqueue(target.getChildren());
    }
  }
}
```

The most effective measure we take to handle performance issue is *along-the-path-expansion*. Programmers' behavior also exhibits certain local nature [14]: no matter how the project scales, one programmer usually works on a small subset of the artifacts. Therefore, there is no need to expand the whole tree. Initially, Team Radar only loads the root of the tree. When a user opens a file, Team Rader will automatically expand the nodes in the path from the root to the file, and keep other nodes folded.

The scalability of a visualization is often affected by excessive information. Along-the-path-expansion could significantly improve the scalability of the system by showing a minimal subset of the nodes. Labeling is another factor to the viewability of our visualization. Displaying all the names of the nodes would overwhelm the screen, as some of the names could be very long. Team Radar only shows the labels of the nodes in the path from the root to the file currently being edited.

Developers can protect their privacy using two types of event filters: *incoming filter* and *outgoing filter*. The incoming filter defines what kind of events and whose events will be received, which helps the user to concentrate on his interested events and coworkers. The outgoing filter defines what kind of events will be broadcasted. Developers can make an agreement on the configuration of filters based on their organizational cultures. Fig. 3 shows the interface of Qt Creator with the Team Radar plug-in. Fig.4 is a screenshot of Team Radar server that logs all the traffic.
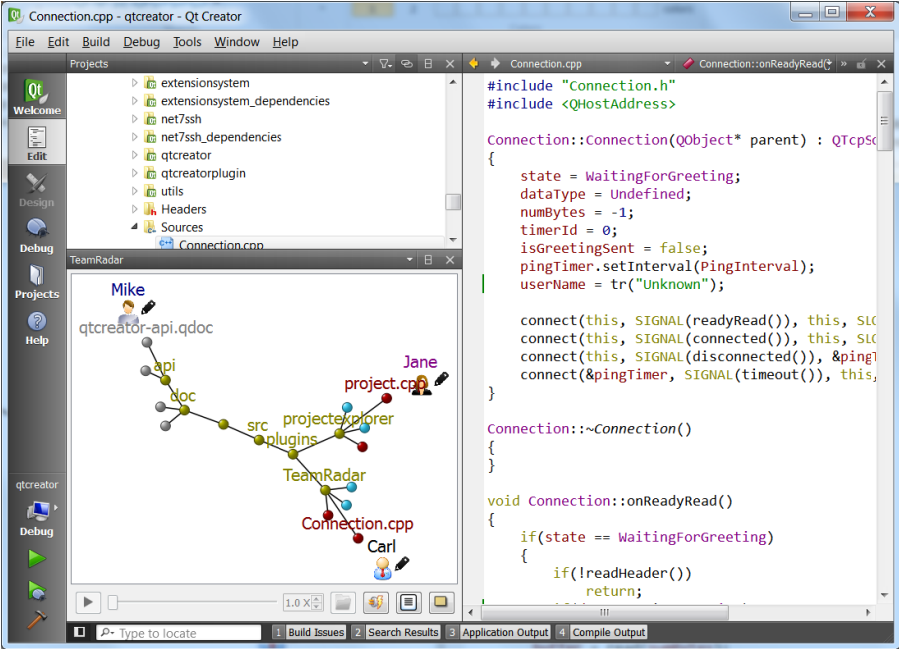
Fig. 3. Qt Creator with the Team Radar plug-in shown as an embedded window at the left-bottom corner
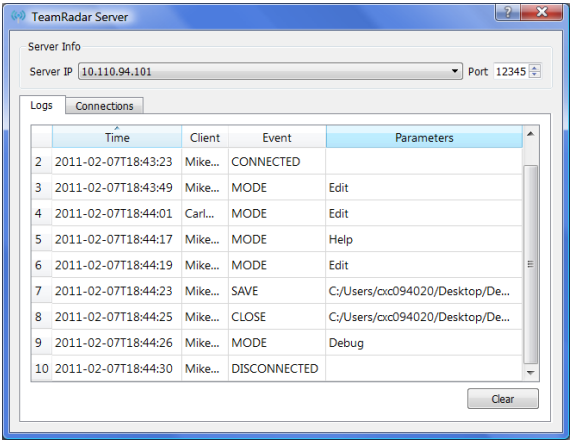

Fig. 4. A screenshot of Team Radar server

4   Conclusions

This paper has reported our ongoing work to promote team awareness and stimulate collaboration in the context of distributed software development. A prototype implementation has been built and tested. The novelty of our approach is that with

afterimage technique and radar metaphor, our visualization integrates both past and present information at the same time, which we believe would achieve a better balance of the tradeoff between providing more information and avoiding information overflow. As our future work, an experiment will be conducted to further evaluate the effectiveness of the approach.

References

1. Atkinson, R.C., Shiffrin, R.M.: Human Memory: A Proposed System and Its Control Processes. In: Spence, K.W., Spence, J.T. (eds.) The Psychology of Learning and Motivation: Advances in Research and Theory, vol. 2, pp. 89–195. Academic Press, New York (1968)
2. Begel, A., Khoo, Y.P., Zimmermann, T.: Codebook: Discovering and Exploiting Relationships in Software Repositories. In: 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 125–134. ACM, New York (2010)
3. Caudwell, A.H.: Gource: Visualizing Software Version Control History. In: ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, pp. 73–74. ACM, New York (2010)
4. Dourish, P., Bellotti, V.: Awareness and Coordination in Shared Workspaces. In: 1992 ACM Conference on Computer-Supported Cooperative Work, pp. 107–114. ACM, New York (1992)
5. Fennell, M.T., Wishner, R.P.: Battlefield awareness via Synergistic SAR and MTI Exploitation. IEEE Aerospace and Electronic Systems Magazine 13, 39–43 (1998)
6. Fitzpatrick, G., Marshall, P., Phillips, A.: CVS Integration with Notification and Chat: Lightweight Software Team Collaboration. In: 20th Anniversary Conference on Computer Supported Cooperative Work, pp. 49–58. ACM, New York (2006)
7. Froehlich, J., Dourish, P.: Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In: 26th International Conference on Software Engineering, pp. 387–396. IEEE Computer Society, Washington (2004)
8. Grinter, R.E.: Using a Configuration Management Tool to Coordinate Software Development. In: Conference on Organizational Computing Systems, pp. 168–177. ACM, New York (1995)
9. Gutwin, C.A.: Workspace Awareness in Real-Time Distributed Groupware, p. 250. University of Calgary (1998)
10. Hadany, R., Harel, D.: A Multi-Scale Algorithm for Drawing Graphs Nicely. In: Widmayer, P., Neyer, G., Eidenbenz, S. (eds.) WG 1999. LNCS, vol. 1665, pp. 262–277. Springer, Heidelberg (1999)
11. Hattori, L.: Enhancing Collaboration of Multi-developer Projects with Synchronous Changes. In: 32nd ACM/IEEE International Conference on Software Engineering, Vol.2, pp. 377–380. ACM, New York (2010)
12. Herbsleb, J.D., Mockus, A., Finholt, T.A., Grinter, R.E.: Distance, Dependencies, and Delay in a Global Collaboration. In: 2000 ACM Conference on Computer Supported Cooperative Work, pp. 319–328. ACM, New York (2000)
13. Heywood, S., Churcher, J.: Eye Movements and the Afterimage—I. Tracking the Afterimage. Vision Research 11, 1163–1168 (1971)
14. Kersten, M., Murphy, G.C.: Using Task Context to Improve Programmer Productivity. In: 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1–11. ACM, New York (2006)

15. Lanza, M., Hattori, L., Guzzi, A.: Supporting Collaboration Awareness with Real-time Visualization of Development Activity. In: 14th IEEE European Conference on Software Maintenance and Reengineering, pp. 207–216. IEEE Computer Society, Los Alamitos (2010)
16. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining Mental Models: A Study of Developer Work Habits. In: 28th International Conference on Software Engineering, pp. 492–501. ACM, New York (2006)
17. Neamtiu, I., Foster, J.S., Hicks, M.: Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In: 2005 International Workshop on Mining Software Repositories, pp. 1–5. ACM, New York (2005)
18. Qt Creator, `http://qt.nokia.com/products/developer-tools/`
19. Sandom, C.: Success and Failure: Human as Hero – Human as Hazard. In: 12th Australian Workshop on Safety Critical Systems and Software and Safety-related Programmable Systems, pp. 79–87. Australian Computer Society, Inc., Darlinghurst (2007)
20. Sarma, A., Noroozi, Z., van der Hoek, A.: Palantir: Raising Awareness among Configuration Management Workspaces. In: 25th International Conference on Software Engineering, pp. 444–454. IEEE Computer Society, Washington, DC (2003)
21. Sarma, A., van der Hoek, A.: Palantír: Coordinating Distributed Workspaces. In: 26th Annual International Computer Software and Applications Conference, pp. 1093–1097. IEEE Computer Society, Washington, DC (2002)
22. Schneider, K.A., Gutwin, C., Penner, R., Paquette, D.: Mining A Software Developer's Local Interaction History. In: IEE Seminar Digests 2004, pp. 106–110 (2004)
23. Thompson, P., Burr, D.: Visual Aftereffects. Current Biology 19, 11–14 (2009)

Model-Driven Test Code Generation

Beatriz Pérez Lamancha[1], Pedro Reales[2], Macario Polo[2], and Danilo Caivano[3]

[1] University of Republic, Montevideo, Uruguay
bperez@fing.edu.uy
[2] University of Castilla-La Mancha, Ciudad Real, Spain
{pedro.reales,macario.polo}@uclm.es
[3] University of Study, Bari, Italy
caivano@di.uniba.it

Abstract. Model-driven Testing (MDT) refers a model-based testing that follows Model Driven Engineering paradigm, i.e., the test cases are automated generated using models extracted from software artifacts through model transformations. In previous work, we developed a model to model transformation that takes as input UML 2.0 sequence diagrams, and automatically derive test cases scenarios that conforms the UML Testing Profile. In this work, these test case scenarios are automatically transformed using model to text transformation. This transformation, which can be applied to obtain test cases in a variety of programming languages, is implemented with MOFScript, which is also an OMG standard.

1 Introduction

Currently, new technologies, new tools and new development paradigms exist that help to reduce software development time. Increasingly, software development models are being used to a greater or lesser degree.

These models can be used for requirements elicitation, to achieve a common understanding with stakeholders or to build and share the architecture solution. Model-Driven Engineering (MDE) considers models for software development, maintenance and evolution through model transformation [1]. Testing must support software development, reducing testing time but ensuring the quality of the product generated. Model-based testing (MBT) provides techniques for the automatic generation of test cases using models extracted from software artifacts (Dalal et al., 1999). Several approaches exist for model-based testing [2,3]. Nonetheless, adoption of model-based testing by the industry remains low and signs of the anticipated research breakthrough are weak [4]. In this work, we use the term model-driven testing to refer to a model-based testing approach that follows the MDE paradigm, i.e., using model transformations.

In previous work [5,6], we defined an automated model-driven testing framework. This framework uses two types of transformations, the first of which is model-to-model transformation to generate test models from design models. This transformation takes UML 2.0 models as input and through QVT, produces UML Testing Profile models (this can be consulted in [6]). The second type of transformations is test model to test code transformation, which is the main contribution of this paper. Figure 1 describes
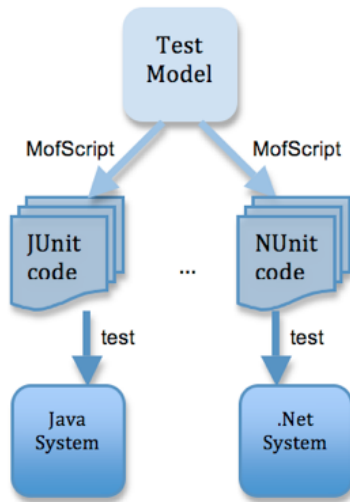
Fig. 1. Test model to test code transformations

the transformation from test model to test code. The transformation is developed using the MofScript tool[1], which implements the OMG's MOF model-to-text transformation [7]. In this work, the transformation generates JUnit code , which makes it possible to automate the coding of Java test cases and their management. It is also possible to generate other testing code, for example, NUnit to test .Net systems.

Once the test code is obtained by the transformation, it can be compiled and executed to test the system under test (SUT) and to obtain the test case verdict, i.e., whether it fails or passes. Section 2 presents the metamodels and standards used in this paper. Section 3 describes the approach for model-driven testing and presents the automated testing framework. Section 4 summarizes the model to model transformations defined in the framework. Section 5 describes transformations from test models to test code using MofScript in detail. Section 6 summarizes the work related to our approach. Finally, Section 7 presents conclusions and future work.

2   Metamodels and Standards

One of the central parts of MDE is model transformation, defined as the process of converting one model to another model of the same system [8]. Even with the source code, programs are expressed in a programming language; if we make the correspondence between a grammar and a metamodel explicit, programs may be converted into equivalent MDA-models [9]. A transformation requires: (i) source and target models, (ii) source and target metamodels and (iii) the definition of the transformation [8]. In this work the metamodel used is the UML Testing Profile.

UML 2.0 Testing Profile (UML-TP) [10] defines a language for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of test systems. It
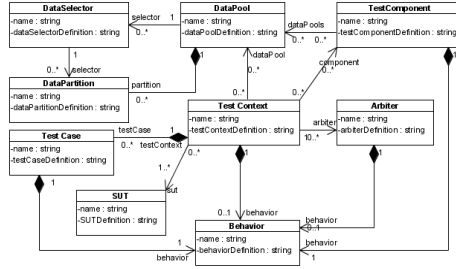
---

[1]   http://www.eclipse.org/gmt/mofscript/

Fig. 2. Metamodel

extends UML 2.0 with specific concepts for testing, grouping them in test architecture, test data, test behavior and test time. Figure 2 shows an excerpt of the UML-TP metamodel. The test architecture in UML-TP is the set of concepts to specify the structural aspects of a test situation. It includes the TestContext, which contains the test cases (as operations) and whose composite structure defines the test configuration. The test behavior specifies the actions and evaluations necessary to evaluate the test objective, which describes what should be tested. The TestCase specifies one case to test the system, including what to test it with, the required input, result and initial conditions. It is a complete technical specification of how a set of TestComponents interacts with a System Under Test (SUT) to realize a TestObjective and returns a Verdict value [10].

We use two transformations: **for model to model transformation (M2M)** we selected the OMG's Queries, Views and Transformations (QVT) standard [11]. The QVT standard describes three languages for transformations: Relations, Operational and Core. Of these, we used the Relations language, where each relation specifies how an element (or set of elements) from the source models is transformed into an element (or set of elements) of the target model. The Operational language can be used to implement one or more Relations from a Relations specification when it is difficult to provide a purely declarative specification of how a Relation is to be populated. QVT Core is a low-level language into which the others can be translated [11]. One of the advantages of the QVT standardization is its adoption by tool vendors, which also entails the possibility of interchanging models across different platforms.

For model to code transformation (M2C), we use the MOFScript tool, an implementation of OMG's MOF Model to Text transformation language (MOF2Text) [7]. Each transformation defined with this language is composed of a *texttransformation* element. A *texttransformation* is the main element that transforms a model into text. These models are specified as inputs in the transformation. Also, a texttransformation can import other previously defined transformations. A *texttransformation* is composed of rules. A rule is basically the same as a function. Each rule performs a sequence of operations or calls to other rules in order to analyze the input models and generate the desired text. Each rule has a context type, which is a type of input metamodel. This represents the type of elements to which the rule can be applied. Also, a rule can have a return element, which can be reused in other rules and input parameters to perform the operations defined in the rule. Both the return and the input parameter have a type of input metamodel or basic type, which is defined by MOFScript language. A texttransformation

element can also have an entry point rule. This is a special type of rule called main. This rule is the first rule to be executed when the transformation is executed and has the responsibility for executing the rest of the transformation rules.

The M2C transformation in our case generates **xUnit** code. xUnit is a family of frameworks, which enable the automated testing of different elements (units) of software. Such frameworks are based on a design by Kent Beck, originally implemented for Smalltalk as SUnit [12]. Gamma and Beck ported SUnit to Java, creating JUnit[2]. From there, the framework was also ported to other languages, as NUnit [3]for .NET.

3   Model-Driven Testing Approach

Our proposal for model driven testing automatize the generation of test cases from design models using model transformations. We have defined an automated framework, based on Dai's idea [13].
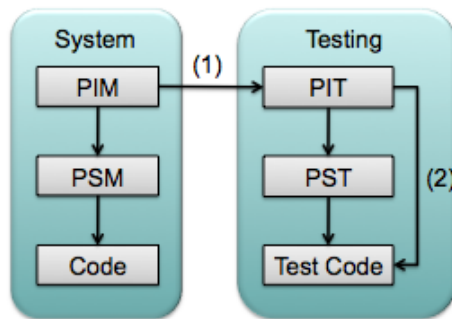
Fig. 3. Model-driven testing approach

Figure 3 shows the models involved in the framework, which is divided vertically into System models (left) and Testing models (right). For System models, the framework follow the MDA [8] levels. MDA defines three viewpoints of a system [14]: (i) the Computation Independent Model (CIM), which focuses on the context and requirements of the system without considering its structure or processing; (ii) the Platform Independent Model (PIM), which focuses on the operational capabilities of a system outside the context of a specific platform; and (iii) the Platform Specific Model (PSM), which includes details relating to the system for a specific platform.

The philosophy of MDA can be applied to test modeling. As Figure 3 shows, the same abstraction levels (PIM, PSM) can be applied to test models. The Test levels defined are [13]: (i) platform independent test model (PIT), (ii) platform specific test model (PST) and (iii) executable test code.

Furthermore, with the adequate transformations, test models can directly proceed from system designs. The arrows in Figure 3 represent transformations between models.

The main characteristics of the automated framework for model-driven testing that we have defined and implemented are [5]:

---

[2] http://www.junit.org/
[3] http://www.nunit.org/

- **Standardized.** The framework is based on Object Management Group (OMG) standards, where possible. The standards used are UML, UML Testing Profile as metamodels, and Query/View/Transformation (QVT) and MOF2Text as standardized transformation languages.
- **Model-driven Test Case Scenario Generation.** The framework generates the test cases at the functional testing level (which can be extended to other testing levels); the test case scenarios are automatically generated from design models and evolve with the product until the test code generation. Design models represent the system behavior using UML sequence diagrams.
- **Framework Implementation using Existing Tools.** No tools have been developed to support the framework: existing market tools that conform to the standards can be used. The requisite is that the modeling tool can be integrated with the tools that produce the transformations.
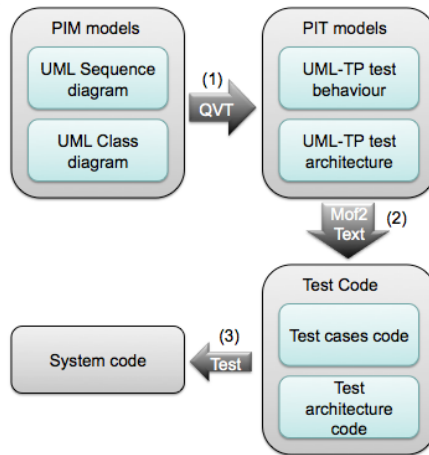
**Fig. 4.** Metamodels involved in the testing framework

Figure 4 shows the UML diagrams used in the framework. For each functionality represented as a sequence diagram at PIM level, the test case is automatically generated using QVT (arrow 1). The transformation generates the test case behavior as another sequence diagram and a class diagram representing the test architecture. Both models conform to the UML Testing Profile (UML-TP). Earlier work [6,5], presented this transformation, summarized in Section 4.

In this paper, the transformation from test models to test code is described. This transformation corresponds to arrow (2) in Figure 3 and Figure 4. With this transformation the entire cycle is closed, and the framework is completed. As result, an executable test code is generated from a test model, which in turn proceeds from the design model. For the transformation in arrow (2), test models represented using UML-TP are the input, and the test code is the output. This test code can be written according to several testing frameworks (for example JUnit, the unit testing framework for Java). This transformation is done using MOF Model-to-Text [7]. Once the test code is obtained, it can be compiled and possibly executed. With this executable test code, the system can be tested (arrow 3 in Figure 4).

4   Test Model Generation

This section explains how the test cases can be derived from sequence diagrams at functional test level, corresponding to arrow 2 in Figure 4. A UML Sequence diagram is an Interaction diagram, focused on the message interchange between lifelines. A sequence diagram describes sequences of events. Events are points on the lifeline, such as the sending of a message or the reception of a message [15]. A sequence diagram can be used to show the system behavior for a use case scenario in a design model as well as to show the behavior of a test case in a test model.
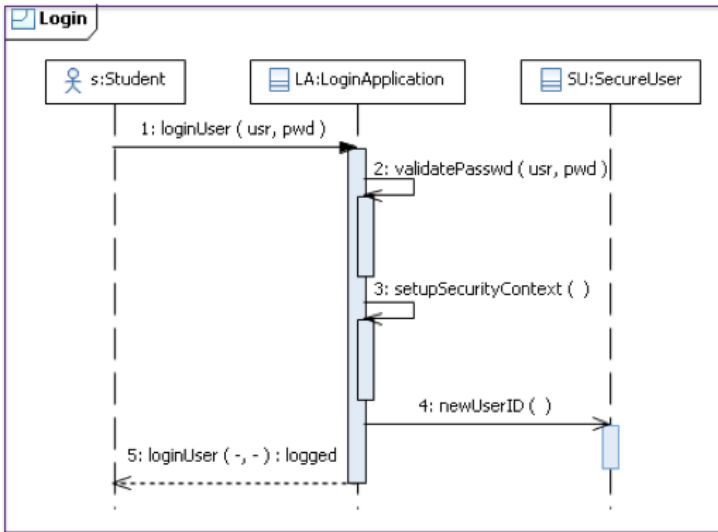

Fig. 5. UML sequence diagram for "Login"

Figure 5 shows the main scenario of the "Login" use case, where a user gives his/her user name and password and the system checks whether both parameters are valid; if they are, the system creates a new session for that user. To generate the test case for a sequence diagram, from a functional testing point of view, the system must be considered as a black box and the stimulus from the actor to the system must be simulated and vice versa. Using the UML-TP, actors are represented with TestComponents, whilst the System is represented with the SUT.

In our proposal, each message between the actor and the SUT must be tested. For this, the following steps in the test case behavior are generated:

- Obtaining the test data: To execute the test case, the required test data is stored in the DataPool. The TestComponent asks for the test data using the DataSelector operation in the DataPool.
- Executing the test case in the SUT: The TestComponent simulates the actor and stimulates the SUT. The TestComponent calls the SUT functionality to be tested: i.e., TestComponent calls the message to test in the SUT.

  – Obtaining the test case verdict: The TestComponent is responsible for checking
    whether the value returned for the SUT is correct, and uses the Validation Action
    for that.
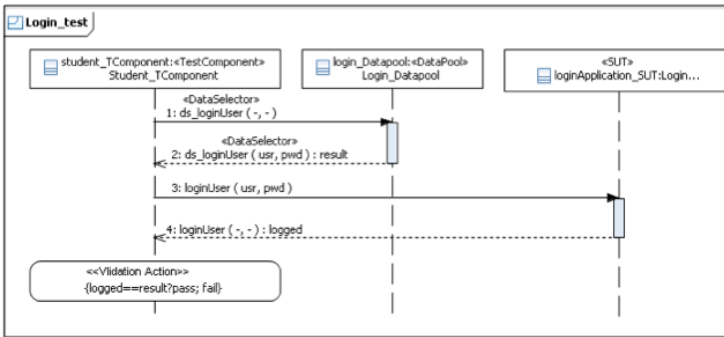
Fig. 6. Test case automated generated using QVT transformation from Login sequence diagram

Figure 6 shows the test case generated to test the functionality of Figure 5.

The TestComponent (Student_TComponent) simulates the Student actor in Figure
5. It obtains the test data necessary from the DataPool, executes the operations of the
system, and finally uses a ValidationAction to check the correct running of the system.
The first message in Figure 6 calls the *loginUser(uid,psw):Boolean*. To test this, first,
the arguments are taken from the DataPool using a DataSelector for each argument;
the DataPool retrieves the user (uid), password (pwd) and the expected result (result).
The TestComponent executes the loginUser method in the SUT (message labelled 3 in
Figure 6), and the return from the SUT is the real result (logged). Finally, the Validation
Action is responsible for the test case verdict: the test case passes if the expected result
is equal to the actual result; otherwise, it fails.

Figure 7 shows the resulting test architecture derived for this example, which con-
forms to the UML-TP metamodel. Since the UML-TP is a UML Profile, the classes
defined in the test architecture are stereotyped.

The main concepts generated are:

  – Login_TestContext: Stereotyped as <<TestContext>>, includes the operation Lo-
    gin_test for executing the test.
  – Login_DataPool: Stereotyped as <<DataPool>> contains the test data. Operations
    in this class are stereotyped as <<DataSelector>> and will be used in the tests to
    obtain the test data. Includes the operation DataSelector ds_loginUser.
  – Student_TComponent: Stereotyped as <<testComponent>> is responsible for initi-
    ating the test case and interchanging events with the SUT to test the functionality.

More information about the semantic of the transformations from design to test models
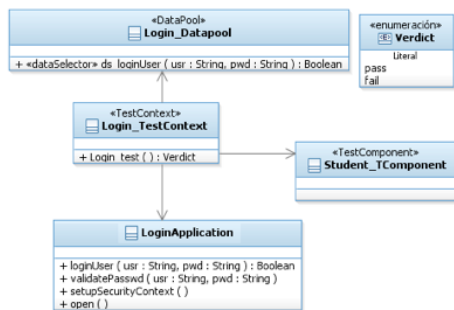and about how QVT transformations were developed can be consulted in [6].

Fig. 7. Test architecture generated

UML TP artefact	Junit element	Semantic
<<TestContext>> uml.Class	Class that extends TestCase	A test context is a test container. This test container is represented in JUnit as a class that extends TestCase class.
<<TestCase>> uml.Interaction	Test method of the TestCase class	Each interaction diagram that represents a test case is translated into a test case method of a test suite (a class that extends TestCase class)
<<TestComponent>> uml.Class	none	There is no TestComponent per se in JUnit.
<<SUT>> uml.LifeLine	Field of the class that extends TestCase	A sut is a part of the system under test. In JUnit the system is referenced as a field in the test suite class.
<<DataPool>> uml.Class	Class	A data pool contais the data for the test. In JUnit it is represented with a simple class with *get* methods that is used by the test suite in order to get test data.
<<DataSelector>> uml.Operation	A method of the class that represent the data pool	A data selector represents an operation to get test data form the DataPool class.

Fig. 8. Transformation rules semantic for test architecture (adopted from UML-TP)

5   Transformations from Models to Code

This section presents the main contribution of the paper: transformations from test models to test code, which corresponds to the arrow labelled 2 in Figure 5.

Our approach applies the idea of MDA development to testing. MDA separates business complexity from the implementation details, by defining several software models at different abstraction levels [16,17].

Once the test cases and the test architecture are obtained, the next step is to obtain the test code to test the system. Table 8 shows how the test model is transformed to test code. We use JUnit test code to exemplify the transformation. The transformation

UML TP artefact	Junit element	Semantic
Each uml.Message to a <<SUT>> lifeline	A call to the system to execute a method	A message to the sut represents the execution of a functionality. In Junit, it is represented by the execution of a method.
Each uml.Message to the DataPool	A set of call to the class that represents a dataPool	A message to the data pool is executed to obtain the test data. In JUnit, it is represented by the execution of different method of the data pool, one for each required test value and one for the expected result.
<<ValidationAction>> uml.StateInvariant	Junit assertion	The validation actions represents post conditions of the test. These post conditions are represented as Assertions in Junit.

Fig. 9. Transformation rules semantic for test behavior

takes UML-TP models as input and generates JUnit Code as output. Table 8 shows the semantic of the transformation rules to generate the test code. The first column shows the UML-TP artifact, the second shows the JUnit element generated and the third describes the semantic of the transformation. UML-TP specification describes the transformations to JUnit for the test architecture. However, transformation rules for behavioral test cases are defined by us, taking into account the characteristics of the sequence diagrams generated (see Table 9).

5.1 MOFScript Transformations

Two MofScript transformations have been implemented to perform the transformations in Table 8 and Table 9. These MofScript transformations are TextContextMapping and DataPoolMapping. TestContextMapping transformation is responsible for generating the JUnit code that contains the test cases, and the body of the test cases. This transformation has a set of rules that can be split into two:

1. rules to create the architecture (the test suite class and the test case methods) and
2. rules to create the body of the test cases (in the next section).

The first kind of rule analyzes the packages, classes and sequence diagrams that represent test cases and create a specialization of TestSuite class for each class stereotyped as <<TestContext>>. Parameters and methods in the model are in turn translated into Java parameters and methods (excepting the operations which are realized by sequence diagrams stereotyped as <<TestCases>>).

The second kind of rule creates the test cases. They analyze the sequence diagrams stereotyped as <<TestCase>>. Each time an operation of a test context is carried out, a new method is created in a test suite (previously generated from the text context). The method name starts with the word "test" and it has not returned value to the parameters. Then, the rules generate the body of the method analyzing the sequence of messages inside the sequence diagram. The transformations performed by this kind of rule are described in details in the following section. The DataPoolMapping transformation is responsible for creating the Java classes that represent DataPools for the tests. This

```
 1: uml.Interaction::mapAsAMethod(name:String){
 2:   <%public void test%> name.firstToUpper() <%(){%>
 3:   var lista:List
 4:   var retorno:uml.Parameter
 5:   self.fragment->forEach(boe:uml.BehaviorExecutionSpecification){
 6:       var op:uml.Operation = boe.start.event.operation
 7:       var c:uml.Class = op.class
 8:       var ll:uml.Lifeline = boe.start.covered
 9:       var msg:uml.Message = boe.finish.message
10:     if(c.hasStereotype("DataPool")){
11:         lista.clear()
12:         op.ownedParameter->forEach(p:uml.Parameter){
13:         module::addVariableDeclaration(p)
14:         c.name.firstToUpper()<%.%> op.name+p.name.firstToUpper()<%();\n%>
15:         if(p.direction<>"return"){lista.add(p)
16:         }else{retorno = p}}
17:     }else{
18:         var retornoOP:uml.Parameter
19:         var prl:list= op.ownedParameter->select(px:uml.Parameter|px.direction = "return")
20:         if(prl.size()>0){
21:             var pr:uml.Parameter = prl.first()
22:             module::addVariableDeclaration(pr)
23:             retornoOP = pr}
24:         var lifeline:uml.Lifeline
25:         self.lifeline->forEach(ll2:uml.Lifeline){
26:             if(ll2==ll){
27:                 lifeline = ll2}}
28:         lifeline.name<%.%>op.name<%(%>
29:         while(lista.size()>0){
30:             var p:uml.Parameter = lista.first()
31:             <% %>p.name
32:             lista.remove(p)
33:             if(lista.size()>0){<%,%>}
34:         }<%);\n%>
35:         self.fragment->forEach(si:uml.StateInvariant|si.hasStereotype("validationAction")){
36:             <%\n//asertion for %>op.name<%\n%>
37:             var s:String = si.invariant.specification.body
38:             <%assertTrue(%>
39:             if(retorno.type.oclIsKindOf(uml.PrimitiveType)){
40:                 retorno.name<%==%>retornoOP.name<%);\n%>
41:             }else{
42:                 retorno.name<%.equals(%>retornoOP.name<%));\n%>
43:             }}}
44: }<%}%>}
```

Fig. 10. MofScript rule: MapAsAMethod to transform an iteration into a method

transformation is only composed of rules to create the architecture, because the body of the methods simply returns a value.

5.2   An Example of MofScrip Rule: uml:Interaction::mapAsAMethod

This section presents an example of a transformation rule using MofScritp. Rule uml:Interaction::mapAsAMethod of the transformation TextContextMapping is shown in Table 3.

This rule transforms a UML Interaction stereotyped as "TestCase" into a JUnit test method that belongs to the resulting test suite class. Basically, the rule creates the header

of the method and searches sequences of three elements (as shown in Table 3: i) a call to the DataPool, ii) a call to the SUT and iii) a state invariant, in order to create the body of the method. Statement 2 creates the method header. Then statement 5 creates a loop that goes all over the messages, searching the messages for the DataPool, SUT and the stateinvariant. When a message to the DataPool is found, it searches for the remaining calls described above.

At this point the execution of two iterations is required. The first iteration creates the calls to the DataPool and stores the required information for the next iteration. Statements 11-16 translate the message to the DataPool into a set of calls to the DataPool, one for each parameter passed by the reference. This division is required because in UML a method can have many parameters by reference but in Java the parameters are passed by value and there is only a return parameter. Another possibility would be to create a method that returns a vector in order to contain all the parameters by reference, but for simplicity's sake, we chose to create several calls. To create these calls, the auxiliary function addVariableDeclaration is used. This function creates the declaration of the variable that will contain the value retuned by the DataPool.

The second iteration creates a call to the SUT. Statements 18-34 deal with the translation of the message to the SUT into a call to the SUT. These statements can be split into two parts. The first part is composed of statements 18-23. These statements check when the call to the SUT has a return value, and in that case create a variable declaration using the addVariableDeclaration function that will contain the value returned by the SUT. Statements 24-34 compose the second part. These statements create the call to the SUT using the variables that contain the data obtained from the DataPool.

At the end of the second iteration, an assertion with the information stored in the state invariant element is generated, which is just after the message element that represents the call to the SUT. Statements 35-43 deal with translating the state invariant elements into JUnit assertions. The statements simply create an assertion and compare the expected result obtained from the DataPool with the result obtained from the SUT.

5.3 JUnit Code Generated

Once MofScript transformations are executed, the JUnit test case is obtained. Figure 11 shows the JUnit test code generated.

This test code could be compiled and executed. After this compilation, JUnit shows its execution results (Figure 12).

5.4 Model-Driven Testing Framework implementation

The implementation of the framework requires the selection of a modeling tool from those on the market, as well as the identification of the tools to perform transformations between the models and from model to code. Our selected tool was IBM Rational Software Architect (IRSA). This tool graphically represents the sequence diagrams and exports them to UML2 through XMI.

The Eclipse IDE makes it possible to use modeling tools in an integrated way, using extensions in the form of plug-ins. Eclipse plug-ins, which are used to perform

```
package testGenerated;

import java.util.Vector;
import junit.framework.TestCase;
import System.LoginApplication;

// TextContext
public class Login_TContext extends TestCase {

        private LoginApplication loginApplication;
        // Default constructor without parameters
        public Login_TContext () {
        }

        public void setUp(){
                loginApplication = new LoginApplication();
        }
        public void testLogin_test(){
                //Method calls to Datapool
                String usr = Login_Datapool.ds_loginUserUsr();
                String pwd = Login_Datapool.ds_loginUserPwd();
                boolean result = Login_Datapool.ds_loginUserResult();
                //Call to SUT
                boolean loged = loginApplication.loginUser( usr, pwd)
                //asertion for loginUser
                assertTrue(result==loged);
        }
} // End of class Login_TContext
```

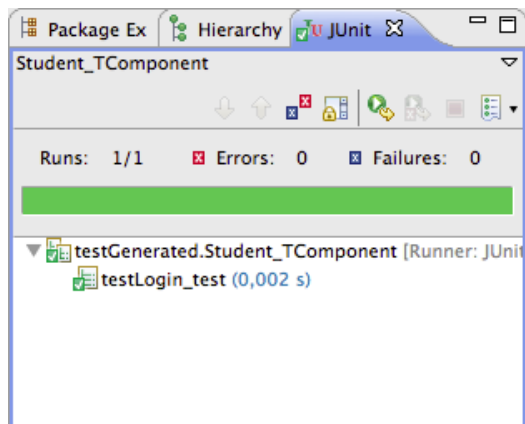**Fig. 11.** JUnit test code generated

Fig. 12. JUnit test case execution

modeling tasks, exist. The Eclipse Modeling Framework (EMF) plugin allows the development of metamodels and models: from a model specification described in XMI, it provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model. UML2 is an EMF-based implementation of the UML 2.0 OMG metamodel for the Eclipse platform. UML2 Tools is a Graphical Modeling Framework editor for manipulating UML models.

The transformation between models (arrow 1 in Figure 4) uses QVT language, which requires the tool that implements the standard. medini QVT is a plugin for eclipse that

implements OMG's QVT Relations specification in a QVT engine. We used it to develop and execute the QVT transformations [6]. The model-to-text transformations have been defined with MofScript language, and it thus requires a tool that supports this language. The MOFScript tool is a plugin for Eclipse that makes it possible to develop transformations with the language MofScript. This tool has been used to develop and perform the transformations presented in this paper. It has a code editor to define the transformations, which brings out the reserved word of the language and has autocompletion features. This tool also has a MofScript checker and an execution engine to check the syntax of the defined transformations and execute them.

6   Related Work

Many proposals for model-based testing exist [2,3], but few of them focus on automated test model generation using model transformations.

Dai [13] describes a series of ideas and concepts to derive UML-TP models from UML models, which are the basis for a future model-based testing methodology. Test models can be transformed either directly to test code or to a platform specific test design model (PST). After each transformation step, the test design model can be refined and enriched with specific test properties. However, to the best of our knowledge, this interesting proposal has no practical implementation for any tool.

Baker et al. [15] define test models using UML-TP. Transformations are done manually instead of using a transformation language.

Naslavsky et al. [18] use model transformation traceability techniques to create relationships among model-based testing artifacts during the test generation process. They adapt a model-based control flow model, which they use to generate test cases from sequence diagrams. They adapt a test hierarchy model and use it to describe a hierarchy of test support creation and persistence of relationships among these models. Although they use a sequence diagram (as does this proposal) to derive the test cases, they do not use it to describe test case behavior.

Javed et al. [19] generate unit test cases based on sequence diagrams. The sequence diagram is automatically transformed into a unit test case model, using a prototype tool based on the Tefkat transformation tool and MOFScript for model transformation. This work is closed to ours, but they don't uses the UML-TP. We generate the unit test case in two steps and they in only one. We think that use a intermediate model using UML-TP as PIT is more appropiate to follow a MDE approach.

7   Conclusions

We have presented our framework for automated model-based testing using standardized metamodels such as UML and UML-TP. In this paper the complete transformations cycle defined in the framework is implemented, obtaining executable test cases procedures in JUnit code. To obtain complete test cases we also need to define the way in that test data are generated: at this moment, both the test data and the expected result (which are required for the test oracle) are manually stored in the datapool. Our ongoing work uses UML State Machines to define the test oracle. Future work includes implementing

MOFScript transformations to generate NUnit test cases, the application of the entire framework in an industrial project and, as we have pointed out, to take advantage of state machine annotations to automatically include the oracle in the test cases.

References

1. Mens, T., Van Corp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Sciences 152, 125–142 (2006)
2. Prasanna, M., Sivanandam, S., Venkatesan, R., Sundarrajan, R.: A survey on automatic test case generation. Academic Open Internet Journal 15, 1–5 (2005)
3. Dias Neto, A., Subramanyan, R., Vieira, M., Travassos, G.: A survey on model-based testing approaches: A systematic review, pp. 31–36 (2007)
4. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: Internation Conference on Software Engineering, pp. 85–103. IEEE Computer Society (2007)
5. Perez Lamancha, B., Polo, M., Piattini, M.: An automated model-driven testing framework for model-driven development and software product lines, pp. 112–121 (2010)
6. Perez Lamancha, B., Reales Mateo, P., Garcia, I., Polo Usaola, M., Piattini, M.: Automated model-based testing using the uml testing profile and qvt. In: Workshop on Model-Driven Engineering, Verification and Validation, pp. 1–10. ACM, Denver (2009)
7. OMG: Mof model to text transformation language. Technical Report Formal/2008-01-16, OMG (2008)
8. Miller, J., Mukerji, J.: Mda guide version 1.0.1. Technical Report OMG/2003-06-01 (2003)
9. Bezivin, J.: On the unification power of models. Software and Systems Modeling 4, 171–188 (2005)
10. OMG: Uml testing profile version 1.0. Technical Report formal/05-07-07 (2005)
11. OMG: Mof query/view/transformation specification. Technical report (2007)
12. Beck, K.: Kent Beck's guide to better Smalltalk: a sorted collection. Cambridge University Press (1999)
13. Dai, Z.: Model-driven testing with uml 2.0. In: Workshop on Model Driven Architecture with an emphasis on Methodologies and Transformations, EWMDA (2004)
14. Harmon, P.: The omg's model driven architecture and bpm. Newsletter of Business Process Trends (2004)
15. Baker, P., Dai, Z., Grabowski, J., Schieferdecker, I., Haugen, O., Williams, C.: Model-Driven Testing: Using the UML Testing Profile. Springer, Heidelberg (2007)
16. Kleppe, A., Warmer, J., Bast, W.: MDA Explained; The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
17. Mellor, S., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley (2004)
18. Naslavsky, L., Ziv, H., Richardson, D.: Towawrds traceability of model-based testing artifacts. In: Workshop on Advances in Model-based Testing, pp. 105–114. ACM, London (2007)
19. Javed, A., Strooper, P., Watson, G.: Automated generation of test cases using model-driven architecture. In: International Workshop on Automation of Software Test (2007)

Comparing Goal-Oriented Approaches to Model Requirements for CSCW

Miguel A. Teruel, Elena Navarro, Víctor López-Jaquero, Francisco Montero, and Pascual González

LoUISE Research Group, I3A, University of Castilla-La Mancha, Castilla-La Mancha, Spain
{miguel,enavarro,victor,fmontero,pgonzalez}@dsi.uclm.es

Abstract. Collaborative systems are becoming increasingly important, because they enable several users to work together and carry out collaboration, communication and coordination tasks. We have to highlight that, to perform these tasks, the users have to be aware of other users' actions, usually by means of a set of awareness techniques. Usually, the specification of this set of techniques has to be done by means of Non-Functional Requirements, related to quality factors such as ease of use or helpfulness. Therefore, choosing a technique to model the requirements of this kind of systems is an important issue. In previous works, we analyzed different Requirements Engineering (RE) techniques, and we concluded that Goal-Oriented is the most promising one for modeling collaborative systems. Based on these conclusions, in this paper we compare three Goal-Oriented approaches, namely NFR framework, i* and KAOS, in order to determine which one is the most suitable to model CSCW stakeholder requirements.

Keywords: Goal-oriented, KAOS, NFR, i*, Collaborative systems, CSCW, Awareness, Requirements engineering, Non-functional requirements, Quality.

1   Introduction

A collaborative system (a.k.a. Computer Supported Cooperative Work system, CSCW system) is software whose users can perform collaboration, communication and coordination tasks. Unlike conventional single-user systems, a CSCW system has to be specified by using a special set of requirements of a non-functional nature. These requirements usually result from the users' need of being aware of the presence and activity of other remote or local users with whom they perform the above mentioned tasks, that is, the *Workspace Awareness* (WA) [1].

Workspace Awareness is "the up-to-the-moment understanding of another person's interaction within a shared workspace. Workspace awareness involves knowledge about where others are working, what they are doing now, and what they are going to do next" [1]. Gutwin et al. presented a conceptual framework to establish what information makes up workspace awareness. This information is obtained by answering the questions "who, what and, where" (see Table 1). That is, when we work with others users in a physical shared space, we know who we are working with, what they are doing, where they are working, when various events happen, and how those events happen.

Table 1. Elements of Workspace Awareness

Category	Element	Specific questions
Who	Presence	Is anyone in the workspace?
	Identity	Who is participating? Who is that?
	Authorship	Who is doing that?
What	Action	What are they doing?
	Intention	What goal is that action part of?
	Artefact	What object are they working on?
Where	Location	Where are they working?
	Gaze	Where are they looking?
	View	Where can they see?
	Reach	Where can they reach?

In this context, a proper specification of the system, identifying clearly the requirements of the system-to-be, specially the awareness requirements, is one of the first steps. The awareness requirements can be considered non-functional requirements (NFR) or extra-functional requirements (EFR), because they are usually constraints regarding quality (e.g. functionality, usability) [2]. However, the specification of this kind of requirements is not a trivial issue, because of the high number and diversity of requirements they are related to, and their high impact in terms of the final architecture of the system. Therefore, the proper selection of the requirement specification technique becomes a challenging and important decision.

In a previous work [3] it was analyzed which technique, Goal-Oriented (GO), Use Cases or Viewpoints is more appropriate to specify the requirements of collaborative systems and it was determined that GO provides more facilities for this kind of systems. In this paper, we study the applicability of three Goal Oriented (GO) approaches (NFR Framework [4], *i** Framework [5] and KAOS Methodology [6]) for the specification of collaborative systems, paying special attention to the awareness requirements. In order to carry out this study, the awareness requirements of a real system (Google Docs [7]) were specified. After modeling the system, an empirical analysis was conducted in order to compare these different techniques goal-oriented techniques.

This paper is structured as follows. After this introduction, in Section 2, the selection of GO techniques for modeling this kind of systems is justified. In Section 3, three GO approaches applicable to awareness requirements for collaborative systems are analyzed. In Section 4, an example of a widely known collaborative system is presented: Google Docs. In Section 5, an empirical evaluation of the previous techniques for modeling awareness requirements in Google Docs is presented. Finally, some conclusions and future works round up this work.

2   Related Works

This paper is a follow-up of the work presented in [3], where we analyzed different Requirement Engineering techniques applied to collaborative systems. The main result of this evaluation was that the most appropriate technique for this kind of systems is Goal Oriented (GO). Nevertheless, in [3] the evaluation did not focus on a specific GO proposal.

In the context of Requirements Engineering, the GO approach [6] has proven its usefulness for eliciting and defining requirements. More traditional techniques, such as Use Cases [8], only focus on establishing the features (i.e. activities and entities) that the system-to-be should support. Nevertheless, GO proposals focus on why systems are being constructed by providing the motivation and rationale to justify the software requirements specification. They are not only useful for analyzing goals, but also for elaborating and refining them.

A GO model can be specified in a variety of formats, by using a more or less formally defined notation. These notations can be informal, semi-informal or formal approaches. Informal approaches generally use natural language to specify goals; semi-formal use mostly box and arrow diagrams; finally, in formal approaches goals are expressed as logical assertions in some formal specification language [9]. No matter its formality, a goal model is built as a directed graph by means of a refinement of the systems goals. This refinement lasts until goals have enough granularity and detail so as to be assigned to an agent (software or environment) so that they are verifiable within the system-to-be. This refinement process is performed by using AND/OR/XOR refinement relationships.

There are a wide number of proposals ranging from elicitation to validation activities in the RE process (see [9] for an exhaustive survey). However, some concepts are common to all of them:

• Goal describes why a system is being developed, or has been developed, from the point of view of the business, organization or the system itself. In order to specify it, both functional goals, i.e., expected services of the system, and softgoals related to the quality of service, constraints on the design, etc should be determined.

• Agent is any active component, either from the system itself or from the environment, whose cooperation is needed to define the operationalization of a goal, that is, how the goal is going to be provided by the system-to-be. This operationalization of the goals is exploited to maintain the traceability throughout the process of software development.

• Refinement Relationships: AND/OR/XOR relationships allow the construction of the goal model as a directed graph. These relationships are applied by means of a refinement process (from generic goals towards sub-goals) until they have enough granularity to be assigned to a specific operationalization.

It must be pointed out that one of the main advantages exhibited by this approach is that it introduces mechanisms for reasoning about the specification. It facilitates the process of evaluating designs or alternative specifications of the system-to-be [3,10]. In this work, three different GO proposals are used to model the requirements of a collaborative system: Google Docs. This system will allow us to evaluate which proposal is the most useful to describe the requirements of the so called workspace awareness.

3   Goal-Oriented Proposals: An Analytical Background

This Section presents briefly the GO proposals, NFR, *i** and KAOS, analyzed to determine which one is the most appropriate for specifying collaborative systems.

They are used in Section 5 to describe the running example in order to perform the evaluation.

3.1   NFR Framework

This GO proposal was proposed by [4] and aims at dealing with Non-Functional Requirements (NFRs), also known as Quality Requirements. Unlike Functional Requirements, NFRs specify constraints for the system, as well as particular notions of *quality factors* a system should meet, such as, accuracy, usability, safety, performance, reliability or security. Hence, it can be stated that while functional requirements describe "what" the system will do, NFRs constraint "how" the system will accomplish the "what". As a consequence, NFRs are always linked to a Functional Requirement.

To elicit NFRs, the authors propose the use of a strategy anchored in *Language Extended Lexicon* (LEL) [11]. LEL is based on a controlled vocabulary system made up of *symbols* being each one of them an entry expressed in terms of *notions* and *behavioural* responses. A notion records the meaning of a symbol and its fundamental relationships to other entries. A behavioural response specifies the connotation of a symbol in the universe of discourse. Each symbol may also be represented by one or more aliases and will be classified as a *subject*, a *verb* or an *object*. Once the Lexicon is finished, it is enriched with NFRs by using a knowledge base, presented as catalogues, to guide the analyst to select the likely needed NFRs and their related operationalizations.

Fig. 1. Elements of the NFR Framework model

According to the NFR Framework, NFRs goals can conflict among them and must be represented as softgoals to be satisfied. Each softgoal is decomposed into sub-goals represented by a graph structure inspired by the *And/Or* trees used in problem solving. This decomposition is done by using contribution links. Contribution links can be categorized as either *or* contributions or *and* contributions. Contribution links allow one to decompose NFRs to the point that one can state that the operationalizations of the related NFR have been met. Operationalizations are decisions about the system to meet NFRs. The elements of the NFR GO model can be seen in Fig. 1.

3.2   i* Framework

The *i** Framework [5] consists in an approach for dealing with requirements in various phases of the software development process (Early and Late Requirements Analysis, Architectural and Detailed Design).

During early requirements analysis, the requirements engineer gathers and analyzes the intentions of stakeholders. These are modeled as goals which, through some form of a goal-oriented analysis, eventually lead to the functional and non-functional requirements of the system-to-be. In *i**, early requirements are assumed to

involve social actors who depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. The *i** framework includes the *strategic dependency model* for describing the network of relationships among actors, as well as the *strategic rationale model* for describing and supporting the reasoning that each actor goes through concerning its relationships with other actors. The model elements can be seen in Fig. 2.
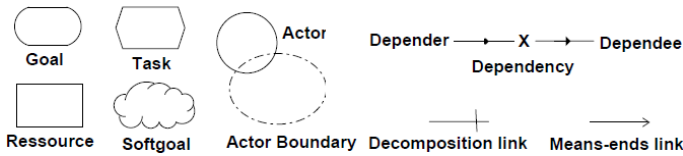

Fig. 2. Elements of the *i** Framework model

Late Requirements Analysis results in a requirements specification which describes all functional and non-functional requirements for the system-to-be. In Tropos [12], a framework for requirements-driven software development, the information system is represented as one or more actors who participate in a strategic dependency model, along with other actors from the system's operational environment. In other words, the system comes into the picture as one or more actors who contribute to the fulfillment of stakeholder's goals.

During architectural design we have to select among alternative architectural styles by using as criteria the desired qualities identified earlier in the process. The analysis involves refining these qualities, represented as softgoals, to sub-goals that are more specific and more precise and then evaluating alternative architectural styles against them.

The detailed design phase is intended to introduce additional details for each architectural component of a system. To support this phase, the authors propose to adopt existing agent communication languages and message transportation mechanisms among other concepts and tools.

3.3 KAOS Methodology

The KAOS modelling language is part of the KAOS framework [6] for eliciting, specifying, and analysing goals, requirements, scenarios, and responsibility assignments. A KAOS model entails six complementary views or sub-models (goal, obstacle, object, agent, operation and behaviour model) all of them related via traceability links [13].
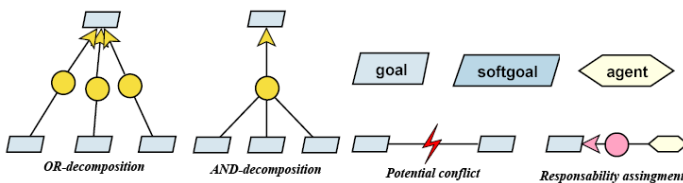

Fig. 3. Basic constructs of the KAOS framework

Fig. 3 depicts the basic constructors for documenting agents' responsibilities for goals provided by the KAOS framework. KAOS has the following elements:

- Goal: A goal describes a set of admissible system behaviors. Goals should be defined in a clear-cut manner so that one can verify whether the system satisfies a goal or not.

- Softgoal: In KAOS, softgoals are used to document preferences among alternative system behaviors. In a similar way to i*, there is no clear-cut criterion for verifying the satisfaction of a softgoal. Softgoals are hence expected to be satisfied within acceptable limits.

- While i* focuses primarily on agents within organizational structures, the agents defined in KAOS primarily relate to users and components of software-intensive systems. Therefore, an agent is defined as an active system component which has a specific role for satisfying a goal. An agent can be a human agent, a device or a software component.

Dependencies between goals are represented in the KAOS goal model by using AND/OR-decompositions and conflict links. In KAOS, goals can be assigned to agents by means of responsibility assignment links. We briefly explain these goal dependencies:

- AND/OR-decomposition: An AND-OR decomposition link relates a goal to a set of sub-goals, documenting that the goal is satisfied if all, or at least one sub-goal, is satisfied.

- Potential conflict: This link documents that satisfying one goal may prevent the satisfaction of other goal under certain conditions.

- Responsibility assignment: This link between a goal and an agent means that this agent is responsible for satisfying the goal.

4   Running Example

As running example to assess how these GO approaches perform for collaborative system, Google Docs [7] has been used from now on in this paper. Google Docs is a free, Web-based word processor, spreadsheet, presentation and form editor whose data storage service is provided by Google. Google Docs serves as a collaborative tool for editing documents so that they can be shared, opened, and edited by multiple users at the same time. This system was selected for our analysis because it is widely-known and it features a clear collaborative focus as its main goal.

As a starting point for our evaluation of the requirements techniques, we identified those design solutions for awareness requirements in Google Docs from the set of techniques proposed by Gutwin [14]. These techniques, which are commented in the following Subsections, can be found also as patterns for user collaboration in [15].
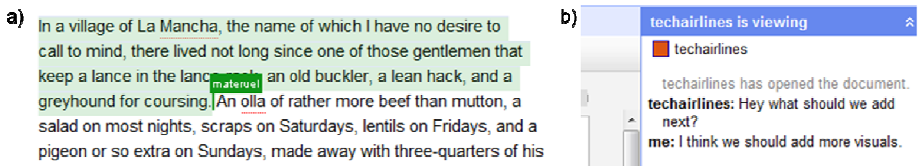
4.1   Remote Cursors

This technique, based in Gutwin's *telepointers* [14], allows us to be aware of the other user's cursor position and whether they have selected a text fragment or not (see

Fig. 4). Thus, when a remote user is writing other users can notice it in real-time. Close to the cursor the user's nickname appears overlapped with the text. In addition, if the user selects some text, it is highlighted by marking it with the user's color.

## 4.2   Participant List and Chat

Google Docs does not implement Gutwin's *avatar* [14] technique itself. Instead it shows a list of participants that are editing simultaneously the same document (see Fig. 4). By using this list, users can communicate with each other by using a chat, which can be shown or hidden at any time. In addition, by using this chat view, users can notice the color assigned to each one of their collaborators.

Fig. 4. Remote cursor and remotely selected text fragment (a) and two users chatting through the participant list (b)

4.3   Revision History

The techniques identified by Gutwin *expressing information about authorship / about the past* [14] are used to make available to the users the history of changes carried out. They have been implemented by Google Docs by using a *revision history*. It allows the system to keep track of all the changes made by the users to the different types of documents being edited (see Fig. 5). This revision history provides a mean for users to review the changes made to the documents. In this revision history the changes made by each user are denoted by using different colors. In addition, if the change made is a deletion, then the text will be also in strikethrough style. This functionality can be activated or deactivated at anytime. This revision history has two levels of detail, depending on the amount of shown information. The user may switch between these two levels of detail at anytime.
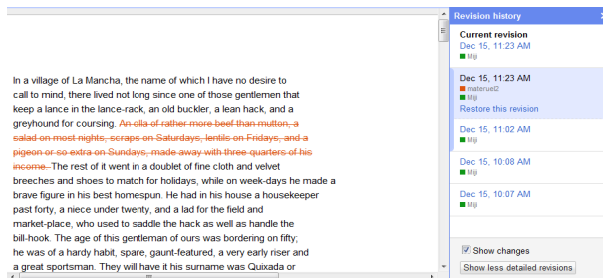
Fig. 5. Revision story showing text elimination

5   Empirical Evaluation

To evaluate the different GO approaches mentioned in Section 3, each one of the above mentioned awareness features is modeled in the following by using the different techniques. First, we have to distinguish what Google Docs characteristics can be modeled by using functional or non-functional requirements. The *telepointer* and *avatar* techniques result in NFRs because they contribute to increase some operability, such as ease of use and helpfulness. Nevertheless, the third characteristic (*Expressing information about authorship / about the past*), despite contributing positively to the above mentioned quality features, it should be considered functional, due to the historical information storage and the rollback function. In addition, we have also associated the awareness functionalities both with the three characteristics of the collaborative systems (collaboration, communication and coordination) and, with the characteristics of the ISO/IEC 25010 [16]. This standard has been used to organize properly the specification of the system following the recommendations of Moreira et al. [17]. Next, the evaluation is presented following the chronological order it was carried out. First, in Section 5.1 it is described how the case study was modeled by applying the three approaches. Second, in Section 5.2, the results of the evaluation are presented.

5.1   Modeling the Running Example

After analyzing the characteristics of Google docs described in Section 4, and according to Gutwin's framework for collaborative systems, we have specified the systems' FRs (Table 2 illustrates a partial description of the system). Next, as can be observed in Table 3, each awareness functionality feature detected in the system has been related to some quality factors in the SQuaRE standard, in order to identify the NFRs of Google Docs. For the sake of clarity, and understanding of the evaluation, only some requirements of Google Docs are described.

Table 2. Relation between awareness elements and FRs

Category	Element	Functional Requirement
Who	Presence	Know who is participating
What	Action	See other user's actions
Where	Location	
Who	Authorship	Keep the changes' authorship
When	Event history	

Table 3. Relation between quality factors and awareness functionalities

Quality Factor	Awareness Functionality
Functional Suitability	Revision History, Telepointers, Participant List
Reliability	Revision History
Performance Efficiency	Telepointers
Operability	Telepointers, Participant List
Security	Revision History

NFR Framework. In this approach, the SQuaRE quality factors have been modelled by using softgoals. Nevertheless, the SQuaRE standard was used instead of the NFR collections proposed by [4] definition to create the NFR hierarchy. Thus, it can be observed the impact that the quality sub-characteristic has on main characteristic by means of contribution links. In the same way, each characteristic contributes to achieve the software product quality (see Fig.8).

The problem here is that we are not able to represent the Functional Requirements (because this model aims only at non-functional ones), therefore the three general tasks of collaborative systems (collaboration, communication and coordination) cannot be defined. This lack of expressiveness led us to have an incomplete representation of system's requirements, so that we have to use additional models or extend this framework.

i* Framework. In order to carry out the specification of Google Docs, the *i** notation was used. Using this notation, we specified each one of the SQuaRE quality factors previously identified in Table 3, as root softgoals of the system as shown in Fig. 9. These softgoals were refined into other softgoals by selecting those SQuaRE quality factors more appropriate for the system. Each one of the awareness functionalities were specified as resources provided by the system that contribute positively to satisfy some of the softgoals, that is, some quality factors. However, it can be noticed that also some of them contribute negatively because the constraints they impose. This is the case of remote cursors, because they increase the resource utilization. Moreover, the ease of use depends, among other factors, on the user's experience with this kind of systems. In addition, the three FR identified in Table 3 have been specified as goals of the system that have dependency relationships with the resources. It has been also specified how the awareness techniques contribute positively to the functional aspects of collaborative systems specified as tasks in the goal model.
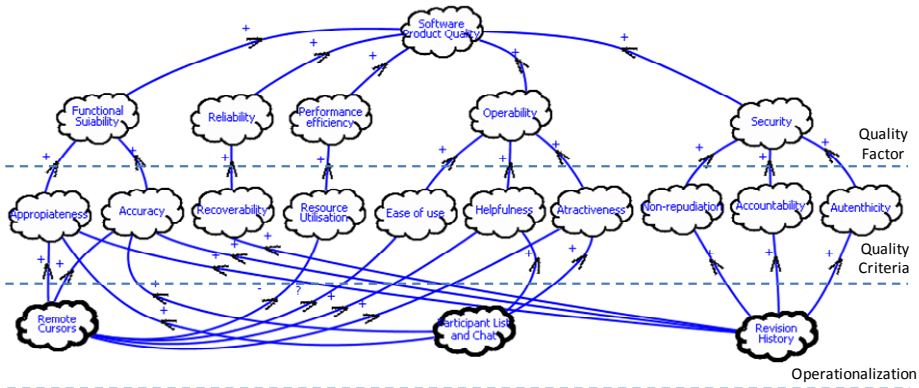

Fig. 6. NFR Goal-Oriented model

KAOS Methodology. To model the system using this methodology, and unlike *i**, the model was decomposed in three sub-models as can be seen in Fig. 6. Hence, the individual models represent (a) awareness goals, (b) collaborative systems goals and (c) software quality goals.
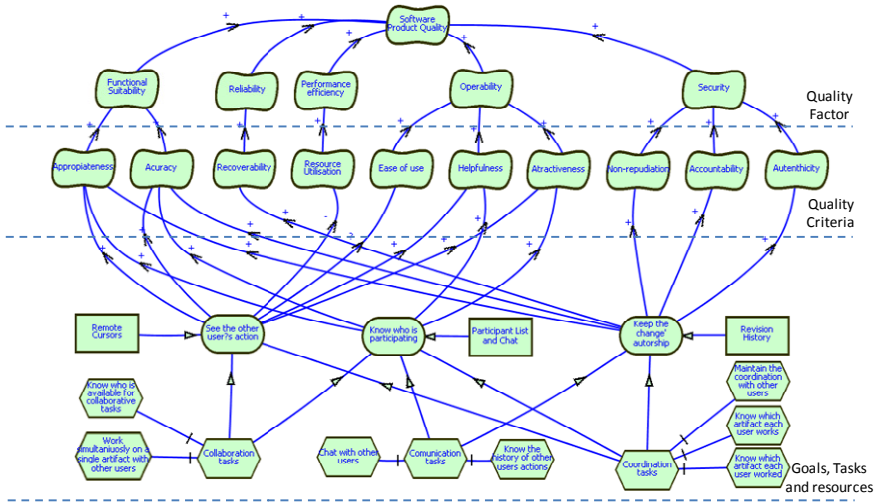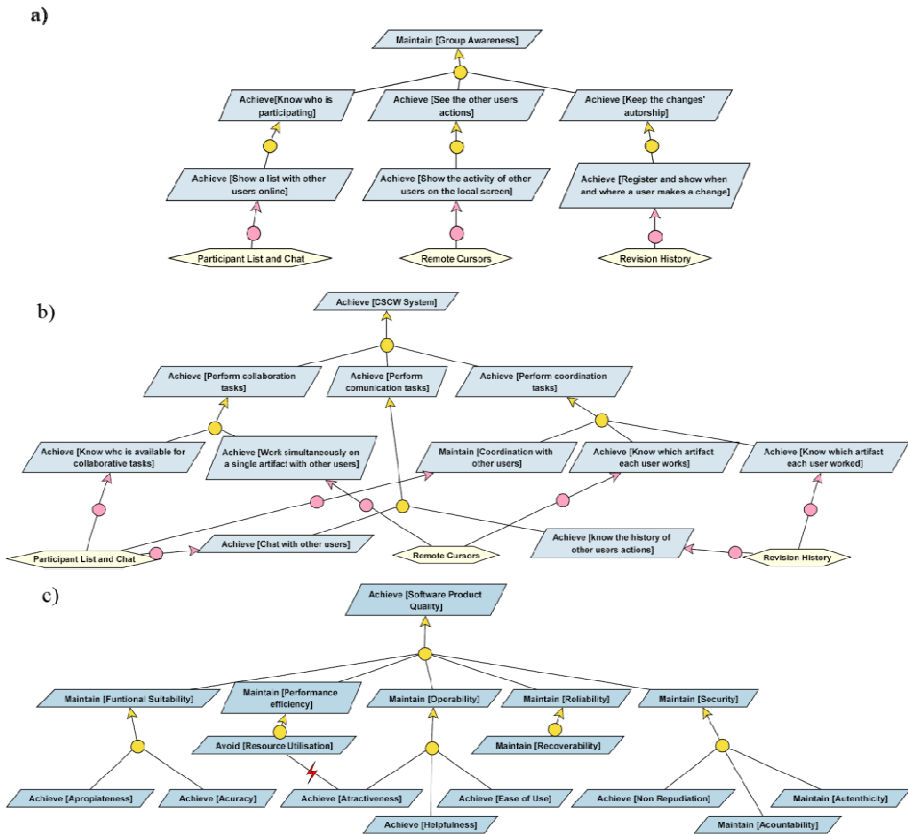
Fig. 7. i* Strategic Rationale Model

Fig. 8. KAOS Goal and Responsibility Model

These diagrams (Fig. 6) show three main goals and its decomposition in its sub-goals. The implemented awareness techniques have been represented here by using agents, because this element is used to represent responsibility assignment when using KAOS.

In addition, Fig. 6c illustrates a potential conflict between two softgoals related to two quality sub-factors: *avoid resource utilization* and *achieve attractiveness*. Usually, a very attractive user interface will cause higher resource utilization. This conflict is denoted in the graph by using a red ray.

5.2   Evaluating GO Approaches

Using as input the different specifications of the system, the evaluation of the different RE techniques was carried out by using *DESMET* [18]. It is a set of techniques applicable to evaluate both Software Engineering methods and tools. We have used the method based on a qualitative case study that describes a feature-based evaluation. Following the guidelines of this technique, an initial list of features was prepared that a GO approach for collaborative systems should provide (see Table 4). As can be observed, some of those features are directly related to the specification of NFRs.

Table 4. List of Features for approaches evaluation

Feature	Description
FR and NFR Representation	The model should be able to represent graphically FR and NFRs and differentiate them
Collaborative Systems Characteristics	The model has to represent the collaboration, communication and coordination characteristics
Awareness Representation	The model should allow one to represent the awareness characteristics of the system
Quality Factors Representation	The model must represent the SQuaRE characteristics and sub-characteristics
Importance of Requirements	The model should represent the importance and preference between requirements
Hierarchical Representation	The relation between the model elements should be hierarchical
Model Complexity	The model complexity should not be too high
Quantitative Model	The model must allow one to quantify the relations between represented elements
Traceability	The represented requirements should be traceable throughout the software development process

Once Table 4 is filled in, DESMET establishes that an importance degree should be assigned to each identified feature. Specifically, the degrees to apply are *Mandatory* (M), *Highly Desirable* (HD), *Desirable* (D) and *Nice to have* (N)

By using these degrees, Table 5 was filled in. As can be noticed, the most important features to be supported are both the NFR representation and the traceability required by collaborative systems.

Table 5. Importance of the features

Feature	Importance
FR and NFR Representation (RR)	M
Collaborative Systems Characteristics (CSC)	M
Awareness Representation (AR)	M
Quality Factors Representation (QFR)	HD
Importance of Requirements (IR)	HD
Traceability (T)	HD
Quantitative Model (QM)	D
Hierarchical Representation (HR)	D
Model Complexity (MC)	N

Next, according to DESMET, a scale to evaluate each one of the described features should be provided. The scale proposed by DESMET (see Table 6) was applied to evaluate each feature according to the following factors: *Conformance Acceptability Threshold* (CAT) and *Conformance score obtained for candidate method* (CSO)

Table 6. Judgement scale to assess support for a feature

Generic scale point	Definition of Scale point	Scale Point Mapping
Makes things worse	Cause Confusion. The way the feature is represented makes difficult its modelling and/or encourage its incorrect use	-1
No support	Fails to recognise it. The approach are not able to model a certain feature	0
Little support	The feature is supported indirectly, for example by the use of other model/approach in a non-standard combination	1
Some support	The feature is explicitly in the feature list of the model. However, some aspects of feature use are not catered for.	2
Strong support	The feature is explicitly in the feature list of the model. All aspects of the feature are covered but its use depends on the expertise of the user	3
Very strong support	The feature is explicitly in the feature list of the model. All aspects of the feature are covered and the approach provides a guide to assist the user	4
Full support	The feature appears explicitly in the feature list of the model. All its aspects are covered and the approach provides a methodology to assist the user	5

Once each feature was evaluated, the difference between CAT and CSO factors was computed as shown in the column *Difference* (Dif) in Table 7

Next, we should highlight that a variation of the DESMET method was used. The *importance* (Imp) of each feature has been weighted in a scale from 1 to 4 (Nice to have – 1, Desirable – 2, Highly Desirable – 3, Mandatory – 4). The importance was used to compute the final score of each feature by multiplying the *Importance* by the

Difference. This computation is shown in the column *Score* (Sco) in Table 7. Lastly, the final score of each technique (*Total*) was obtained by adding the scores of all the features. This framework has been used to evaluate all the different GO approaches studied.

Fig. 7 shows graphically the scores obtained by each one of the GO approaches. As can be observed, the *i** approach is the only one that has a positive score. Despite this positive score, it has been negatively evaluated for the Quantitative Model feature, since *i** only provides a partial support for quantifying the relations among requirements when using contribution links. The *i** approach also fails in representing the requirements importance, giving no support to determine which requirements are more important than others. Nevertheless, the other two GO approaches also share this lack of representation of the importance of each requirement. KAOS also fails in the same features than *i** but, unlike this approach, KAOS obtains a lower (or the same) score in almost all features except for the Hierarchical Representation feature, thanks to its tree-based representation. Finally, the NFR framework is the less suitable approach, obtaining a very low score, because of both the lack of expressiveness to specify FRs and its lack of adaptability to represent Collaborative Systems Characteristics.

Table 7. Results of approaches evaluation

Approach	NFR Framework					i*					KAOS				
Feature	Imp	CAT	CSO	Dif	Sco	Imp	CAT	CSO	Dif	Sco	Imp	CAT	CSO	Dif	Sco
RR	4	5	3	-2	-8	4	5	5	0	0	4	5	5	0	0
CSC	4	4	1	-3	-12	4	4	5	1	4	4	4	4	0	0
AR	4	4	4	0	0	4	4	5	1	4	4	4	4	0	0
QFR	3	3	5	2	6	3	3	5	2	6	3	3	4	1	3
IR	3	3	0	-3	-9	3	3	0	-3	-9	3	3	0	-3	-9
T	3	3	3	0	0	3	3	3	0	0	3	3	4	1	3
QM	2	2	1	-1	-2	2	2	1	-1	-2	2	2	0	-2	-4
HR	2	2	3	1	2	2	2	3	1	2	2	2	4	2	4
MC	1	1	3	2	2	1	1	1	0	0	1	1	2	1	1
Total					-21					5					-2
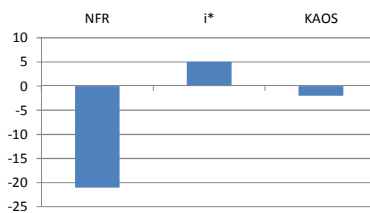
Fig. 9. Empirical analysis results

In addition, as DESMET suggests, we have performed a comparative of the percentage of each feature satisfied by each analyzed GO approach. Fig. 8 illustrates that the NFR approach only exceeds its competitors in the Model Complexity feature,

due to the simplicity their models have. Similarly, KAOS supersedes *i** in this feature because *i** has more modeling elements for the sake of expressiveness. In addition, the evaluation of Hierarchical Representation and Traceability features for KAOS is better than for *i** because *i** models are usually defined following a network structure and do not provide a sophisticated support for traceability. Other meaningful fact is that no approach is able to represent the importance of the requirements, something that should be considered in future works. Another significant result is that, despite *i** and KAOS have the same score for the feature FR and NFR Representation, *i** supersedes KAOS in the most important features (mandatory and high desirable ones) except for the Traceability feature. Nevertheless, KAOS obtains a better score in the less valuated features, like Hierarchical Representation and Model Complexity.
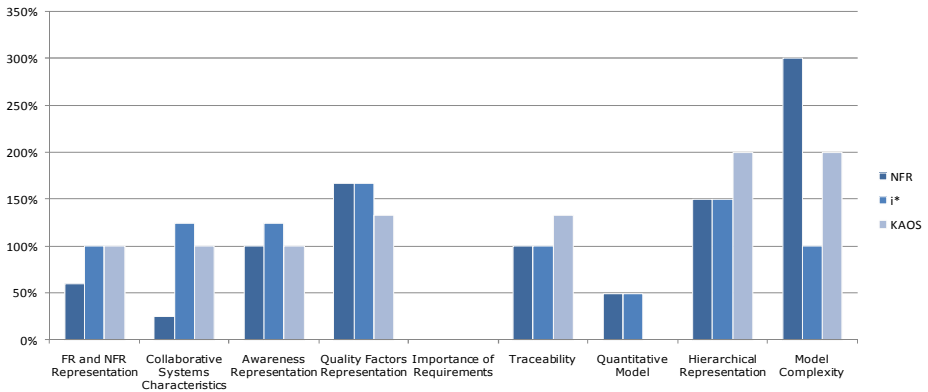

Fig. 10. Results relative to distinct features

6   Conclusions and Further Works

Collaborative systems are highly demanding in terms of NFRs. Therefore, the selection of a RE technique with proper support for their successful specification is a must. In this sense, the exploitation of the GO approach emerges as the most appropriate proposal [3]. However, up-to-date several RE techniques have been proposed that follow this approach. In order to select the most suitable one, in this paper the results of an empirical experiment of several GO techniques considering the special needs of CSCW systems have been conducted.

After this empirical experiment, we can conclude that the analyzed GO approaches are not fully appropriate to model collaborative system characteristics and its relationships with awareness and quality requirements. Among the analyzed GO techniques, the *i** approach is the only one that has a positive score for the analyzed features related to CSCW systems. In addition, *i** is the only one that provides (partial) support for quantifying the relations among requirements when using contribution links. However, this technique also exhibits some shortcomings, such as the lack of a hierarchical representation or support for specifying the importance of the requirements. But perhaps, the most significant shortcoming is that the comprehensibility of the awareness requirements is not appropriate. For instance, *i**

does not provide support to specify when a task is carried out by several roles, what is very common in a CSCW system.

These conclusions, along with the results shown in [3], support our initial hypothesis: current Requirement Engineering techniques should be enriched to address the issues identified during this study regarding CSCW systems. As was shown in this study, *i** is the most promising technique to be used as the foundation for this improvement. This constitutes one of our ongoing and challenging works: to adapt/extend *i**for this kind of systems [19,20].

In addition, a future work is the definition of techniques that support that the defined models can be used for validation purposes. That is, its conformance with the SQuaRE Quality in Use factors (usability, flexibility and safety) should be evaluable in an easy and intuitive way, once the system is fully developed.

References

1. Gutwin, C., Greenberg, S.: A Descriptive Framework of Workspace Awareness for Real-Time Groupware. Computer Supported Coop. Work 11, 411–446 (2002)
2. Hochmuller, H.: Towards the Proper Integration of Extra-Functional Requirements. Australasian Journal of Information Systems 6, 98–117 (1999)
3. Teruel, M.A., Navarro, E., Jaquero, V.L., Montero, F., González, P.: An Empirical Evaluation of Requirement Engineering Techniques for Collaborative Systems. In: 15th Int. Conf. on Evaluation and Assessment in Software Engineering, Durham, UK (2011)
4. Cysneiros, L.M., Yu, E.: Non-Functional Requirements Elicitation (Perspectives on Software Requirements). Springer, Heidelberg (2003)
5. Castro, J., Kolp, M., Mylopoulos, J.: A requirements-driven development methodology. In: Dittrich, K.R., Geppert, A., Norrie, M. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 108–123. Springer, Heidelberg (2001)
6. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: Fifth IEEE International Symposium on Requirements Engineering, pp. 249–263 (2001)
7. Google, "Google Docs" (2001), `http://docs.google.com`
8. Cockburn, A.: Writting Effective Use Cases. Addison-Wesley (2000)
9. Kavakli, E., Loucopoulos, P.: Goal Modeling in Requirements Engineering: Analysis and Critique of Current Methods. Information Modeling Methods and Methodologies, 102–124 (2004)
10. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: No Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers (1999)
11. Sampaio do Prado Leite, J.C., Franco, A.P.M.: A Strategy for Conceptual Model Acquisition. In: First Int. Symposium on Requirements Engineering, pp. 243–246 (1993)
12. Mylopoulos, J., Castro, J., Kolp, M.: Tropos: A Framework for Requirements-Driven Software Development. In: Inf. Systems Engineering: State of the Art and Research Themes, pp. 261–273 (2000)
13. Pohl, K.: Requirements Engineering: Fundamentals, Principles, and Techniques. Springer, Heidelberg (2010)

14. Gutwin, C., Greenberg, S., Roseman, M.: Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and Evaluation. In: HCI on People and Computers XI, pp. 281–298. Springer, Heidelberg (1996)
15. Schümmer, T., Lukosch, S.: Patterns for Computer-Mediated Interaction. John Wiley & Sons Ltd. (2007)
16. ISO/IEC 25010:2011, Systems and soft. engineering - Systems and soft. Quality Requirements and Evaluation (SQuaRE) - System and soft. quality models (2011)
17. Moreira, A.M.D., Araújo, J., Rashid, A.: A Concern-Oriented Requirements Engineering Model. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 293–308. Springer, Heidelberg (2005)
18. Kitchenham, B.: A methodology for evaluating software engineering methods and tools. In: Experimental Software Engineering Issues: Critical Assessment and Future Directions, pp. 121–124. Springer, Berlin (1993)
19. Teruel, M.A., Navarro, E., López-Jaquero, V., Montero, F., González, P.: CSRML: A Goal-Oriented Approach to Model Requirements for Collaborative Systems. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 33–46. Springer, Heidelberg (2011)
20. Teruel, M.A., Navarro, E., Jaquero, V.L., Montero, F., González, P.: Assesing the Understandability of Collaborative Systems Requirements Notations: an Empirical Study. In: 1st Int. Workshop on Empirical Requirements Engineering, Trento, Italy (2011)

Towards Interdisciplinary Approach
to SOA Implementations

Zheng Li[1,2], He Zhang[1,4], and Liam O'Brien[2,3]

[1] NICTA, Sydney, Australia
{Zheng.Li,He.Zhang}@nicta.com.au
[2] School of CS, ANU, Canberra, Australia
[3] CSIRO, Canberra, Australia
Liam.Obrien@csiro.au
[4] School of CSE, UNSW, Sydney, Australia

Abstract. Driven by the requirement of environment-adaptable business, Service-oriented Architecture (SOA) emerges with promising goals such as agility, flexibility, reusability and efficiency. Before reaching the goals, however, numerous and various challenges are still obstructing the success of SOA implementation. To efficiently deal with existing challenges, we can use an interdisciplinary approach to explore technology independent strategies as valuable and necessary supplements to technical concerns when implementing SOA. Through presenting an organizational view to comprehend SOA and treating SOA implementations as organizational activities, this paper employs useful knowledge in the organization theory area to inspire the research into technology independent strategies of SOA implementation. As a result, four preliminary strategies that can be applied to human organizations are identified to support building SOA systems. Furthermore, the interdisciplinary approach to investigating the success of SOA implementation is revealed, which encourages interdisciplinary research across service-oriented computing and organization theory.

Keywords: Service-oriented architecture (SOA), SOA implementation; organization theory, Organization design, Interdisciplinary approach.

1   Introduction

1.1   Challenges of Successful SOA Implementation

Service-Oriented Architecture (SOA) emerges with the requirements of quick response to the rapid and often unpredictable changes in business environment for modern enterprises. Motivated by the expectations of the people who are engaged in SOA activities, SOA has goals such as reusability of software assets across multiple platforms and applications, agility of support to business processes, efficiency in terms of development time and cost, and flexible integration of existing and legacy information systems. Unfortunately, there are numerous challenges of successful SOA

implementation before reaching the promising goals of SOA. For example, Gu and Lago's [15] systematic literature review on Service-Oriented Software Engineering identified 413 challenges in total under 45 topics, eight types, and two dimensions; Papazoglou, et al. [29] investigated the Service-Oriented Computing research road map, and drew 3 planes plus 1 other aspect each of which comprised 4 or 5 major challenges; and Kontogiannis, et al. [21, 22] used four domains to cover 86 challenges by exploring the landscape of Service-Oriented Systems. To deal with those challenges, some strategies have been proposed and developed over the past decade. For example, Newcomer and Lomow [26] use Web services to concrete the conceptual SOA; Krafzig, Banke and Slama [23] focus on the practical application of SOA in enterprises with discussion of the roadmap of relevant technologies; Erl [9, 10] summarizes a full scope of implementation strategies through eight principles of service design and 17 SOA design patterns. To the best of our knowledge, however, most of existing strategies for SOA implementation only pay attention on the technology aspect, particularly relying on the current state of the art of Web technology. As we know, technology is a necessary but not a sufficient condition for successful SOA implementations [31]. In other words, technology cannot guarantee the success of building an SOA system. Thus, the factors other than technology could be also vital for successfully implementing SOA. To efficiently address technology independent factors, we can try to resort to interdisciplinary approaches.

1.2   Interdisciplinary Research

As one of the most productive and effective method of solving complex questions and problems, Interdisciplinary Research (IDR) has been defined as "a mode of research by teams or individuals that integrates information, data, techniques, tools, perspectives, concepts, and/or theories from two or more disciplines or bodies of specialized knowledge to advance fundamental understanding or to solve problems whose solutions are beyond the scope of a single discipline or area of research practice." [6] In general, unfolding an IDR project may be a systematic and complex action particularly for those topics established through a top-down way: (1) Research questions are defined under predetermined interdisciplinary issues like world hunger, biomedical ethics, or sustainable resources; (2) Supportive environment is built up including both academic and industry resources; (3) Researchers with suitable interdisciplinary skills are employed; (4) IDR outcomes are finally described and evaluated. However, IDR can also evolve from a single and small problem through knowledge borrowing between different disciplines. Although it has been claimed that true IDR is not just pasting two disciplines together to achieve one goal, knowledge borrowing can be viewed as the initial step of a bottom-up approach to IDR. For example, biology gradually imports mathematical and physical sciences after becoming more quantitative, which eventually brings new interdisciplinary fields. Considering we are not building up a specific IDR project, we can start from knowledge borrowing to deal with SOA challenges in this case.

1.3   Seeking Interdisciplinary Strategies to Deal with SOA Challenges

Given the previous discussion, the question is where we can borrow useful knowledge to deal with SOA challenges. As we know, "SOA is a concept for large distributed systems" [18], which supposes services are decentralized and may be under the control of different owners. When it comes to the research into distributed systems, we can generally adopt two different approaches [12]: one is learning by doing (building real distributed systems), while the other is learning by analogy (drawing upon ideas from other research areas). In fact, the organization theory is usually employed to inspire the research in distributed systems. To the best of our knowledge, the use of organizational theory to guide technology research has proven significantly beneficial particularly in the multi-agent system community. For example:

- Well-known human organizational structures are used for the deployment of multi-agent systems [1];
- Social laws are chosen to simplify multi-agent systems [11];
- Dependency theory of social interaction is used to explain how to achieve social goals of multi-agent systems [34].

Therefore, this paper also presents an organization-based view to comprehend SOA, and treats SOA implementations as organizational activities. Based on the traditional consensus of the organization concept, thinking of SOA organizationally becomes reasonable. Moreover, the parallels are identified between organization design and SOA implementation in general, which follows a pentagonal process with five steps focusing on the *Goal and Strategy*, *Environment and Scope*, *Structure*, *Process* and *Coordination and Control*. Note that the "SOA implementation" we discuss here refers to common SOA implementation practices rather than any particular case. Enlightened by existing work of organization design in the organization theory domain, we have initially identified four strategies as a demonstration to meet four predetermined issues of service-oriented software engineering [22]. These four strategies are independent and each can improve SOA implementations depending on real circumstances. In other words, the strategies can be employed both individually and all together for an SOA implementation instance.

   The remainder of the paper is organized as follows. Section 2 justifies thinking of SOA from an organizational perspective. Section 3 analogizes the procedure of SOA implementation with that of organization design. Section 4 introduces four interdisciplinary strategies enlightened by organization design for SOA implementation. Section 5 uses an example to demonstrate how these four strategies are applied to improve SOA implementation. Conclusions are drawn and some future work is proposed in Section 6.

2   SOA: An Organizational Perspective

Organizations emerged as early as ancient civilizations appeared. Today, organizations have become indispensable and pervasive components of human beings' society, for example, from schools to hospitals and from armies to

governments. When it comes to the SOA area, we can similarly regard service-oriented systems as virtual organizations that are composed of services.

Viewing SOA systems as organizations is to use the organization concept to cover SOA systems, as shown in Fig. 1. Under the same umbrella of organization concept, both traditional organizations and SOA systems consist of organizational units. Organizational units in an SOA system are services, while that in a traditional organization are individuals. Furthermore, different organizational units have different skills and play different roles in an organization. For example, composite services play integrative roles in an SOA system, which parallels the responsibilities of managers in a traditional organization. Although there is no single agreement on the definition of an organization, theorists have traditionally consented that organizations are collectivities of people who are socially arranged to pursue specific purposes and achieve explicit goals [25]. This classical consensus makes it possible to think of SOA from the organizational perspective due to two reasons.
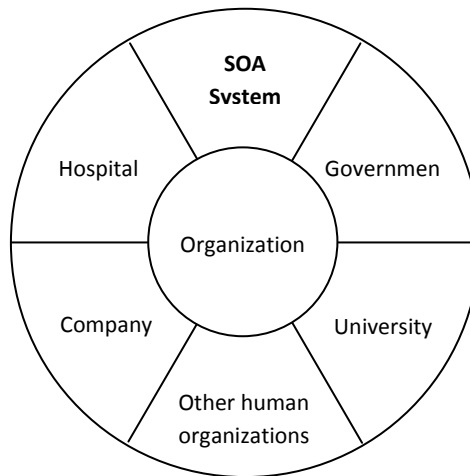

Fig. 1. View SOA system as an instance of organization

First, it is suitable to think of SOA representing organization architecture. The Organization for the Advancement of Structured Information Standards (OASIS) [27] defines SOA as "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations." When it comes to implementation, SOA is used to build up a collection of independent services that can be quickly and easily integrated into different, high-level business services and business processes to create business value and achieve business strategies [31]. To summarize, SOA both in theory and in practice is proposed for organizing services to attain some particular goals. Therefore, SOA can be set under the umbrella of organization theory in terms of the suggestion of traditional organization concept: if the organizing process is about goal attainment, the organization theory could be followed to conceptualize, explain and ultimately guide individuals' activities that should be united together to achieve desirable, common organizational goals [25].

Second, it is reassuring to think of SOA from the organizational perspective. In fact, conceptual challenge might appear when talking of organizations based on having a goal, because the agreement about an organization's purpose amongst members may not exist. In the SOA area, however, this disagreement issue can be ignored. Within SOA systems, a service is a well-defined unit of functionality realized by a service interface and a service implementation [28]. A service interface identifies a service and exposes the semantic description of the service's invocation. A service implementation realizes the work that the service is designed to perform. Unlike people in social organizations, services in SOA do not have mental or psychological attributes. Consequently, services will always obey the control from the "senior management" of the whole SOA system, and may even not be aware of the "organizational goal". When thinking of SOA organizationally, the blind obedience characteristic of services can naturally avoid the challenge of defining organizations in terms of having a goal while not all members freely agree to that goal [2].

Moreover, according to a set of general characteristics of the organizations identified by Campbell and Craig [4], we can find more similar features between SOA system and organization, as listed in our previous work [37]. Benefitting from thinking of SOA from the organizational perspective, we can use organization concept to comprehend SOA systems, and further use organization theory to inspire SOA implementations.

3   Analogies between SOA Implementation and Organization Design
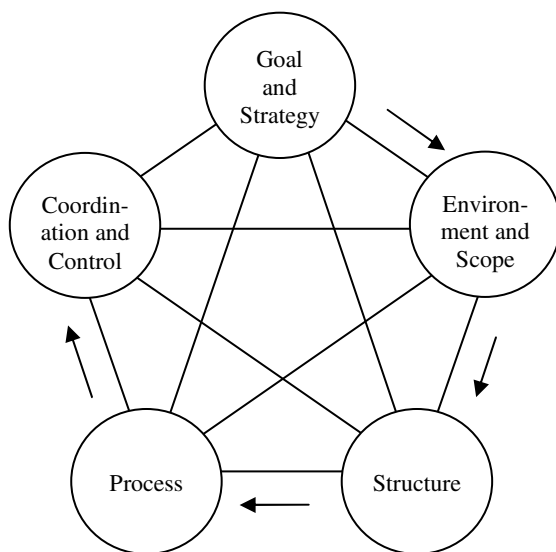
Fig. 2. The pentagonal process of SOA implementation / organization design

The similarity between SOA systems and organizations is not a coincidence. We can find common ground on a pentagonal process of SOA implementation [18, 23, 24, 31] and organization design [3, 7, 8, 19], which is identified through refining the waterfall process of organization design [3]. In the pentagonal process, five steps focusing on the *Goal and Strategy*, *Environment and Scope*, *Structure*, *Process* and *Coordination and Control* are executed generally along the clockwise sequence arrowed in Fig. 2. Meanwhile, each step has influence on as well as is under influence of the other four steps. For example, the goal and strategy together determine the whole process of organization design or SOA implementation, while they will be refined gradually as the process is unfolded.

3.1   Goal and Strategy

As mentioned previously, an organization must have a collective goal according to the traditional consensus of organization concept. Although different parts of the organization may have their own objectives, an overall collective goal can be established by aggregating all the separated objectives together. The overall goal is a desired direction that the organization will head. In practice, an organization's overall goal embodies a set of specified goals, each of which focuses on different aspect of the organization. Daft [7] distinguishes organization's goals into official goals and operative goals. The official goals formally define the business, values and outcomes that the organization attempts to achieve, while the operative goals are more explicit and scattered in different facets such as performance, efficiency, innovation and profit.

Goals of an organization introduce the target that the organization wants to pursue, while strategies define how the organization can pursue its target. Therefore, strategies can be treated as the operationalization of organization's goals [3]. Following the analysis of organization's goals, we can also distinguish organization's strategies into official strategies and operative strategies. The official strategies are essential plans of actions that can realize the corresponding official goals, for example the cost-leadership strategy or differentiation strategy. On the other hand, the operative strategies will aim at different detailed tasks like how to improve working efficiency or increase product profits. As different tasks may have resource conflict with each other, the strategy set should be carefully balanced.

Goals and strategies are in the first phase of organization design and essentially influence how an organization should be designed. Similarly, the first step of SOA implementation is to identify the business strategies and goals, and we can adopt the technique, namely business value chain, to help identify the specific goals and strategies for certain SOA projects [31]. Since service-oriented computing emerged from the requirement of addressing the rapid and usually unpredictable changes that modern enterprises are confronting, SOA systems contribute more promises than the traditional software infrastructures. Therefore, common goals and strategies can be extracted among general SOA implementations, which are emphasized in Section 4.

3.2   Environment and Scope

The environment is the surroundings of a system, and the system influences and is influenced by its environment. Meanwhile, the environment is not static but can be changing continuously and dynamically. Generally, there are five environment patterns interacting with any system, including asymptotic variation, interfering variation, periodic variation, phase-transition variation, and random variation [30].

Both SOA systems and organizations cannot be isolated from their external environments. The environment surrounding an SOA system or organization has a set of factors relating to resources or vulnerabilities. For example, the suppliers, customers, competitors, culture and government are organizations' environmental factors, while the developers, users, legacy system, existing service pool and state of current technology are SOA systems'. Building organization and implementing SOA are highly dependent on the environmental factors. In practice, the number of factors that constitute environment might be considerable. All these factors together reflect the boundary that an organization or SOA system, and then outline a scope, which determines the capability, applicability, competitive advantages and business range for the organization or SOA system.

For organization design, environment restricts organizations within certain scopes, and further influences their processes, structures and controls. For SOA implementation, analyzing the external environment and determining the applicable scope are particularly significant. SOA-based software infrastructure is supposed to be adaptive within an increasingly changing and complex environment. However, the loosely coupled asynchronous SOA systems are inherently more complex than the traditional architecture based systems. Josuttis [18] has pointed out that distributed processing would be inevitably more complicated than non-distributed processing, and any form of loose coupling increases complexity. In practice, building a true heterogeneous SOA for a wide range of operating environments may take years of development time if the company does not have sufficient SOA experience and expertise [17]. Since the more complexity involved in a system, the more difficulty the designers or engineers have to understand the implementation process and thus the system itself [5], SOA should be adopted only in the suitable environment and only when its benefits outweigh any extra costs due to the increased complexity.
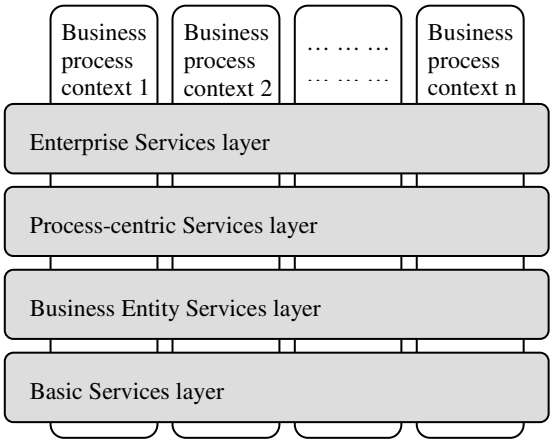
3.3   Structure

Structures of both organizations and SOA systems are established to divide up the work into manageable and measureable units with clear responsibility boundaries. Organization's structure is normally a hierarchy that allocates roles, power, authorities and responsibilities, and determines working relationships and communication channels. Generally, organizational units are arranged around functions, products/services, customers/geographies, or business processes. Therefore, the organization structures can be typically divided in five basic styles: functional, product- or service-based, customer- or geographical area based, business process based, and matrix structure [8]. Each kind of structure has specific advantages and

disadvantages. Unsuitable organization structure will result in formidable obstacle to align the other design elements with the organization's strategy [19]. Consequently, large organizations always build hybrid structures to achieve the combination of the advantages.

SOA systems normally adopt a matrix structure, which simultaneously groups services in two directions: functional direction and business process based direction, as shown in Fig. 3. The functional direction is to classify services according to the type of logic they encapsulate. Although there are quite a few service classifications that we can identify from the literature, most of the existing classifications can be unified and layered as Basic Services, Business Entity Services, Process-centric Services, and Enterprise Services. The Basic Services, settled at the bottom layer of SOA systems, provide reusable, technical, and foundational functionalities. The Business Entity Services represent the entities in business activities, such as employee, customer, contract, and product. Through composing relevant Business Entity Services, a Process-centric Service encapsulates a sequence of activities to complete a specific business task. The Enterprise Services provide endpoints to access the corresponding SOA systems, which could have less reuse potential but enable cross-enterprise integrations.

Fig. 3. The matrix structure of SOA systems

Meanwhile, to obtain the reuse of services in multiple business processes, the landscape of different services within different business process contexts should be described to show how the services work together [31]. The business process based direction of SOA systems' structure can then be outlined. Along the business process based direction, the overall services are grouped according to different roles and responsibilities within the real business. Each group may contain single service or multiple services. Moreover, the relationships to each other among the groups and the places of these groups in the business processes are also described.

3.4   Process

The organization design has been viewed from an information processing perspective [13]. The term process herein means not only business processes that produce products or services for customers, but also non-profit routines that constitute organizational actions. Generally, a process in an organization is a series of connected activities that transfer and transform information and resources through the organization [19], for example approving an application, submitting a report, and managing work progress. From the mid-1990s, many modern enterprises began to evolve into process-focused organizations in order to achieve higher performance and survive the market competition [33]. Currently, a process-focused organization has become the new organizational form with business process as the core concern. As a result, the design of processes significantly impacts on how well the organizational goals can be achieved.

Moreover, processes in an organization have close relationships with the organization's structure and coordination. All kinds of organization's structures inevitably create barriers to collaboration, because boundaries will appear as soon as organizational units are grouped under the structure. To fulfill the effective collaboration targets in organization, however, processes are required to flow cross the boundaries. Therefore, processes can glue the related organizational units to work together.

Like process-focused organizations, SOA systems can also be regarded as process-focused systems comprising of processes such as management process, coordination process, and traditional work process. Among all kinds of processes, SOA inherently concentrates on business process. Essentially, SOA is aligned with business process management (BPM) in business firms in which the criticality of business processes is concerned [24]. The emphasis of SOA is the functional infrastructure and the business services instead of the technical infrastructure and technical services. A business service encapsulates a piece or an entity of a business process. When implementing SOA, it is crucial to analyze business processes before identifying and developing services [31]. Following the analysis of business process, those potentially and even partially suitable services should be identified first. These existing services provide constraints that frame the future SOA system. The business processes are then broken down into business pieces that can be implemented by developing new services.

3.5   Coordination and Control

The coordination problem is one of the central topics in organizational studies [16]. As mentioned previously, individual actions of large numbers of interdependent roles and specialists must be coordinated to constitute processes to fulfill global tasks in an organization. On the other hand, the coordination will increase organizations' information processing capabilities when encountering increasing amount of uncertainty [13]. In practice, the activity of coordinating overlaps the activity of controlling, because the appearance of coordination usually implies the occurrence of

some control [8]. To coordinate and control organizational work, organizations should adopt suitable techniques and mechanisms. Unfortunately, there is not a fixed prescription of methods for coordinating and controlling work. The coordination and control, for example, can be simply related to the structures [19], be utilized by goal setting, hierarchy, and rules [13], or be executed by using four basic techniques: Supervision, Standardization, Building employee commitment, and Teams [8]. However, the principle of these techniques and mechanisms is uniform: to make sure organizational units work appropriately and find out to what extent they are reaching the goals and targets.

When implementing SOA, services must be composed to fully realize the benefits of SOA [32], which also relies on the coordinating and controlling activities. According to the cooperation fashions among component services, the mechanism of coordination and control can be distinguished between *Orchestration* and *Choreography*. Orchestration describes and executes a centralized process flow that normally acts as an intermediary to the involved services. Choreography describes multi-party collaboration and focuses on the peer-to-peer message exchange. In particular, if comparing Orchestration and Choreography with the two classical organization types – Mechanical and Organic organizations, we can find that the fundamental ideas and notions behind these different concepts in two disciplines are nearly the same.

4   Interdisciplinary Strategies for SOA Implementation

Inspired by existing research into aforementioned organization design, we can hereby explore interdisciplinary strategies to facilitate SOA implementation. Following four research topics of service-oriented software engineering [22], we have initially identified four strategies respectively (S1~S4).

4.1   S1: Applying TQM to SOA Implementation

Under the Engineering research topic of service-oriented computing, we propose to use Total Quality Management (TQM) to accommodate to the quality assurance of SOA implementation. Just as the name implies, TQM is a holistic level management for quality, because it can be achieved only if the total quality concept is utilized from the acquisition of resources to the customer satisfaction [20]. When it comes to SOA, the quality management has been emphasized to satisfy the unique characteristics of service-oriented computing. Nevertheless, to the best of our knowledge, existing research into quality management in SOA area is mainly at the service level, which is limited around the Quality of Service (QoS). The overall QoS of an SOA system is determined by all the QoS of component services who compose the SOA system [36]. Based on the QoS management, SOA systems generally replace component services with higher quality services to realize adaptations. Hence, the focus of QoS management is on individual services in an SOA environment.

When applying TQM to the SOA domain, Deming's 14 points [35] can be used as a framework to guide SOA implementations. For example, service suppliers and SOA system users should be taken into account when measuring the total quality of an SOA implementation. Here we focus on the quality of interaction and cooperation process among services. With reference to the explanation of TQM by Deming, in any circumstance, processes should be constantly analyzed to determine what changes can be made to bring improvement. Therefore, TQM introduces a new angle of view to SOA systems when adapting environment. However, employing TQM does not indicate abandoning QoS management. There is no conflict between TQM and QoS management. On the contrary, they are two complementary approaches for SOA to accommodate the changing environment: (1) TQM can be used to adjust the process of interaction and cooperation among services. (2) QoS management can be used to switch services based on the latest quality requirement.

4.2   S2: Flattening the Structure of SOA Systems

Under the Operations research topic of service-oriented computing, we propose to flatten the structure of SOA systems when considering service composition. In human organizations, every level in a hierarchy will inevitably involve more operating costs [14]. In a higher level, integrative roles are full-time managers who are in charge of orchestrating work across units [19]. These managers have accountability for results but are not directly responsible for the resource achieving and specific work that should be accomplished by staff. Considering a flat organizational structure can help reduce the number of integrative roles, organizations may increase efficiency by keeping their structures as flat as possible. Furthermore, a flat structure can decentralize responsibility and control to lower-level employees to take greater advantage of the skills and experience of organization members.

In an SOA system's hierarchy, the number of levels can increase along with the growing cascade of service composition. In general, a composite service is recursively defined as an aggregation of elementary and composite services. When thinking of SOA from the organizational perspective, composite services play integrative roles in an SOA system. Similarly, we can flatten the system's structure and move the additional functionality of original composite services upward, to reduce the composition cost and lower the complexity of the business process implementation.

However, we should keep the tall structure if the composite service already exists or its reusability is to be achieved. In other words, when applying this strategy, the value and cost should be well balanced to determine the extent of flattening structure.

4.3   S3: Taking Measurements at Interim Steps in Business Process

Under the Cross-Cutting research topic of service-oriented computing, we propose to take measurements at interim steps in business process to govern SOA implementation. When generating products following certain working processes or

designing the working processes in an organization, it has been proven valuable to take measurements at interim steps in those processes. The research and practice in organizations during the past decades reveal that it is increasingly important to ensure the work finishes properly the first time instead of having to be redone [8]. The inspections and measurements can be applied to different steps in processes to save the cost of rework and avoid flaws in the end product.

When applying this strategy to SOA implementation, the inspiration is to confirm the individual work of each service in business processes. The idea behind this strategy is to clearly define connected subtasks in a business process, and specify and measure the result of each subtask. It should be noted that measuring interim task mainly concerns the result rather than how the task is performed. Considering a service is such an entity that performs some task while hiding technical details, we can use the interim task measurement to help identify the most suitable services. Once all the services are determined, the relevant business process can then be correctly implemented.

4.4   S4: Building Virtual Business Process Teams

Under the Business research topic of service-oriented computing, we propose to build business process teams to facilitate mapping between business structure and service-oriented environment. In human organizations, teams are cross-functional structures that bring people outside the scope of traditional departments to work together and share collective responsibility for special and complex assignments. A business process team is established around one business process and includes people who can collectively perform all the major activities to carry out the business process from beginning to end.

Building business process teams in an SOA system should be a virtual division without many real actions. All the services involved in a business process logically constitute a team without changing the existing structure of the SOA system. Through virtual business process teams, the focus of coordination and control can be balanced between inward IT and outward business during SOA implementations. Furthermore, considering one service can be involved in different business process teams like the same scenario of organizational teams, we can identify and scale services' dependency of business processes in an SOA system. The more dependency a service has, the more carefully it should be controlled especially when planning to modify or replace this service.

5   An Example Case

Here we use a simplified case to demonstrate how those interdisciplinary strategies can be applied to improve an SOA implementation. The example case is an SOA-based application in a travel agency, as illustrated in Fig. 4.
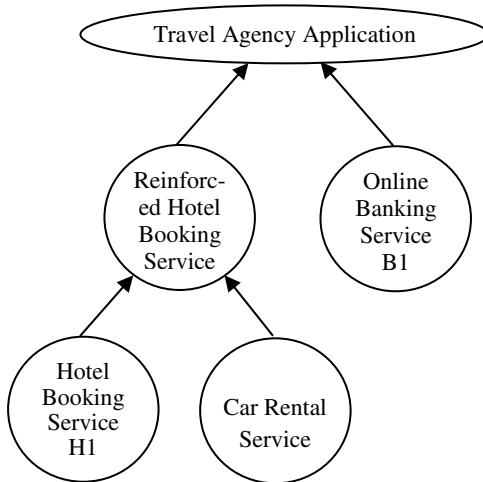
Fig. 4. An SOA-based application of hotel booking in a travel agency

The travel agency books hotel through BPay on behalf of a group of tourists, and will rent a car by money transfer if the number of the tourists is more than ten. Suppose there are three online banking services, two hotel booking services and one car rental service. Each service can fulfill its corresponding business function, while Online Banking Service B1 and Hotel Booking Service H1 are selected according to their reliability and response time. Moreover, the business rule "*rent a car by money transfer if the number of the tourists is more than ten*" is encapsulated in a composite service composed by Hotel Booking Service H1 and Car Renting Service. The composite service is implemented as a Reinforced Hotel Booking Service following the technology based strategy of using service composition to "fulfill a large extent of future business automation requirements" [9].

After a period of operation, the travel agency received some complaints. For example, small groups of tourists feel inconvenient without cars. Hence, the travel agency decides to thoroughly optimize the system. After applying the four proposed strategies, the evolution of this travel agency application can be illustrated in Fig. 5~8.

5.1   Applying TQM (S1)

When applying TQM to the SOA system to check the cooperation among services, we can find that the invocation of Car Rental Service is inflexible because an old business rule is hardcoded in the Reinforced Hotel Booking Service. Therefore, the number of tourists should be set as a variable and exposed as an input parameter of the composite service. The operation of this composite service is then adjusted by accepting one threshold parameter to improve the flexibility of the SOA system. As such, the business rule can be easily changed into "*rent a car by money transfer if the number of the tourists is more than five*". By using the red color to indicate the changed logic, the system is evolved as shown in Fig. 5.
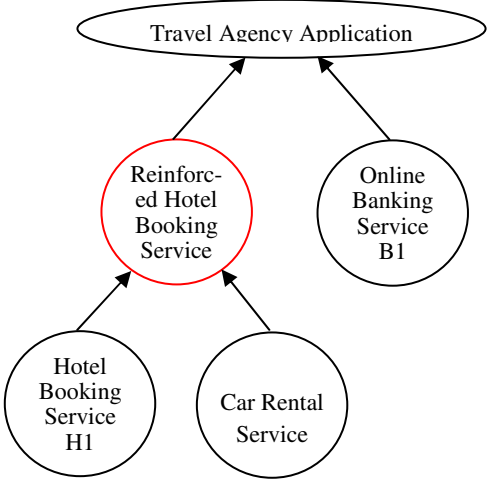
Fig. 5. The travel agency application after applying the first proposed strategy

5.2   Flattening the Structure (S2)

When analyzing the structure of the travel agency's SOA system, we find that the Reinforced Hotel Booking Service does not have any reuse opportunity. Furthermore, the encapsulated business rule can be easily transformed into control flow logic of invoking two component services, and moving the control flow logic into upper business logic will have little increase complexity for the latter. Therefore, we can flatten the structure by removing the Reinforced Hotel Booking Service to reduce the service maintenance effort. Fig. 6. shows the evolution after applying this strategy.
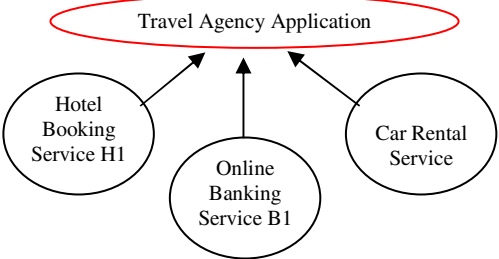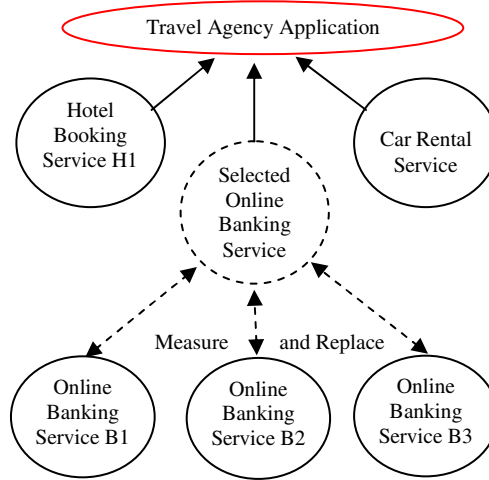

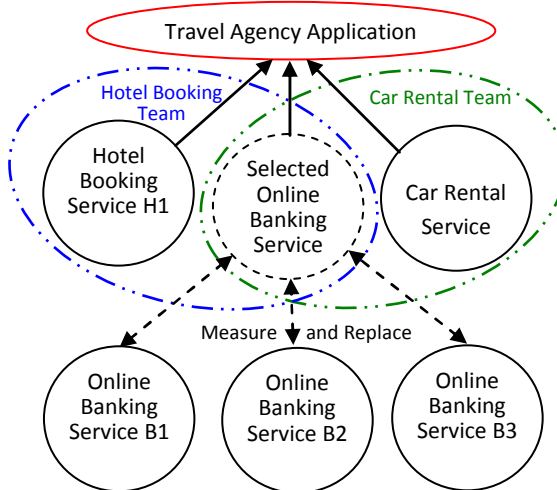**Fig. 6.** The travel agency application after applying the second proposed strategy

5.3   Measuring Interim Steps in Process (S3)

Suppose both BPay and money transfer will result in commission charges, and the charges vary depending on different bank and transaction time. We can then use the criterion "*choose bank with the lowest commission charges*" to constantly and simultaneously measure the three candidate online banking services. The service of

the bank that charges the lowest fee will be dynamically employed by the SOA system to help the travel agency save money. This evolution is illustrated as Fig. 7.

Fig. 7. The travel agency application after applying the third proposed strategy

Fig. 8. The travel agency application after applying four proposed strategies

5.4   Building Virtual Teams (S4)

Based on the business logic behind the travel agency application, we can identify there are two atomic business processes: one is hotel booking through BPay, and another is car rental through money transfer. Consequently, the selected Online Banking Service, Hotel Booking Service and Car Rental Service can be logically grouped into two business process teams. The coordination and control among

services in the hotel booking business process team follows the BPay rules, while in the car rental business process team obeys the money transfer rules. Through the team building, we can naturally arrange different cooperation for services in different teams, and in this case we may further identify the Online Banking Service is the key service when implementing the SOA system.

6   Conclusions and Future Work

The emergence of SOA has been considered a feasible opportunity for modern enterprises to leverage the capabilities of quickly adapting to competitive and changing environment. Compared with systems based on traditional architecture, however, SOA systems are inherently more complicated. Since the more complexity involved in a system, the more difficulty the designers or engineers have to understand the implementation process and thus the system itself [5], it is vital to find a set of implementation strategies to help achieve the promises of SOA. Based on the review of the relevant literature, we can identify a suite of technical strategies based on the current state of the art of Web technology. However, technology based strategies cannot guarantee the success of SOA implementations [31]. We therefore notice that the interdisciplinary strategies could also be emphasized as the supplements in implementing SOA.

By presenting an organization-based view to comprehend SOA, and treating SOA implementations as organizational activities, this paper delivers three main contributions. First, interdisciplinary research opportunities are suggested across the SOA area and the organization theory area. Second, the methodology of investigating strategies for implementing SOA is proposed by analogizing organization design with SOA implementation. Last, benefiting from existing work of organization design in the organization theory domain, four preliminary strategies conforming to the general goals of SOA are identified and suggested at this stage.

In particular, the evaluation for our work is different from traditional research topics. Although an example case has been elaborated to show the applicability of those identified interdisciplinary strategies for SOA implementation, their value and effectiveness still need to be further and widely investigated in practice. As we know, the strategies highlighted in this paper can be applied to different human organizations in general. Therefore, it is improper to use one SOA project or two to justify that the strategies are also generally suitable for SOA implementations. Consequently, our future work is to spread and apply these strategies in real scenarios and try to broadly collect the empirical results, as well as to explore other interdisciplinary strategies for SOA implementation.

References

1. Argente, E., Julian, V., Botti, V.: Multi-agent System Development based on Organizations. Electron. Notes Theor. Comput. Sci. 150(3), 55–71 (2006)
2. Bakan, J.: The Corporation: The Pathological Pursuit of Profit and Power. Free Press, New York (2005)

3. Burton, R.M., DeSanctis, G., Obel, B.: Organizational Design: A Step-by-Step Approach. Cambridge University Press, Cambridge (2006)
4. Campbell, D., Craig, T.: Organisations and the Business Environment, 2nd edn. Butterworth-Heinemann, Burlington (2005)
5. Cardoso, J.: How to Measure the Control-flow Complexity of Web Processes and Workflows. In: Fischer, L. (ed.) Workflow Handbook 2005, pp. 199–212. Layna Fischer (2005)
6. CFIR and CSEPP: Facilitating Interdisciplinary Research. The National Academies Press, Washington, DC (2005)
7. Daft, R.L.: Organization Theory and Design, 10th edn. South-Western College Pub., Mason (2009)
8. Davis, M.R., Weckler, D.A.: A Practical Guide to Organization Design. Crisp Publications, Inc., Menlo Park (1996)
9. Erl, T.: SOA Principles of Service Design. Prentice Hall PTR, Boston (2007)
10. Erl, T.: SOA Design Patterns. Prentice Hall PTR, Boston (2009)
11. Fitoussi, D., Tennenholtz, M.: Choosing Social Laws for Multi-agent Systems: Minimality and Simplicity. Artif. Intell. 119(1-2), 61–101 (2000)
12. Fox, M.S.: An Organizational View of Distributed Systems. IEEE Trans. Syst. Man Cybern. 11(1), 70–80 (1981)
13. Galbraith, J.R.: Organization Design: An Information Processing View. Interfaces 4(3), 28–36 (1974)
14. George, J., Jones, G.: Understanding and Managing Organizational Behavior, 5th edn. Prentice Hall, Boston (2007)
15. Gu, Q., Lago, P.: Exploring Service-Oriented System Engineering Challenges: A Systematic Literature Review. Serv. Oriented Comput. Appl. 3(3), 171–188 (2009)
16. Heath, C., Staudenmayer, N.: Coordination Neglect: How Lay Theories of Organizing Complicate Coordination in Organizations. Res. Organ. Behav. 22, 153–191 (2000)
17. Jamil, E.: SOA in Asynchronous Many-to-one Heterogeneous Bi-directional Data Synchronization for Mission Critical Applications. WeDoWebSphere (2009), http://wedowebsphere.de/node/30604
18. Josuttis, N.M.: SOA in Practice: The Art of Distributed System Design. O'Reilly Media, Inc., Sebastopol (2007)
19. Kates, A., Galbraith, J.R.: Designing Your Organization: Using the STAR Model to Solve 5 Critical Design Challenges. Jossey-Bass, San Francisco (2007)
20. Kaynak, H.: The Relationship between Total Quality Management Practices and Their Effects on Firm Performance. J. Oper. Manag. 21(4), 405–435 (2003)
21. Kontogiannis, K., Lewis, G.A., Smith, D.B., Litoiu, M., Muller, H., Schuster, S., Stroulia, E.: The Landscape of Service-Oriented Systems: A Research Perspective. In: 1st International Workshop on Systems Development in SOA Environments (SDSOA 2007). IEEE Computer Society (2007)
22. Kontogiannis, K., Lewis, G.A., Smith, D.B.: A Research Agenda for Service-Oriented Architecture. In: 2nd International Workshop on Systems Development in SOA Environments (SDSOA 2008), pp. 1–6. ACM Press (2008)
23. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA: Service-Oriented Architecture Best Practices. Prentice Hall, Upper Saddle River (2004)
24. Lawler, J.P., Howell-Barber, H.: Service-Oriented Architecture: SOA Strategy, Methodology, and Technology. Auerbach Publications, Boca Raton (2007)
25. McAuley, J., Duberley, J., Johnson, P.: Organization Theory: Challenges and Perspectives, 1st edn. Prentice Hall, England (2007)

26. Newcomer, E., Lomow, G.: Understanding SOA with Web Services. Addison-Wesley Professional, Upper Saddle River (2004)
27. OASIS: A Reference Model for Service-Oriented Architecture. White Paper, Billerica, MA (2006)
28. Papazoglou, M.P., Heuvel, W.-J.: Service Oriented Architectures: Approaches, Technologies and Research Issues. VLDB. J. 16(3), 389–415 (2007)
29. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. Computer 40(11), 38–45 (2007)
30. Peng, Y., Liu, H., Tao, H.: Analyzing the Pathway of Organizational Change based on the Environmental Complexity. In: 2009 International Conference on Electronic Commerce and Business Intelligence (ECBI 2009), pp. 463–466. IEEE Computer Society (2009)
31. Rosen, M., Lublinsky, B., Smith, K.T., Balcer, M.J.: Applied SOA: Service-Oriented Architecture and Design Strategies. Wiley, Indianapolis (2008)
32. Sarang, P., Jennings, F., Juric, M., Loganathan, R.: SOA Approach to Integration: XML, Web Services, ESB, and BPEL in Real-World SOA Projects. Packt Publishing, Birmingham (2007)
33. Seltsikas, P.: Organizing the Information Management Process in Process-based Organizations. In: 34th Hawaii International Conference on System Sciences (HICSS 34), p. 8066. IEEE Computer Society (2001)
34. Sichman, J., Demazeau, Y.: On Social Reasoning in Multi-agent Systems. Rev. Iberoamericana. de I. A 13, 68–84 (2001)
35. Walton, M.: The Deming Management Method. Perigee Books, New York (1988)
36. Yau, S.S., Ye, N., Sarjoughian, H., Huang, D.: Developing Service-based Software Systems with QoS Monitoring and Adaptation. In: 12th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2008), pp. 74–80. IEEE Computer Society (2008)
37. Li, Z., Zhang, H., O'Brien, L.: Towards Technology Independent Strategies for SOA Implementations. In: 6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2011), pp. 143–154. SciTePress (2011)

Formalisation of a Generic Extra-Functional Properties Framework*

Kamil Ježek and Premek Brada

Department of Computer Science and Engineering, University of West Bohemia
Univerzitni 8, 30614 Pilsen, Czech Republic
{kjezek,brada}@kiv.zcu.cz

Abstract. Approaches to improve software composition become remarkably important with the gradual enlargement of software systems. Together with adaptation of component-based programming to cope with software complexity, extra-functional properties are playing a more important role. The problem addressed in this paper concerns an insufficient adoption of extra-functional properties into practise that consequently limits approaches to modularised software. As a solution this paper presents a comprehensive framework which enables the use of extra-functional properties in existing systems with the promise to improve component application consistency. The framework is described in a formal manner and its practical application is shown on the Spring and OSGi component models.

Keywords: Software components, Extra-functional properties, Compatibility, Repository, Inter-component binding, Framework.

1 Introduction

With today's need for large software systems both industry and the research community invest considerable effort into improving component programming. Despite noticeable benefits of the components, new issues keep to arise. One of the important ones concern the usage of extra-functional properties (EFPs).

Extra-functional properties provide means to assess the applicability and compatibility of components considering (1) qualitative properties such as *speed*, *response time*, *memory consumption* or (2) user requirements such as *marketability*, *price*, *regular updates*, *technical support* or (3) behavior properties such as *synchronisation*, *concurrent access*, *deadlock free computation*.

Research and industrial efforts range from describing EFPs [1,2] through their application in specialized component models [3,4] to usage in quality of service specifications [5,6]. Although these works have already shown the directions, industrial frameworks such as Spring or OSGi do not support EFPs.

In this paper we present a different approach. Rather than creating a native EFP support for a given component model, we propose a structural EFP framework applicable to many existing component frameworks.

1.1   Structure of the Paper

We first introduce the proposed extra-functional framework with its modules in Section 2. An introduction of the concept is followed by formalisations of key parts of the framework in Section 3. Section 4 presents examples of the applicability of the approach to selected industrial frameworks.

2   Extra-Functional Framework Modules

The proposed framework aims at covering the activities related to the use of extra-functional properties in component-based development; namely the definition, attachment and evaluation of EFPs.

The conceptual structure of the framework consists of four modules as depicted in Figure 1. The Repository stores EFP definitions and is accessed by other modules. The EFP Assignment part uses the Repository so it can attach the declared EFPs to each component. Once components are enriched by EFPs the Evaluator takes care of comparing of EFPs to verify the compatibility during their binding.

All these modules are tied together by EFP Types which defines the structure (type and values) of individual EFPs [7].
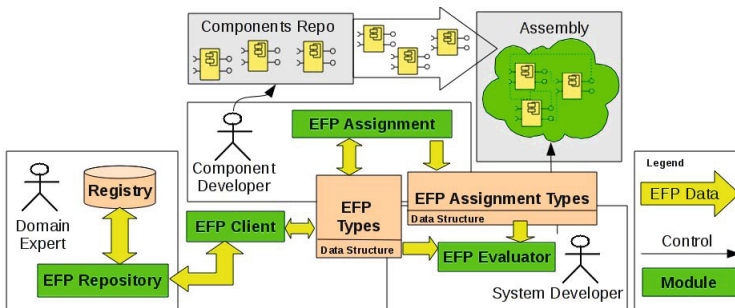

Fig. 1. Framework Overview

The result is a loosely coupled framework applicable to a wide set of component models. The only assumption is that the targeted models recognize required and provided counterpart elements used when creating inter-component bindings [8].

In the following subsections we respectively describe the details of the EFP Types, the Repository, how the EFP Assignment module achieves component framework linking, and EFP evaluation.

2.1   Extra-Functional Properties Types

The interchange of extra-functional properties between the modules of the framework requires a shared understanding of EFP data. This is realized by the EFP Types module which is the implementation of the model of extra-functional properties presented in

[7] and formalized in [9]. It defines individual EFPs, their structure and relations to a system of registries.

Getting some inspiration in NoFun [10], we distinguish between simple and derived extra-functional properties. The semantics is that a derived property is based on a set of other (simple or derived) properties and its value is computed according to a deriving formula.

We formally define a collection of extra-functional properties as a set

$$E = \{e \mid e = (n, T, E_d, \gamma, M)\} \tag{1}$$

where n is the name of a property,

$T \in T_{types} = T_c \cup T_s$ is the type of a property,

$\quad T_s$ is a set of simple (primitive) types; $T_s = \{real, integer, boolean, enum, set, ratio, string\}$,

$\quad T_c = \{(T_1, \cdots, T_N) \mid N > 1, T_i \in T_{types}\}$ is a set of complex types containing non-primitive values,

$E_d \subset E, e \notin E_d$ is a (possibly empty) set of other composing EFPs in case e is a derived EFP,

$\gamma : T \times T \to Z; Z = integer \cup \{\text{``}n/d\text{''}\}$ is a function which compares two instances $x, y \in T$ of the property with type T, stating which of the two values is better. The meaning of the return values is: *negative integer*: x is worse than y, 0: x is equal to y, *positive integer*: x is better than y, "n/d": not-defined. A natural default function (e.g. $x - y$) is used for primitive types.

M is a record with additional information. It currently contains the items $unit, names$, where $unit$ is a measuring unit and $names$ is an ordered enumeration containing all names for the values of this property used in the Local Registries from Section 2.2.

For instance, two simple properties and a derived one are defined as follows:

```
(time_to_process, integer, empty-set, default-gamma,
  M {unit:``ms'', names: {low, average, high}} )
(data_transferred, integer, empty-set, default-gamma,
  M {unit:``MB'', names: {low, average, high}} )

(performance, enum {sufficient, insufficient},
  {time_to_process, data_transferred}, default-gamma,
  M {} )
```

2.2   Universal EFP Repository

In component development, the components are typically developed by many organizations. If two vendors work with the same EFPs, the same understanding of these properties is needed. The role of the repository is to guarantee such understanding. Further, since a given component or service can be run in different environments, an EFPs mechanism must handle this heterogeneity.

We therefore use a layered design of the repository [9]. The upper layer called Global Registry (GR) is a storage of EFP definitions. It collects definitions of EFP types from

Section 2.1. Let us highlight that Global Registry does not contain concrete values of EFPs because they may differ widely among runtime and application environments.

The lower layer of the repository is called Local Registry (LR). For a subset of EFP Types defined in a GR, it stores concrete values pertinent to a particular computational environment.

Formally, the Global and Local Registries are tuples:

$$GR = (id, name, E) \tag{2}$$

$$LR = (id, name, GR, id_{parent}, S, D) \tag{3}$$

id : Integer is the registry's unique identifier,

$name$: String is its human readable name,

E is a set of extra-functional properties (see Section 2.1 above),

GR is the Global Registry this LR is linked to,

id_{parent} : Integer is the identifier of an (optional) parent LR; the semantics is that a value from a parent is inherited unless overriden by this LR.

$S = \{(e, value\_name, v) \mid e \in E \wedge value\_name \in String \wedge v \in V_{LR}\}$ is a set defining context dependent values for simple properties, where

    e is a property from GR,

    $value\_name$ is a name assigned to the value v such that $value\_name \in e :: M :: names$ i.e. it must be selected from the list of names given in the definition of e in GR,

    $V_{LR} = \{v_i\}_{i \in I}$ set holds all values for the given EFP assigned in this LR or its parents; I is the set indexing these values.

$D = \{(e, value\_name, v, r) \mid e \in E \wedge value\_name \in String \wedge r \in R \wedge v \in V_{LR}\}$ is a set of values for derived properties, where R is a set of rules (formulas) deriving an EFP value from other EFPs and their values; i.e. $R = \{f(x) \mid f : E \times V_{LR} \rightarrow V_{LR}\}$.

Local Registry holds context-dependent values with assigned names, so the names remain the same while concrete values differ. The key advantage is that a developer may think of the semantics of the EFP value denoted by the name rather than about a concrete number. In addition, this solution partitions continuums of values into disjunctive named intervals where all values in one partition may be treated as equivalent. For example, memory consumption in the interval $\{1, +\infty\}GB$ may be considered *too high* for resource constrained devices.

The first example below shows a LR for smartphones with GPRS-only connection while second one shows a LR for wifi-connected tablets:

```
(1) time_to_process: low = 10, high=5000, ...
    data_transferred: low = 1, high=100, ...

(2) time_to_process: low = 1, high=1000, ...
    data_transferred: low = 10, high=500, ...
```
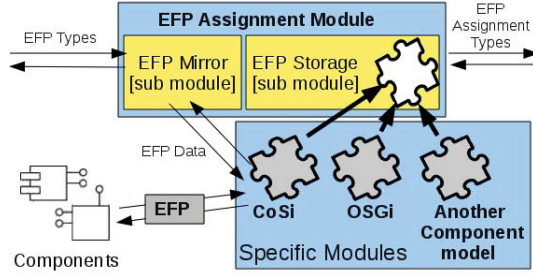
Fig. 2. EFP Assignment Module

2.3 Assigning EFPs to Components: Applicability of EFPs to Variety of Component Models

The assignment of EFPs to components actually consists of two phases: (1) a developer attaches EFPs to components, loading the properties from the repository, (2) the EFP Assignment module shown in Figure 2 provides the previously attached EFPs to other systems in a form of so called EFP Assignment Types.

The EFP Mirror sub-module represents the component framework-independent part of the EFP Assignment module. In the phase of attaching EFPs to a component, a developer loads EFPs from the remote EFP repository to mirror them on the component for a later phase when this module is disconnected from the repository.

Hence, this approach provides component frameworks with a transparent access to the EFP data in an independent format of the EFP Assignment Types.

The Data Storage Sub-module provides an extension point where implementations for supported component models are plugged. This sub-module brings the desired flexibility and applicability for different component models in a form of lightweight plug-ins. Each implementation of this sub-module decides (1) where and how to store EFP data, and (2) how to link the data with concrete features of the component.

2.4 EFP Assignment Types

Since the framework aims at generality, data exchanged between its modules must cover a wide spectrum of component models. For that reason it uses a generic representation of EFPs attached to components, called EFP Assignment Types. It aggregates the EFP Types and the information about the assignment of EFP values to components.

EFP Assignment Types is formally defined as a set $AT = F \times E \times V$ where F is a set of generic representations of all component features (see also [11]):

$$F = \{f \mid f = (name, type, role, mandatory)\} \tag{4}$$

where $name$: String is the name of the feature,

$role \in \{\text{``}required\text{''}, \text{``}provided\text{''}\}$ expresses whether the feature is put on the required or the provided side of a component,

$mandatory$: Boolean determines whether this feature must be bound to another feature in the matching process, and

type represents type information of the feature including a meta-type (for instance "interface" or "package") and parameters (e.g. inputs and outputs of methods). The type can be extended with a version identifier.

Two features are bound to each other by a function:

$$\mu : F \times F \rightarrow Boolean \tag{5}$$

where μ is a matching function on F taking two features as its input and returning $true$ if they can be bound, $false$ otherwise.

A default behavior of the function uses the following rules: (1) names are equal for both features, (2) a provided feature matches only with a required one or vice versa, (3) mandatory required feature must have a provided counterpart, (4) features are compatible in terms of their types (e.g. parameters of interfaces are of the same types). A required feature must be a sub-type of the provided feature. If the type compatibility is explicitly expressed as versions, then a version on the provided side is equal or greater than a version on the required side or vice versa.

However a more sophisticated matching μ function can be provided. For example, compatibility on interfaces using subtype relation [12] would reach more accurate results. Since instances of the features come from the EFP Assignment module, the extension is straightforward: the re-implementation of a component-specific sub-module in the assignment module provides different μ function while the algorithm of the evaluator remains unchanged.

Continuing with the definition of EFP Assignment Types, E is a set of extra-functional properties from Section 2.1 and V is a set of values [7] assigned to the properties which has three forms:

$$V = V_{direct} \cup V_{LR} \cup V_{formula} \tag{6}$$

$v \in V_{direct}$ is a directly assigned value; this assignment is used when the value remains constant independently on a runtime environment.

$v \in V_{LR}$ is a value defined by a local registry of the EFP repository (LR) typically holding values dependent on a context of usage and varying among contexts. A component can thus reference multiple values of a given property for different contexts. When evaluating components, one must select which context a result should be computed for and the evaluator then uses only values valid for the selected context.

$v \in V_{formula}$ is a mathematical formula, declared directly at the component, determining the value of an EFP from other EFPs. This kind of assignment allows to compute EFPs on the provided side of component based on those on the required side, including the ones specified for the deployment environment. For instance, a component may declare its speed-up by the Amdahl's law $\frac{1}{(1-P)+\frac{P}{S}}$. P expresses the amount of a code which may be parallelized and for a particular component it is constant (e.g. 30%). S is a number of processors depending on a runtime environment. Hence the component claims its speed-up based on the input parameter from the runtime.

The following example uses an EFP from Section 2.1 to show assignment of EFP values to a feature using all the above forms:

```
(  # feature
 ("DataAccess", "interface", "provided",
    true, "matched-by-name"), # EFP, values
 (time_to_process, (LR.1::low, LR.2::average, direct::20,
    math::(2 * DataAccess::data_transferred))))
```

The `matched-by-name` denotes a μ function which matches two features with the same names. `LR.1` and `LR.2` are two local registries identified by their IDs, `direct` is a direct value – 20ms in this example, `math` defines a math formula. There must also be an assignment for the `data_transferred` extra-functional property, which we omit here for space constraints. This artificial example is used to show all options; typical cases have only one type of value (LR, direct or math-formula one) defined in the assignment.

2.5 EFP Evaluation and Binding

The final part of the framework is the Evaluator. Its main purpose is to load a set of components and verify their compatibility in terms of extra-functional properties.

The module first obtains EFPs of components by calling the EFP Assignment module for each component. The received data are then composed to a graph representing components and their bindings, which serves the evaluator to find problems in component compatibility. Shortly, binding problems have the form of missing edges in the graph while EFP incompatibilities show as mismatches on respective edges; see Section 3 below for more details.

Unlike other modules the Evaluator is not customizable because it works on a generic model of EFPs and component application architecture.

3   EFP Evaluator Algorithms and Formalizations

The following sections detail the process of the evaluation using more formal means. Formal definitions of data used by the evaluator and the algorithm verifying compatibility of components are also introduced.

3.1 Structure of EFPs Graph

Once the EFP Evaluator obtains a set of EFP Assignment Types, it can compose a graph representing the application structure annotated with properties.

The graph which is created by the EFP Evaluator is an oriented graph $\overrightarrow{G} = (V, E)$ where V is a set of vertexes and E is a set of edges, with specialized types of vertexes and edges:

$$V(\overrightarrow{G}) = V_{component}(\overrightarrow{G}) \cup V_{feature}(\overrightarrow{G}) \cup V_{efp}(\overrightarrow{G})$$
$$E(\overrightarrow{G}) = E_{belong}(\overrightarrow{G}) \cup E_{match}(\overrightarrow{G})$$

The following rules hold for each vertex v:

$v \in V_{component}(\overrightarrow{G})$ if v represents a component. It is a root meta-vertex which purpose is to simply aggregate all features of a component.

$v \in V_{feature}(\overrightarrow{G})$ if the vertex represents a feature. It expresses one type of feature depending on concrete implementation for a particular component model. It may express e.g. "interface", "service" or whole "component".

$v \in V_{efp}(\overrightarrow{G})$ if the vertex represents an EFP. These vertexes are connected with $V_{feature}(\overrightarrow{G})$ vertexes to express EFPs on concrete features of a component.

The E_{belong} edges expresses how components, features and EFPs are connected:

$e \in E_{belong}(\overrightarrow{G})$:
$$\begin{cases} (v_x, v_y) \mid v_x \in V_{component}(\overrightarrow{G}) \land v_y \in V_{feature}(\overrightarrow{G}) : \text{required feature,} \\ (v_x, v_y) \mid v_x \in V_{feature}(\overrightarrow{G}) \land v_y \in V_{component}(\overrightarrow{G}) : \text{provided feature,} \\ (v_x, v_y) \mid v_x \in V_{feature}(\overrightarrow{G}) \land v_y \in V_{efp}(\overrightarrow{G}) : \text{required EFP,} \\ (v_x, v_y) \mid v_x \in V_{efp}(\overrightarrow{G}) \land v_y \in V_{feature}(\overrightarrow{G}) : \text{provided EFP.} \end{cases}$$

The E_{match} edges expresses how features are bound and EFPs are matched. While features are bound by the function μ defined above, EFPs are matched via their names and their relation to a feature. It means that one EFP may be attached to multiple features, but only once to the same feature:

$e \in E_{match}(\overrightarrow{G})$:
$$\begin{cases} (v_x, v_y) \mid v_x \in V_{feature}(\overrightarrow{G}) \land v_y \in V_{feature}(\overrightarrow{G}) : \text{binding features,} \\ (v_x, v_y) \mid v_x \in V_{efp}(\overrightarrow{G}) \land v_y \in V_{efp}(\overrightarrow{G}) : \text{matching EFPs.} \end{cases}$$

Using this model, the Evaluator generates the graph in several steps. It first creates component vertexes ($V_{component}(\overrightarrow{G})$) from a set of components a user desires to evaluate. Secondly, the EFP Assignment Types are added for each component and vertexes for features ($V_{feature}(\overrightarrow{G})$) and EFPs ($V_{efp}(\overrightarrow{G})$) are created. Furthermore, the vertexes are connected using the "belong" edges ($E_{belong}(\overrightarrow{G})$) to express which features and EFPs are attached to the components.

Finally, isolated graphs, representing individual components, produced by the previous steps are connected by matching all pairs of corresponding provided-required features as well as EFPs. Hence, "match" edges of the type $E_{match}(\overrightarrow{G})$ complete the graph. These edges denote the connections of features among components and pairs of EFPs attached to the features.

The final graph completely represents components and their bindings together with their EFPs.

3.2   Evaluation of EFPs

Having a graph representation of component connections, the evaluation is quite straightforward. The evaluator must go through the graph and find possible problems in vertex connections first, then it uses the values attached to EFPs to compare the value pairs of two connected EFPs.
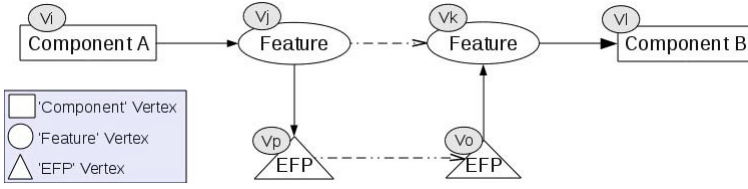
Fig. 3. Example Graph

The algorithm which computes the values of attached EFPs as well as checks the connection of components with each other uses a modified depth first search algorithm. It has the following steps (let us use a notation v_x to denote a particular vertex, e_{xy} to denote an edge from v_x to v_y where $x, y \in I$ and I is a finite index set for indexing vertexes and edges respectively):

1. Input sets of vertexes $V_{component}(\overrightarrow{G})$, $V_{feature}(\overrightarrow{G})$ and $V_{efp}(\overrightarrow{G})$ are established and the first vertex $v_i \in V_{component}(\overrightarrow{G})$ is selected, temporary vertexes $v_j, v_k, v_l = null$, previous vertex $v_{i-1} = null$.
2. Find a first feature vertex finding the first edge $e_{ij} \in E(\overrightarrow{G})$ where $v_j \in V_{feature}(\overrightarrow{G})$. The direction of the edge is from a component to the feature which symbols a required feature. If there is no such edge, the component has no required element and a new input set is specified $V_{component}(\overrightarrow{G}) = V_{component}(\overrightarrow{G}) - \{v_i\}$ and the algorithm goes to step 5. Otherwise, proceed.
3. Find the first edge $e_{jk} \in E(\overrightarrow{G})$ where $v_k \in V_{feature}(\overrightarrow{G})$. A direction is from a feature to another feature representing a connection of required feature to a matching provided one. If the edge is not found and the feature is mandatory, it means that a requirement of the component is not fulfilled $\rightarrow$ ERROR. For non-mandatory features, the algorithm goes back to step 2. Otherwise, set $v_{i-1} = v_i$.
4. An edge $e_{kl} \in E(\overrightarrow{G})$ where $v_l \in V_{component}(\overrightarrow{G})$ is selected. The new first vertex $v_i = v_l$ is set and the algorithm goes back to step 2.
5. If $v_j, v_k \neq null$ then two sets of vertexes $V_{efp_k} = \{v_o \mid \exists k : e_{ok} \in E(\overrightarrow{G}) \wedge v_k \in V_{feature}(\overrightarrow{G}) \wedge v_o \in V_{efp}(\overrightarrow{G})\}$ and $V_{efp_j} = \{v_p \mid \exists j : e_{jp} \in E(\overrightarrow{G}) \wedge v_j \in V_{feature}(\overrightarrow{G}) \wedge v_p \in V_{efp}(\overrightarrow{G})\}$ are selected. V_{efp_k} is a set of EFPs on the feature v_k and their values are computed first, then EFPs V_{efp_j} on the feature v_j are also computed. The vertexes are removed from the input set $V_{feature} = V_{feature} - \{v_j, v_k\}$. The values are computed as follows:
   - Direct value: a concrete value assigned to the EFP is directly used;
   - Local value: a value for a context of usage a user aims to compute is used;
   - Mathematical formula: computed using values on connected components which must be certainly known at this time (because the depth first search algorithm must have already visited connected components).

   In addition, the following must hold: $\forall p \exists o : e_{po} \in E(\overrightarrow{G}) \wedge v_o \in V_{efp_k}(\overrightarrow{G}) \wedge v_p \in V_{efp_j}(\overrightarrow{G})$ meaning that all EFPs on the required side must be connected to their provided counterparts, otherwise $\rightarrow$ ERROR.

6. If $v_{i-1} \neq null$ a new initial vertex is set $v_i = v_{i-1}$ else if the set $V_{component} \neq \emptyset$, select another vertex $v_i \in V_{component}(\overrightarrow{G})$. Then go back to step 2. Otherwise, the graph evaluation ends.

The algorithm verifies any inconsistency in the graph in terms of component bindings. The verification finds missing provided component elements connected to the required sides of other components. It furthermore finds missing EFPs on the provided sides matching EFPs on the required sides attached on bound components.

Once the components are bound and the EFPs are in matching pairs, as a result of the algorithm, it is possible to compare values on the EFPs. This step verifies whether a quality demanded on the required side is guaranteed by the EFPs on the matching provided side.

The verification of values must first select a sequence of required-provided EFP pairs from the graph. The sequence $P(V(\overrightarrow{G}), V(\overrightarrow{G})) = \{(v_x, v_y) \mid \forall x \exists y : e_{xy} \in E(\overrightarrow{G}) \wedge v_x \in V_{efp}(\overrightarrow{G}) \wedge v_y \in V_{efp}(\overrightarrow{G})\}$ contains EFP vertexes to be compared. A sequence of EFP values attached to these vertexes is obtained applying the function $value : V(\overrightarrow{G}) \times V(\overrightarrow{G}) \rightarrow T \times T$ where T is a set of EFP value instances computed on respective vertexes.

Furthermore the function $\gamma : T \times T \rightarrow Z$ (Section 2.1, equation 1) compares value pairs returning a numeric result. Taking it together, the sequence of vertex pairs is transformed to a set of numbers.

$$\gamma \circ value : V(\overrightarrow{G}) \times V(\overrightarrow{G}) \rightarrow Z \tag{7}$$

Using the functions, vertexes from the input sequence are finally compared:

$$z_k = \gamma_k(value_k(P(V(\overrightarrow{G}), V(\overrightarrow{G}))_k)),$$
$$k = 1..|P(V(\overrightarrow{G}), V(\overrightarrow{G}))|$$

The resulting sequence of numbers is checked. A non-negative number means that a quality has been satisfied. For that reason the evaluator verifies that $\forall k \exists z_k : z_k \in [0, \infty) \subset Z$ holds. Otherwise the evaluator signals an error for the EFP wrapped in the respective vertex.

For instance, let us assume the property `time_to_process` with numeric values and a γ function $\gamma(x, y) = x - y$ (shorter processing time is better). Following Figure 3 with values assigned to vertexes $v_p := 10$ and $v_o := 30$ the evaluation returns $\gamma(value(v_p, v_o)) = \gamma(value(10, 30)) = 10 - 30 = -20$. The result of the evaluation for these vertexes leads to incompatible EFPs. For different values $v_p := 40$ and $v_o := 30$ the evaluation would succeed.

4   Application to Industrial Frameworks

This part will demostrate the presented approach on the Spring IoC Container and the OSGi framework. They have been selected as two widely used component frameworks with no EFP support to show the strenghts of the proposed EFP approach to enrich existing systems with EFPs.

4.1 Spring IoC Container

Components in Spring [13] have forms of so called Beans expressed as Java classes with dependencies written in configuration XML files[1] together composing Application Context.

One of the approaches to connect communicating Beans is by setters. For that reason each setter of a Bean denotes the required side of the Bean and the binding of the provided to required side is equivalent to the examination of values (objects) injected into the Bean.

Since Spring does not handle extra-functional properties, they must be explicitly added by the EFP framework. The EFP Assignment module must therefore be extended to attach EFPs to Spring Beans. This may for instance be achieved by extending Spring's XML configuration files using additional XML name-spaces to include the EFP declarations – this way, new XML tags do not clash with the existing ones and the configuration separates concerns. We suggest a solution in which the EFP data are mirrored in a stand-alone XML file and the links between the mirror and Spring Beans are stored in the extended Spring XML files.

Here is an example of the solution based on the extended XML files:

```
<bean id="data" class="cz.zcu.kiv.example.DataAccess" >
  <property name="jdbc" ref="jdbcDriver" />
  <efp:name="response-time" property="jdbc" gr_id="1">
    <efp:values>
      <efp:lr id="1" value="average" />
      <efp:direct value="100" />
    </efp:values>
  </efp:name>
</bean>
```

In order to evaluate EFPs attached on Spring Beans, the EFP evaluator may obtain Bean binding directly from the container life-cycle using so called Bean Post Processors. It provides developers with a rich spectrum of call-back methods allowing to observe the container life-cycle.

For the purposes of component matching, the *InstantiationAwareBeanPostProcessorAdapter* class is useful to monitor bindings of Bundles with one another. Its implementation prepares the pairs of matching Beans for the μ function when other Beans are injected into the current one.

An application of the EFP Evaluator for Spring is straightforward. Using the strategy with Bean Post Processors, the evaluator is invoked as a new Bean is instantiated first, then the attached EFPs are evaluated (Section 3.2). Depending on particular needs, the evaluator can be invoked for each change in the Application Context or only once when the system starts. Any errors found in the evaluating process may cause the Application Context to stop as well as the errors to be logged.

---

[1] For the purposes of clarity, this paper targets only the Spring's XML based configuration. Other means such us the Spring's annotation driven configuration is avoided without a loss of generality.

4.2   OSGi

The OSGi framework [14] packs components called Bundles as Java JAR files. Each Bundle consists of a set of services (Java classes) communicating with services of other Bundles. A specification of each Bundle is written in a text form as a part of the manifest file. Bundle services are grouped into packages which in OSGi must be explicitly imported (required) or exported (provided) in the manifest file to allow communication with other Bundles.

Hence, a first option to extend OSGi by EFPs is to supplement the content of the manifest file. For instance, a database layer of an application may be enriched with extra-functional information as follows:

```
Manifest-Version: 1.0
Bundle-Name: Data
Export-Package: cz.zcu.kiv.osgi.example.dao;
  efp:=1.db_engine=LR.2.memory
```

meaning that a property `db_engine` from GR with the identifier 1 has been assigned a name `memory` from the context of a LR with the identifier 2. It is assumed that the meaning of this name is stored separately, in the EFP data mirror attached to the Bundle.

This concept is similar to OSGi *capabilities* (OSGi release 4 specification [14]) which use name-spaces in similar manner to LR value names. However the capabilities lack unification such as provided by the Registries. Moreover both approaches may be too coarse-grained since the provided and the required elements are on the package level. Therefore, this EFP assignment option does not necessarily prevent incompatible services (Java classes in practice) to be run.

Another innovative concept of the OSGi 4 are Declarative Services (DSs). DSs provide Bundles with a fine tuned declaration of particular services stored in XML files. Hence, EFPs can be in detail defined for services using an idea equivalent to name-spaces, developed in this paper for Spring, applied to DSs.

Extending the manifest file with a link to a Declarative Service specification

```
Manifest-Version: 1.0
Bundle-Name: Data
Service-Component: OSGI-INF/dao.xml
```

the `dao.xml` file contains the declaration of one particular service `DataAccess` implemented by a `DAImplHSQL` class. This can be enhanced with EFPs:

```xml
<component name="dao">
<implementation class="cz.zcu.kiv.osgi.app.dao.DAImplHSQL"/>
  <service>
    <provide interface="cz.zcu.kiv.osgi.app.dao.DataAccess"/>
      <efp:name="response-time" gr_id="1">
        <efp:values>
          <efp:lr id="2" value="average" />
        </efp:values>
      </efp:name>
    </provide> </service> </implementation>
</component>
```

According to the principles mentioned already for Spring, the evaluation process of Bundles enriched with EFPs would take part in a component life-cycle. OSGi provides a `BundleListener` which may be used by any Bundle to observe changes (starting, stopping, installing, etc.) of other Bundles. Hence a Bundle invoking EFP framework modules for each Bundle will determine compatibility in the phase of starting or installing other Bundles [15].

5   Related Work

This work has been partly based on our previous research. Namely, the structure of data stored in the EFP repository is an implementation of our formal definitions published in [9]. EFP Types has been implemented using meta-models detailed in [7]. Hence, the repository part of the framework is mostly a complement of our previous work while the other part is a new contribution.

There are a lot of other approaches targeting extra-functional properties. They usually cover a rich spectrum of issues, from formal definitions to practical implementations.

An often addressed issue is the description of extra-functional properties. One of the expressing means are specialized languages, for example CQML [16] that serves as a complete extra-functional language, CQML+ [17] that explicitly takes a runtime environment dependency into account, or NoFun [10] that distinguishes between simple and derived extra-functional properties. Furthermore there exist rather specialized languages such as TADL [18] which is a language describing architectures of systems with a concern of EFPs, HQML [19] as a language targeted to web-development, or the SLAng language suited especially for service-level agreement specifications [20]. A general advantage of such approaches is that they provide an answer of what an extra-functional property should stand for. On the other hand they do not address the question of how the properties should be evaluated. Developing our approach, we use these languages to consolidate typical features of extra-functional properties into our model.

Other works propose component frameworks taking extra-functional properties into account as a part of their component models. Let us name at least Palladio [21] that targets mainly performance characteristics, Robocop [3,4] for real-time characteristics, or ProCom [22]. These approaches typically lack modularity in terms of the peculiar ways of using extra-functional properties which prevents their EFPs to be used in other component frameworks.

Comparing these approaches to our contribution, we aim at a system which is not tied with a concrete component framework, is not intrusive and provides easy integration with other frameworks.

6   Conclusions

This paper has presented a generic approach to address the need for improving current component based development by extra-functional properties. The key contribution of this paper lies in the definition of core EFP model structures and algorithms for working

with EFPs in a comprehensive manner. Furthermore, their implementation in the form of an independent framework has been described which includes a layered repository of extra-functional properties, a module for assigning the properties to components and an evaluator of the properties to determine EFP-based component compatibility.

Moreover, a problem of applicability of extra-functional properties to practice has been identified, in particular the discrepancy of industrial and research component frameworks together with slow adoption of extra-functional properties in practice. The presented approach seamlessly enriches current industrial component frameworks with EFPs and aims at filling the gap between the extra-functional properties research and the practically used component frameworks.

We have created a set of tools to manage the EFP repositories and interact with the Assignment and Evaluator modules of the framework; the implementation is presented in [23]. Future work on the EFP model and framework includes integration with further component models and investigation of its interplay with standard component binding algorithms.

References

1. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. International Series in Software Engineering. Springer, Heidelberg (1999)
2. ISO/IEC: Informational technology – product quality – part 1: Quality model. International standard ISO/IEC 9126, International Standard Organization (2001)
3. Muskens, J., Chaudron, M.R.V., Lukkien, J.J.: A Component Framework for Consumer Electronics Middleware. In: Atkinson, C., Bunse, C., Gross, H.-G., Peper, C. (eds.) Component-Based Software Development for Embedded Systems. LNCS, vol. 3778, pp. 164–184. Springer, Heidelberg (2005)
4. Bondarev, E., Chaudron, M.R., de With, P.H.: Compositional performance analysis of component-based systems on heterogeneous multiprocessor platforms. In: Proceedings of Euromicro conference on Software Engineering and Advanced Applications, pp. 81–91. IEEE Computer Society (2006)
5. Yan, J., Piao, J.: Towards QoS-Based Web Services Discovery. In: Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008. LNCS, vol. 5472, pp. 200–210. Springer, Heidelberg (2009) ISBN: 978-3-642-01246-4
6. García, J.M., Ruiz, D., Ruiz-Cortés, A., Martín-Díaz, O., Resinas, M.: An Hybrid, QoS-Aware Discovery of Semantic Web Services Using Constraint Programming. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 69–80. Springer, Heidelberg (2007)
7. Ježek, K.: A complex meta-model for extra-functional properties concerning common data types their comparing and binding. In: 2nd World Congress on Software Engineering (WCSE 2010), vol. 2, pp. 71–74 (2010) ISBN:978-0-7695-4303-1
8. Szyperski, C.: Component Software, 2nd edn. ACM Press, Addison-Wesley (2002)
9. Jezek, K., Brada, P., Stepan, P.: Towards context independent extra-functional properties descriptor for components. In: Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010). ENTCS, vol. 264, pp. 55–71 (2010) ISSN: 1571-0661
10. Franch, X.: Systematic formulation of non-functional characteristics of software. In: Proceedings of International Conference on Requirements Engineering (ICRE), pp. 174–181 (1998)

11. Snajberk, J., Brada, P.: ENT: A generic meta-model for the description of component-based applications. In: Proceedings of the 8th International Workshop on Formal Engineering approaches to Software Components and Architectures, Satellite event of ETAPS 2011, Saarbrcken, Germany (2011)
12. Bauml, J., Brada, P.: Automated versioning in OSGi: A mechanism for component software consistency guarantee. In: Proceedings of the EUROMICRO-SEAA Conference, pp. 428–435. IEEE Computer Society Press (2009)
13. Spring Comunity: Spring Framework, Reference Documentation. ver. 3 edn. (2010), `http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/`
14. The OSGi Alliance: OSGi Service Platform, Release 4 (2005), `http://www.osgi.org/`
15. Brada, P.: Enhanced OSGi bundle updates to prevent runtime exceptions. In: Proceedings of the 34th Euromicro SEAA Conference. IEEE CS, Parma (2008)
16. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo (2001)
17. Röttger, S., Zschaler, S.: CQML+: Enhancements to CQML. In: Bruel, J.M. (ed.) Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France, Cépaduès-Éditions, pp. 43–56 (2003)
18. Mohammad, M., Alagar, V.S.: TADL - an Architecture Description Language for Trustworthy Component-Based Systems. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 290–297. Springer, Heidelberg (2008)
19. Gu, X., Nahrstedt, K., Yuan, W., Wichadakul, D., Xu, D.: An XML-based quality of service enabling language for the web. Journal of Visual Language and Computing, Special Issue on Multimedia Language for the Web 13, 61–95 (2001)
20. Lamanna, D.D., Skene, J., Emmerich, W.: Slang: A language for defining service level agreements. In: The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems, FTDCS 2003. IEEE Computer Society, Los Alamitos (2003)
21. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82, 3–22 (2009); Special Issue: Software Performance - Modeling and Analysis
22. Sentilles, S., Štěpán, P., Carlson, J., Crnković, I.: Integration of Extra-Functional Properties in Component Models. In: Lewis, G.A., Poernomo, I., Hofmeister, C. (eds.) CBSE 2009. LNCS, vol. 5582, pp. 173–190. Springer, Heidelberg (2009)
23. Ježek, K., Brada, P.: Correct matching of components with extra-functional properties – a framework applicable to a variety of component models. In: Evaluation of Novel Approaches to Software Engineering (ENASE 2011). SciTePress (2011) ISBN: 978-989-8425-65-2

Author Index