
February 8-9, 2001
Enschede, The Netherlands

Edited by Jens H. Jahnke and Conor Ryan

Held at Hotel Golden Tulip De Broeierd

Sponsors

University of Twente
The Netherlands

Computer Science
University of Victoria
Organization Committee

General Chairs
Pim van den Broek, pimvdb@cs.utwente.nl, University of Twente, NL
Mehmet Aksit, aksit@cs.utwente.nl, University of Twente, NL

Local Chair
Richard van de Stadt, stadt@cs.utwente.nl, University of Twente, NL

Program Co-Chairs
Jens H. Jahnke, jens@acm.org, University of Victoria, Canada
Conor Ryan, conor.ryan@ul.ie, University of Limerick, Ireland

Program Committee
Forrest H Bennett III, FX Palo Alto Laboratory, USA
James Boyle, Argonne National Laboratory, USA
Tony Cahill, University of Limerick, Ireland
Peter Eades, University of Newcastle, Australia
Ken Satoh, Hohhaido University of Japan
Georg Lehrenfeld, Heinz-Nixdorf Institute, Germany
Joe Marks, Mitsubishi Electronics Research Laboratories, UK
Steve McDonell, University of Otago, NZ
Tim Menzies, University of BC, Canada
Witold Pedrycz, University of Alberta, Canada
Francesco Marcelloni, University of Pisa, Italy
# Table of Contents

**preface**

## Keynotes

- Computational Intelligence as a Knowledge-Based Environment for Software Engineering  
  *Witold Pedrycz*

- Many Maybes Mean (Mostly) the Same Thing  
  *Tim Menzies, Harhsinder Singh*

## Technical Papers

### Modeling and Reuse

- Coping with Uncertainty in Software Retrieval Systems  
  *Abdelhamid Bouchachia, Roland T. Mittermeir*

- Reasoning with Design Knowledge for Interactively Supporting Framework Reuse  
  *Wolfgang De Meuter, Maja D’Hondt, Sofie Goderis, Theo D’Hondt*

- Minimally Adequate Synthesizer Tolerates Inaccurate Information during Behavioral Modeling  
  *Johannes Koskinen, Erkki Mäkinen, Tarja Systä*

### Metrics

- An Assessment of Metric Contribution in the Construction of a Neural Network-Based Effort Estimator  
  *Gary D. Boetticher*

- Scheduling Multiprocessor Tasks in Presence of the Correlated Failures  
  *Piotr Jedrzejowicz, Ireneusz Czarnowski, Ewa Ratajczak, Walter J. Gutjahr, Aleksander Skakowski, Izabela Wierzbowska, Walter J. Gutjahr, Aleksander Skakowski*

### Program Understanding

- Expressing Similarity in Software Engineering: A Neural Model  
  *W. Pedrycz, G. Succi, M. Reformat, P. Musilek, X. Bai*

- Recovering UML Diagrams from Java Code using Patterns  
  *Jörg Niere, Jörg P. Wadsack, Albert Zündorf*

- Interactive Graph Clustering based on User Hints  
  *Hugo A. D. do Nascimento, Peter Eades*
Requirements and Design
Design and Experimentation of a Fuzzy Logic Controller for Evaluating Domain Knowledge
Bedir Tekinerdogan

Security Concepts for Agents-Based Systems
Georg Lehrenfeld, Wolfgang Mueller, Rainer Tellmann

Delaying Inconsistency Resolution Using Fuzzy Logic
Francesco Marcelloni
Preface

Welcome to SCASE’01 - the second workshop on Soft Computing Applied to Software Engineering.

The phenomenal growth of both the number and size of complex software systems has meant that Software Engineering faces an increasingly difficult task. It is no longer feasible to expect engineers to have a complete mental model of the processes that they are attempting to represent in software.

CASE tools have attempted to aid engineers with this problem, but many have faced scaling problems, as well as issues of dealing with imprecise or even imperfect information. On the other hand, Soft Computing methods are currently enjoying much popularity and success in a wide variety of areas, many of which are characterised by the accuracy issues mentioned above.

Recently, a new field is beginning to emerge, the application of Soft Computing to Software Engineering. It is the goal of SCASE’01 to facilitate not only researchers in this field, but also those from the parent areas. The workshop contains papers on a variety of topics, including Design, Modeling, Effort Estimation and Scheduling, using a diverse array of Soft Computing methods, such as Neural Networks, Fuzzy Logic and Evolutionary Algorithms.

We are sure that the workshop will provide a much needed forum to researchers, as well as acting as an advertisement for the new field of Soft Computing Applied to Software Engineering.

Jens Jahnke & Conor Ryan (Program Co-Chairs)
Keynotes
Computational Intelligence as a Knowledge-Based Environment for Software Engineering

Witold Pedrycz

Department of Electrical & Computer Engineering
University of Alberta, Edmonton, Canada

Systems Research Institute Polish Academy of Sciences
Warsaw Poland

Abstract

The discipline of Software Engineering is abstract and complex with all its endeavors being cast in a knowledge-rich environment. It is not surprising that there have been a number of important initiatives that address a burning need for solid development tools and comprehensive environments supporting an in-depth analysis. The objective of this study is to discuss a role of Computational Intelligence (CI) being viewed as a sound methodological and algorithmic environment for knowledge-oriented Software Engineering. The CI itself is regarded as a synergistic consortium of granular computing (including fuzzy sets) promoting abstraction, neurocomputing supporting various learning schemes and evolutionary computing providing important faculties of global optimization. By its very nature, CI embraces a diversity of design paradigms; in particular it promotes a top-down approach (when exploiting fuzzy sets first and afterwards working in the neural network environment) or bottom-up style (where these two technologies are used in a reverse order).

Software Engineering and CI are highly compatible: they are knowledge-intensive, human-oriented, and have to deal with various manifestations of the abstract world of software constructs and thought processes. This multifaceted conceptual compatibility is a prerequisite for the development of vital synergistic links that bring the technology of CI into Software Engineering. The symbiosis accrues considerable benefits for both technologies by posing new categories of challenging and highly stimulating problems. We highlight a number of selected and most visible trends occurring at the junction of CI and Software Engineering.

1. Introductory Remarks

The research agenda of Software Engineering is ambitious: we are after building high quality software products and develop better software processes. The difficulties in accomplishing this main goal are substantial. In an attempt to pursue fruitful developments it is beneficial to identify sources of such difficulties. The bottom line is this: products and processes of Software Engineering are very human-intensive while at the same time they deal with high level abstract concepts and result in constructs whose functioning is not governed by the laws of physics. There is no direct use of ideas such as continuity that is very intuitive and helpful in the physical world. In Software Engineering, small changes to the requirements may result in far drastic and radical changes in a total cost of the overall project. There is no concept of time as such as software products do not wear out in contrast to physical systems that are subject to such deterioration.

There are two main camps of thought in Software Engineering. The qualitative one focuses on qualitative findings and attempts to encapsulate and organize the best practices and descriptive models already found in the area. The quantitative one attempts to focus on numeric experimental data and build numeric models of software processes and software products. The qualitative approach is legitimate as data in Software Engineering especially those numeric, are scarce. They usually come from various projects that make such experimental evidence more difficult to interpret and exploit in the form of some quantitative and tangible models. The qualitative approach to Software Engineering can be viewed as knowledge-based oriented (no matter in which level of specificity this knowledge becomes available). It is quite dominant when it comes to the managerial aspects of Software Engineering. The
quantitative approach is very much data driven, leads to highly operational models. The models need to be developed in a prudent fashion and one has to be fully cognizant of their limitations and a range of possible applicability. Once used carefully, they form an indispensable analysis and design vehicle. The data – knowledge duality permeates the entire area of studies in Software Engineering. We may anticipate that a fruitful way to proceed with advancements in this area is to determine a viable balance between them that is look at processes and characterize the software products from the qualitative – quantitative standpoint. To accomplish this, we need a new methodology that makes this dual look feasible. In particular such framework should support processing in a highly heterogeneous environment (involving data and knowledge). We postulate that CI can serve as a conceptual and algorithmic framework of Software Engineering.

Computational Intelligence, a term coined in the early 90s, is a coherent and symbiotic collection of information technologies, namely fuzzy sets, neural networks, and evolutionary computing. Recently fuzzy sets have become a part of a global picture of granular computing. The objective of this study is to provide a general overview of CI, identify the role of CI in Software Engineering as a useful vehicle in modeling and synthesis and provide with some representative examples.

The organization of the material follows the main scope of the study. First, we identify the main technologies of CI and discuss their main features (Section 2). Next, we proceed with more detailed discussion on neurofuzzy systems that tend to occupy a central place in CI (Section 3). This study reveals how such systems become beneficial in the design of software models. In the three consecutive sections (Section 4, 5 and 6) we present selected case studies. The first one revolves around neurocomputing and shows how self-organizing maps help visualize software data. The second one exploits the technology of fuzzy sets in the development of models of software cost estimation. The third one shows an example of a neurofuzzy system used in modeling software quality.

Even though the study deals with CI, we devote our particular attention to granular computing as an important conceptual backbone of the overall paradigm. While the area is still in a development phase and a lot of issues need to be looked into, there has been an array of vital research pursuits. The list of references is far from being complete; the reader may consult the edited volume [31] that includes a comprehensive bibliography.

2. Defining Computational Intelligence

In this section, we highlight the main features of the contributing technologies of CI [30], namely granular computing, neural networks and evolutionary optimization.

2.1. Granular Computing

The essence of granular computing is to carry out computing that exploits information granules [32][43][44][45][46]. Information granules are regarded as a collection of elements that can be treated together because of their similarity, functional properties or spatial or temporal adjacency. By forming information granules we realize a process of information abstraction. The process of forming information granules comes under the name of information granulation. There are several key conceptual platforms of information granulation such as sets, fuzzy sets, random sets, probability, shadowed sets, rough sets, etc. Let us remind that sets (intervals) are the simplest models of abstraction: we encapsulate elements in a form of a single granule (set). All elements in a set are indistinguishable. Fuzzy sets offer a formal vehicle of expressing an effect of a smooth transition from complete membership to complete exclusion. This helps address issues such as deferring of elimination of design alternatives [1], reducing quantization error [21], tolerating inconsistencies in software design [22], dealing with software process deviations [7], describing software quality [9][37]. Probability helps form information granules that are expressed in the language of frequency of occurrence and come in the form of probability density functions or probability functions.

The size of the information granules can be expressed in many different ways. In general, this is associated with the way in which the information granules are articulated. For instance

- When dealing with sets (interval analysis), the size of an information granule is expressed by its cardinality, card(A) where A is a set under discussion. In other words, card(A) = ∑χA(x) where χA is a characteristic function of A
- In fuzzy sets the above definition needs to be refined as we are faced with a concept of partial membership. This leads to so-called σ-count that is computed as σ(A) = ∑A(x)
When dealing with probabilistic granules that are expressed as probability density functions, their standard deviation can serve as a useful indicator of granularity.

The larger the value of cardinality, \( \sigma \)-cardinality or standard deviation, the lower the granularity of the corresponding information granule. More importantly, with the larger the information granules we exercise a more general, abstract view at the system (say, software process) by hiding details. By subscribing to smaller information granules, we start concentrating on more details of the system.

### 2.2. Neurocomputing in the design of adaptive systems

In a nutshell, neurocomputing is about developing adaptive systems. Neural networks come with panoply of various learning algorithms, both supervised and unsupervised that help implement a nonlinear mapping between input and output variables. Interestingly, most learning occurs at the parametric level meaning that the parameters (connections) of the network are modified while the structure remains unchanged. Neural networks are highly distributed structures. This makes them robust. This also contributes to their black-box behavior: once trained (optimized) the network cannot be easily interpreted.

The applications of neurocomputing in Software Engineering can be found in problems of software testing [3], software cost estimation [13], prediction of software reliability [16][17], software reusability and document classification [16][27][34], software quality [32][33], software cost estimation [40], prediction of test effectiveness [41], classification of software components [29].

### 2.3. Evolutionary Computing as a Vehicle of Global Optimization

The area of evolutionary computing (EC) arose in the 60s and is now considered as a vehicle of population-based global optimization. There are various methods that contribute to EC: genetic algorithms, evolutionary programming, evolutionary strategies, genetic programming, to name a few. The primordial advantage of EC lies in the fact that we can tackle a vast array of problems of structural optimization (say, design of an optimal topology of a neural network) that are otherwise out of reach. The potential of EC in the realm of Software Engineering has been investigated in software testing and test generation [15][28] [39][42], software understanding [38].

### 3. Granular Computing and Neural Networks in CI

Being extremely rich in the spectrum of research endeavors, granular computing and neural networks are highly complementary. The synergy reflects a way in which systems are constructed. The following observations form a crux of our claim:

- Granular computing is aimed at dealing with information granules. Owing to their logic-based fabric, the models are transparent to the designer yet by operating at the certain level of granularity, the details may be hidden (or simply ignored).
- Neurocomputing is aimed at processing numeric data. Due to the distributed character of the resulting constructs, they constitute black box architectures (not transparent to the user/designer) that make them difficult to interpret and gain a better insight into the nature of the phenomenon under study.

Subsequently, dwelling on the synergy of granular computing and neural networks, two different design methodologies are promoted:

- Granular computing supports the top-down design mode: we start with granules of low granularity (that is not very specific) and attempt to refine them in light of more detailed evidence. The design relies quite strongly on accommodating sources of domain knowledge (for instance, human experts). Such designs are knowledge-driven.
- Neural networks operate in the bottom-up design mode: they transform large numeric datasets into a form of nonlinear mapping. The networks are inherently data driven. To assure their success we need representative data. In a nutshell, neural networks “see” the problem through the data generated by it.

The synergy between these technologies comes into play profoundly to the top-down design: being imitated in the setting of granular computing, further refinements leading to more details are realized through neural networks. Depending on the problem, in Software Engineering we encounter a mix of numeric and granular data. The composition of the mix could be different and this implies a different level of contribution coming from granular computing and neural networks. The synergy of this nature gives rise to a plethora of neurofuzzy systems [30] [35]. The level of contribution coming from the side of neural networks and granular
systems depends upon the type of data and knowledge hints being available.

In the series of the ensuing sections we unveil examples of the synergistic interaction of the above technologies as they occur in the realm of Computational Intelligence.

4. Software Cost Estimation

Cost estimation is a process where we are continuously faced with uncertainty. The level of uncertainty (or granularity) varies depending upon the phase of the project. Naturally, at the beginning of the project this level is high (so the granularity of our estimates is low) and this may get more specific (detailed) over the progress of the project.

There has been an array of various models of software cost estimation, to mention only a class of COCOMO models or function point models [12]. There have been a number of neural models exploited in this area. Surprisingly, the factor of uncertainty (granularity) is not taken into consideration. Take, for instance, a standard COCOMO model in its generic form (our intent is not to move to the details of the model but rather highlight the issue). The model expresses effort, \( E \), given a size of the software \( K \) (described in K lines of code) that reads as

\[
E = aK^b
\]

where “\( a \)” and “\( b \)” are two parameters of the model. Their values depend upon the type of the software (that is embedded, semi-detached or organic).

However, the size of the code is, particularly at the beginning of the project, a matter of rough estimation (e.g., based on some previously completed projects that resemble the current one). Obviously, correctness and precision of such estimates are limited. It is critical to recognize this situation and come up with a technology using which we can quantify the associated imprecision residing within the final results of cost estimation. Fuzzy sets can be used as convenient descriptors of this situation. Two commonly used classes of fuzzy sets are depicted in Figure 1.

Given the size of code represented as a fuzzy set, the calculations of effort follows the extension principle [35][46] that is the core part of the computational environment of fuzzy sets. The detailed formula reads as

\[
E(e) = \sup_{x \in \mathbb{R} : e = ax^b} [K(x)]
\]

where \( E \) is a fuzzy set defined in the space of effort. We can rewrite the above by computing the supremum. This leads to the formula

\[
E(e) = \sup_{x \in \mathbb{R} : e = ax^b} [K(x)] = K((\frac{e}{a})^{1/b})
\]

An example showing the fuzzy set of effort vis-à-vis various levels of granularity of the size estimate is visualized in Figure 2.

![Figure 2. Fuzzy set of size of code \( K \) and associated fuzzy set of effort \( E \). The changes in granularity of \( K \) are reflected in the granularity of effort \( E \)](image)

It is worth stressing that the calculus of information granules is omnipresent in other areas of Software Engineering. For instance, when computing risk exposure, \( RE \), we consider it be a function of Risk_severity and Time_frame both of these being quantified as information granules, cf [14, p.93]. For the first we have three of them, namely low, moderate, high. The time frame is granulated as short, medium, and long. The quantification with the use of fuzzy sets can be helpful in function point software effort estimation models [12].
5. Neural Models of Visualization of Software Data

This case study is concerned with the use of the technology of neural networks, more specifically self-organizing maps [18][19]. We are interested in visualization of relationships between software modules where each of them is described through a collection of software measures [6][47]. Self-organizing maps complete unsupervised learning. The visibility of the revealed structure in data is crucial for any system supporting exploratory data analysis. It is worth stressing that statistical analysis of software data may be faced with two limitations. First, the assumptions being made in most statistical tests may not hold. Second, the results of statistical analysis are very compact and could hamper any possibility to gain a more thorough, detailed and user-friendly description of the data and relationships between various factors (software measures) existing there. For instance, correlation analysis returns a single numeric quantification (correlation coefficient) describing strength of dependency between two software measures. This type of finding is very compact yet hides a lot of details the designer may be interested in. Is the strength of relationship uniform across the entire range of numeric values of the software measures? If not, how does it look like? The role of visualization environment is immense: it conveys a lot of essentials in an organized, coherent fashion that can be easily comprehended by the system designer. In the undertaken study we investigate a collection of objects coming from a C++ project that are characterized by standard OO software measures [8] as summarized in Table 1.

<table>
<thead>
<tr>
<th>Software Measure</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Lines Of Code</td>
</tr>
<tr>
<td>NOM</td>
<td>Number Of Methods</td>
</tr>
<tr>
<td>NOC</td>
<td>Number Of Children</td>
</tr>
<tr>
<td>DIT</td>
<td>Depth Of Inheritance Tree</td>
</tr>
<tr>
<td>CBO</td>
<td>Coupling Between Objects</td>
</tr>
<tr>
<td>RFC</td>
<td>Response For Class</td>
</tr>
<tr>
<td>LCOM</td>
<td>Lack Of Cohesion Of Methods</td>
</tr>
</tbody>
</table>

Table 1. A list of software measures describing software modules

The concept of a self-organizing map (SOM) has been originally coined by Kohonen [18]. As usually emphasized in the literature, SOMs are regarded as neural structures (neural networks) formed as a grid of artificial neurons that visualize highly dimensional data in a low-dimensional structure, usually emerging in the form of a two- or three-dimensional map. To make such visualization meaningful, an ultimate requirement is that such low-dimensional representation of the originally high-dimensional data has to preserve topological properties of the data set. In a nutshell, this means that two data points (patterns) that are close each other in the original feature space should retain this similarity (or closeness) when it comes to their representation (mapping) in the reduced, low-dimensional space in which they are visualized. The two-dimensional map is shown in Figure 3.

![Figure 3. SOM as a grid of processing elements (neurons)](image-url)

The underlying learning principle is straightforward: we determine a winning neurons (namely the one that is the closest to the input data) and award it by modifying its connections. The neuron with the shortest distance (d) between the input and the connections becomes activated to the highest extent and is called a winning neuron. Let us denote its coordinates (position on the map) by (i0, j0). More precisely, we have

\[
(i_0, j_0) = \arg \min_{(i, j)} d(w(i, j), x)
\]

The winning neuron matches (responds to) x. As a winner of this competition, we reward the neuron and allow it to modify the connections so that they are even closer to the input data. The update mechanism is governed by the expression

\[
w_{new}(i, j) = w(i_0, j_0) + \alpha \Phi(i, j, i_0, j_0)(x-w(i, j))
\]

where \(\alpha\) denotes a learning rate, \(\alpha > 0\). The higher the learning rate, the more intensive updates of the connections. In addition to the changes of the connections of the winning node (neuron), we allow this neuron to affect its neighbors (viz. the neurons located at similar coordinates of the map). The way in which this influence is quantified is expressed via a neighbor
function $\Phi(i, j, i_0, j_0)$. In general, this function satisfies two intuitively appealing conditions: (a) it attains maximum equal to one for the winning node, $i = i_0, j = j_0$ and (b) when the node is apart from the winning node, the value of the function gets lower (in other words, the updates are less vigorous). The learning is repeated until no significant changes in the connections of the neurons are encountered.

The results of self-organization are shown in Figure 4. The regions of high level of brightness indicate neurons of similar values of the connections. The darker the color, the more profound the differences between the neighboring neurons and less homogeneous the group (cluster) of the classes formed there. The numbers associated with each node (neuron) indicates the number of the data (classes) attached to this particular location in the map.

Another example of the use of SOMs for analysis of software data [34] is shown in Figure 5. Here we are concerned in finding groups in a collection of standard MS-DOS commands. For clustering purposes, each command is described in a multidimensional space of descriptors such as file, disk, directory, copy, compare, info, pipe, edit, delete, undo, low-level, display, time, substitute, change. The 10 by 10 map, shown in Figure 5, was created using 8,000 iterations with a learning rate set up to 0.4 (which decreased linearly in time), and an initial neighborhood size of 5 nodes (which also decreased linearly over time of training).

![Figure 5. SOM visualization of the data set](image)

The results are highly convincing and easily interpretable. One can easily see that similar commands such as copy and xcopy are grouped in the same area of the map. Similarly, some other groups are also intuitively appealing.

### 6. Neurofuzzy Model of Software Quality

This study is based on the McCall hierarchical software quality measurement framework [23][24][25]. High level approaches toward modeling software quality are based on decomposing quality attributes. This methodology emphasizes the user's view of the final product (namely, executable code) and focuses on an identification of several key attributes of quality stemming from this standpoint. Especially, the resulting hierarchy comprises three well-defined levels:

- **quality factors** (factors) that are high-level software attributes including reliability, usability, etc. In fact, these attributes are usually of too high level to be meaningful and manageable.

- **quality criteria** (criteria, for short) are results of decomposition of the quality factors into lower level internal attributes. These include notions like consistency, accuracy, testability, etc.

- **quality software metrics** comprise the lowest decomposition level.

For example in Fenton [10][11], the factor of maintainability splits into correctability, testability, and...
expansibility. The testability, in turn, splits into a number of software metrics dealing with degree of testing (including statement coverage, branch coverage, path coverage, etc.) and effort (comprising resource prediction and effort expenditure).

The basic components of the model arise in the form of binary Boolean (two-valued) relations summarizing associations between factors and criteria as well as criteria and software metrics. In this study, let the number of factors, criteria, and metrics be equal n, m, and p, respectively. The relations of the model are represented as matrices $A$ and $B$, $A = [a_{ij}], i=1, 2, ..., n, j=1,2, ..., m; B= [b_{ij}], i=1, 2, ..., m, j=1, 2, ..., p$. The calculations of the criteria based on the available factors employ the standard min-max composition (denoted here by •). The vector (set) of criteria $c$ is determined as

$$c = f \bullet A$$

Being more specific, we write this down in the form of the individual coordinates (elements) of the vectors

$$c_j = \min_{i=1,2,..,n} \left( \max_{i} \left( f_1, a_{ij} \right) \right) \quad j=1,2,...,m$$

Similarly, for the given set of criteria $c$, the vector of software metrics ($m$) calculates as

$$m = c \circ B$$

where the symbol $\circ$ denotes the max-min composition. This means that the computations of the coordinates of $m$ are governed by the expression

$$m_j = \max_{i=1,2,..,m} \left( \min_{i} \left( c, b_{ij} \right) \right) \quad j=1,2,..,p$$

The overall architecture can be portrayed with three well-defined layers shown in Figure 6.

![Diagram](image)

**Fig. 6. Software quality model: a general logic structure of a fuzzy neural network**

We can highlight an interesting interpretation of the min-max composition by observing that the links between the given criterion and the associated factors are of an and-like nature [35]

$$\text{criterion}_i = \text{factor}_1 \text{ and factor}_2 \text{ and } ... \text{ and factor}_j$$

Due to the max-min composition [35], the logical relationships between the software metrics (measures) and the criteria are linked or-wise meaning meaning that

$$\text{metric}_j = \text{criterion}_{1} \text{ or criterion}_{2} \text{ or ...criterion}_k$$

The hierarchical model of software quality is inherently normative as a general structure does not necessarily reflect any specificity of a certain software company. In a particular commercial setting characteristic to a certain software company, this model could be overly rigid and usually calls for some additional calibration to reflect many efficacies of the software development environment the model has to cope with. These difficulties could be numerous; let us highlight the two of them:

- system developers may feel more comfortable with some software metrics while discounting the importance of others (this reluctance could be a result of a lack of understanding of the metric, its high complexity, poor software tools supporting its implementation or unsuccessful utilization of the metric in the previous software projects).
- there could be some perception about more detailed numeric quantification of some associations between different components in the matrices; one may feel uneasy about a binary (yes-no) quantification of these dependencies.

These difficulties call for further enhancements of the generic software quality measurement framework. The improvements in the model are to address a descriptive aspect of the model. Such model is achieved by introducing fuzzy sets that allow us to capture graded associations between various objects. The second portion of the agenda (viz., optimization) is realized by providing the model in the form of a fuzzy neural network. Two chief characteristics of this new software quality model are its plasticity and learning ability. The plasticity of the new model is results from the expression of factors and criteria in terms of fuzzy sets, which yield an intuitive, yet precise form to these essential underpinnings in software quality measurements. The learning capability of the new software quality model makes it an ideal candidate to express the relevance of the quality requirements of particular software development environments that developers need to capture.
7. Conclusions

In this study, we have discussed a role of CI in Software Engineering by showing how the main features of CI contribute to the discipline. CI is heterogeneous in the sense of its ability to handle knowledge and data in a coherent manner. CI comes with an orchestrated set of knowledge inclined techniques, learning mechanisms and solid tools of global optimization whose usage in Software Engineering is evident. We discussed three detailed examples looking into a detailed picture of using the fundamental technologies of CI in software problems.

Acknowledgment

Support from the Natural Sciences and Engineering research Council of Canada (NSERC) and the Alberta Software Engineering Research Consortium (ASERC) is gratefully acknowledged.

8. References

27. D. Merkl, Self-organizing maps and software reuse, In [31], 65-95.


Many Maybes Mean (Mostly) the Same Thing

Tim Menzies‡, Harhsinder Singh¶

‡Department of Electrical & Computer Engineering, University of British Columbia,
¶Department of Statistics, West Virginia University

<tim@menzies.com, hsingh@stat.wvu.edu>

Abstract

At the core of soft computing is the intuition that from imprecise knowledge, we can still make reasonable inferences. This paper offers experimental and mathematical evidence for this intuition. Based on a literature review and a newly developed mathematics of "reachability", it is argued that searches through a space containing uncertainties, most of the reachable conclusions will be reached via a small number of "master variables" in a "narrow funnel". Such narrow funnels can be found using very simple randomized search methods. Consequently, when using N soft reasoning tools, it is best to at least start with the simplest and only complicate your soft reasoner if and only if that initial simple tool somehow fails you (and reachability theory predicts that it won’t often fail you).

1 Introduction

Some intuitions, while compelling, may not be correct. For example, consider Zadeh’s intuition that:

... as the complexity of a system increase, our ability to make precise and yet significant statements about its behavior diminishes until a threshold is reached beyond which precision and significance (or relevance) become almost mutually exclusive properties.

–Lofti Zadeh [37]

My own pre-experimental notions was that this intuition was essentially correct. It seemed clear to me that the more we say, the less we are certain on what we say. As theory complexity increases, the certainty of that theory’s assertions decreases as we struggle to fill details which we may never have explored before. One way in which complex theories get imprecise is the presence of “many maybes”; i.e. multiple points where it is unclear which mutually incompatible assertion should be made. This may be as simple as a dispute between different designers over the size of a numeric constant in an equation. Alternatively it may be as complex as a qualitative reasoner that generates innumerable possible conclusions, one for each set of consistent possibilities within a large space of contradictions. In either case, the problem is the same: assertions about some point are contradictory.

However, after reviewing the available evidence, these intuitions must be revised. A repeated observation is that within the current generation of software, many maybes mostly mean the same thing. That is, if we ask software containing contradictory assertions to report on all the ways we achieve certain goals, then there emerge certain goals that are always true, or always false, across the wide space of “maybes”. For these stable inferences, we can make precise and categorical statements in the presence of complex and possible uncertain assertions. Suppose these experimental observations are a general result, and not just a result of a quirky selection of case studies. If so then we have an explanation for the success of fuzzy logic [37], genetic algorithms [11], neural nets [33], qualitative reasoning [20], heuristic programming [6], stochastic inference (e.g. ant intelligence [14], ISAMP [12], HT0 [28] GSAT [32], black-box testing [17]) and many other approximate soft reasoning techniques. These techniques work not because of their intrinsic power, but because many probes across a space of uncertainties will achieve the same result.

This article tests the generality of the experimental observation that many maybes mostly mean the same thing. At issue is how much effort we should spend on the construction of elaborate soft computing tools. We will offer experimental and theoretical evidence that, in the general case, very simple tools such as the random search of HT0 (described below) should suffice for soft computing tasks that reason about a space of uncertainties.
The theoretical case that many maybes mean mostly the same thing is based on the “funnel theory” of Menzies, Easterbrook, Nuseibeh and Waugh [27]. Funnel theory, as presented in §2, has an intuitive appeal and explains the counter-intuitive experimental observations listed in §3. However, until this article, funnel theory had no formal basis. Based on a mathematical argument, it will be shown that we can routinely expect our software to contain narrow funnels. This maths will be presented in two parts. Firstly, in §4, an average case reachability model is presented that computes the odds of reaching some randomly selected part using a theory that contains contradictions. This model has an odd behavior: the number of contradictions per literal does not greatly effect the output of the model. This odd behavior prompted the development a second model. Based on a simulation of an abstract model of funnels, §5 argues that if some conclusion can be reached via a narrow funnel and a wide funnel, then a random search will tend to use the narrower funnel. The argument is recursive: given a narrow funnel and a narrower funnel, random search will favor the narrower funnel over the narrow funnel. Hence, the narrowest funnels act like strange attractors in chaos theory, pulling in all the arguments. Since these arguments will use narrow funnels, there will be few points of disagreement. Hence, the net result of most of the disagreements will be very similar that most maybes will mean the same thing.

2 Funnel Theory

According to funnel theory, arguments within software are pathways through a space of possible contradictions. Each pathway leads to some desired goals and contains a set of assignments to variables. Given a set of goals, if we build proof trees for each goal separately, then it is possible that these proofs will demand different assignments to the same variables. That is, the proofs are contradictory around those variables. The set of variables with contradictory assignments are called the funnel of an argument. The cardinality of this set is a measure of how much the conclusions from this theory can vary. Given and $S$ arguments about the assignments to $N$ variables in the funnel, then there are $S^N$ combinations of proof trees that we can believe at the same time. Depending on which assignment we endorse, different proof trees will be endorsed and different goals will be reachable.

As the funnel size $N$ shrinks, then there are exponentially less different ways to resolve the contradictions in a theory and exponentially less methods for reaching different goals. Funnel theory claims that most searches through a space of contradictory options will lead to the same goals if the pathways cross very narrow funnels. Narrow funnels have two properties suggesting that many maybes will lead to the same consequences. Firstly, narrow funnels dictate how arguments must be resolved around the funnel. If an argument must make it through a funnel in order to reach a goal, then that argument must adapt itself to the shape of the funnel. Secondly, narrow funnels let us ignore certain disagreements. Consider two arguments: one around a narrow funnel and another very peripheral to that funnel. The funnel argument could be resolved quickly since only certain resolutions will pass through the funnel. Further, we need not spend much time on the peripheral argument since it is likely that most pathways will never use that peripheral part of the model.

To understand the effects of funnels consider some ants at the neck of Figure 1 arguing about how to best crawl down to the feet. Each ant’s argument relates to one possible pathway across the skeleton. Note that our search space has funnels: all the pathways must pass through the lumbar spine just above the hips. Our ants might have different disputes about the best way to handle fingers, ribs, and the lumbar spine. Some of these arguments are irrelevant. For example, arguing about how to traverse a finger is irrelevant to the goal of reaching the feet since no pathway through the fingers takes us to the ground without returning to our current position at the neck. Also, with respect to some stated goal, the presence of funnels ensures that some of these arguments only have one possible resolution. For example, suppose one of our ants prefers not to crawl around the lumbar spine since the bones there are too pointy. Given the goal of vertical motion to the feet across the skeleton,
that ant must surrender to the inevitable and travel across the pointy lumbar spine. Clearly, if the software contains the same narrow funnels as Figure 1, then we would expect that the net effect of the contradictory possibilities within that software would be the same.

3 Experimental Evidence

This section reviews empirical evidence that narrow funnels are common in software systems. Elsewhere, Menzies and Cukic [25] have cataloged the number of tests used to certify expert systems. In theory, probing a space to find a bug with probability $10^{-2}$ takes $4.6 \times 10^2$ tests to be 99% sure of finding that event. To show this, note that $N$ randomly selected inputs has certainty

$$C = 1 - ((1 - x)^N)$$

of finding some event with probability $x$. Hence, at a 99% certainty, $C = 0.99$ and this equation becomes:

$$N = \frac{\ln(1 - 0.99)}{\ln(1 - x)} = \frac{-4.6}{\ln(1 - x)} \quad (1)$$

If the space is being probed by a nondeterminate search engine, as often used in a heuristic-based expert system, then $4.6 \times 10^2$ would be a theoretical lower bound on the required number of tests. Nevertheless an often repeated observation is that a small number of inputs can often result in significant errors in a program (see Figure 2). One explanation for this surprising observation is that narrow funnels quickly drive a small number of test cases towards the reachable failures.

Similarly, in conventional software, surprisingly few random probes will detect significant errors in a system. Leveson heuristically applied partial descriptions of software fault trees to ADA code. She claims that this heuristic search detected as many errors in her studied system as a much longer, and much more formal, analysis [22]. If conventional software contained narrow funnels, that would explain how Leveson’s heuristic partial probing was so successful since any argument, generated either by formal or informal methods, would both be sucked towards the funnels.

Another method of probing a system is mutation testing. In mutation testing, a test suite is assessed via its ability to distinguish a program from some mutation of that program. Numerous researchers in mutation testing report that most mutations give rise to the same nominal and off-nominal behaviors [1, 7, 29, 35]. This result can be explained assuming narrow funnels. Mutators are applied randomly and if the funnels are small, it is unlikely that the mutators will stumble across them.

Another reason to believe in narrow funnels is that the overall structure of our programs may not support wide chains of arguments. Bieman and Schultz [3] report that a seemingly complex natural language processing system contains, on average, a small number of narrow du-pathways. A du-path is a link from where a variable is defined to where it is used. Clearly, the upper bound on the number of du-pathways in a program is exponential on the number of program statements. The lower bound on the du-pathways is 1; i.e. the tail of each path touches the head of another path. Figure 3 shows the Bieman and Schultz results: 95.1% of the modules in their system held less that 50 du-pathways. Analogous results has been seen in procedural code. In one analysis of 4000 FORTRAN functions and 3147 “C” functions, the control flow graph of those functions grows linearly with the number of statements [19]. That is, the control-flow diagram forms a single-parent tree. Arguments extracted from single-parent trees would be very narrow indeed.

There is much evidence that the average size of a funnel in AI-based systems is very narrow. Researchers in AI and requirements engineering explore inconsistent theories. A repeated result consistent with narrow funnels is that committing to a randomly selected resolution to a conflict reaches as much of a program as carefully exploring all resolutions to all conflicts. For example, Figure 5 shows Crawford and Baker’s [12] comparison of a standard depth first search backtracking algorithm (TABLEAU) to ISAMP, a randomized search theorem prover (shown in Figure 4). ISAMP randomly assigns a value to one variable, then infers some consequences using a fast forward chainer. After forward chaining, if incomparable conclu-
for TRIES := 1 to MAX-TRIES
{set all vars to unassigned;
loop
{if everything assigned
then return(assignments);
else pick any var v at random;
set v's value at random;
forwardChainFrom(v);
if contradiction
then exit loop;
fi
fi
}
} return failure

Figure 4. The ISAMP algorithm [12]

<table>
<thead>
<tr>
<th>TABLEAU: full search</th>
<th>ISAMP: partial, random search</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Success</td>
<td>Time (sec)</td>
</tr>
<tr>
<td>A</td>
<td>90</td>
</tr>
<tr>
<td>B</td>
<td>100</td>
</tr>
<tr>
<td>C</td>
<td>70</td>
</tr>
<tr>
<td>D</td>
<td>100</td>
</tr>
<tr>
<td>E</td>
<td>80</td>
</tr>
<tr>
<td>F</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 5. Average performance of elaborate search (TABLEAU) vs randomized search (ISAMP) on 6 scheduling problems (A..F) with different levels of constraints and bottlenecks.

Experiments were reached, ISAMP re-assigns all the variables and tries again (giving up after MAX-TRIES number of times). Otherwise, ISAMP continues looping till all variables are assigned. When implemented, Crawford and Baker found that ISAMP took less time than TABLEAU to reach more scheduling solutions using, usually, just a small number of TRIES. Crawford and Baker offer a speculation why ISAMP was so successful: their systems contained mostly “dependent” variables which are set by a small number of “control” variables. Note that this dependent-control theory is consistent with narrow funnels: the small number of control variables are those found in the narrow funnels. TABLEAU failed since it’s rigorous search demanded the resolution of unimportant arguments outside the narrow funnels.

Experiments with randomized multiple-worlds inference engines support the thesis that narrow funnels are common. If multiple worlds of belief are created via very wide funnels, then we would expect a large number of worlds created with each world condoning possibly different inferences. However, if such worlds are created through narrow funnels, then we would see that only a small number of worlds are created. Further, since there are few disagreements between the worlds, it is likely that the created worlds would condone similar inferences. In results consistent with narrow funnels, Menzies, Easterbrook, Nuseibeh and Waugh [27] report that exploring a few set-covering worlds returned nearly as much information as a rigorous exploration of all worlds. That experiment is described below.

The Menzies, Easterbrook, Nuseibeh and Waugh study compared the behavior of two multiple-world reasoners: HT0 and HT4. HT4 generates all pathways from inputs to

\%test 'ors' in a random order
X or Y :- maybe -> (X;Y); (Y;X).

\%test 'ands' in a random order
X and Y :- maybe -> (X,Y); (Y,X).

\% Assuming that an object O’s attribute
\% A is X is legal if this assumption does
\% not conflict with previous assumptions.
\% Otherwise, make assume that O.A=X but
\% remove it if we backtrack to this point.
A of O is X :- a(A,O,Old), !, Old = X.
A of O is X :- assert(a(A,O,X)).
A of O is X :- retract(a(A,O,X)), fail.

\% N times, zap assumptions, try the goal list.
ht0(0,_) :- !.
ht0(N0,G0) :- rememberBestCover(G0),
retractall(a(_,_,_)),
\% Goals with lower weights
\% are tried first
sort(G0, G1),
maplist(prove,G1,G),
N is N0 - 1,
ht0(N,G).

\% Lower/raise a goal’s weight by a random amount
\% if it fails/works respectively.
prove(In/Goal,Out/Goal):-
X is 1 + random(10^3)/10^6,
(call(Goal) -
Out is In+X; Out is In-X).

\% E.g: 5 times, random search for "sad" or "rich".
:- ht0(5,[1/sad,1/rich]).

Figure 6. HT0, simplified (handles acyclic ground theories only). The full version contains many more details such as how variables are bound within \texttt{rand}s and the implementation of \texttt{rememberBestCover}. For full details, see [28].

\%test `ors' in a random order
X or Y :- maybe -> (X;Y); (Y;X).

\%test `ands' in a random order
X and Y :- maybe -> (X,Y); (Y,X).

\% Assuming that an object O’s attribute
\% A is X is legal if this assumption does
\% not conflict with previous assumptions.
\% Otherwise, make assume that O.A=X but
\% remove it if we backtrack to this point.
A of O is X :- a(A,O,Old), !, Old = X.
A of O is X :- assert(a(A,O,X)).
A of O is X :- retract(a(A,O,X)), fail.
goals and sorts them into consistent worlds of belief [23]. HT0 just returns the first world it finds randomly [28]. HT0 randomizes the order in which it searches for proofs. During the proof of goal i, when processing a set of goals in a disjunction or a conjunction, the order of the processing is selected randomly (see rand/2, ror/2 in Figure 6). If a proof of goal i fails, the system does not backtrack to retry one of goal i...i−1. Instead, HT0 lowers a weight associated with goal i and moves on to try goal i + 1 (see prove/2 in Figure 6). When HT0 has finished with all the goals, it wipes all the assumptions, sorts the goal list according to the adjusted weights, then tries to prove them all again (see htw/2 in Figure 6). When HT0 and HT4 were run on the same examples, HT4’s runtimes were observed to be exponential while HT0 was less than cubic [28]. Also, and most important for our discussion, the random search of HT0 reaches nearly as many goals as the rigid search of HT4. Menzies, Easterbrook, Nuseibeh and Waugh executed thousands of models using HT0 and HT4. To generate these models, mutators would corrupt influences in a theory; e.g. proportional signs were flipped to inversely proportional and visa versa, influences were added at random, and less and less data was offered to the reasoner. In a result consistent with most maybes mean the same thing, the average difference in covered goals between the random partial search of HT0 and the rigorous search of HT4 was less than 6% (see Figure 7).

4 Generalizing HT0 with Reachability Theory

Did HT0 work because of quirks in its case study? Or was it an example of a general principle? This section argues that HT0’s results are quite general: the average odds of reaching a goal across a space containing contradictions is quite high. These average-case odds can be calculated using the reachability analysis [26] described below.

Reachability studies the odds of reaching a random part of NAYO graphs like Figure 8. Such NAYO graphs contain No-edges, And-nodes, Yes-edges, and Or-nodes. Yes-edges denote valid inferences and no-edges denote incompatibilities “maybes”. The V nodes of a NAYO graph are divided into and-nodes and or-nodes with ratios and f and or f respectively (and f + or f = 1). In the reachability model, and-nodes and or-nodes have mean parents and p, or p respectively. Or-node contradict, on average, no other or-nodes. and p, or p, no are random gamma variables with means and f, and f, or f, no: “skews” and p, or p, no; and range 0 ≤ γ ≤ ∞. and f is a random beta variable with mean and f and range 0 ≤ β ≤ 1. And-nodes are reached at height j via one parent at height i = j − 1 and all others at height:

\[ i = \beta(depth) * (j - 1) \]  

so 0 ≤ i ≤ (j − 1). Note that as depth decreases, and-nodes find their pre-conditions closer and closer to the inputs.

The probability \( P[j]_{and} \) of reaching an and-node at height \( j > 0 \) is the probability that one of its parents is reached at height \( j − 1 \) and the rest are reached at height \( 1.(j − 1) \); i.e.

\[ P[j]_{and} = P[j - 1] * \left( \prod_{2}^{\text{and p}[j]} P[i] \right) \]  

Or-nodes are reached at height j via one parent at height \( i = j - 1 \). The probability \( P[j]_{or} \) of reaching an or-node at height \( j > 0 \) is the probability of not missing any of its parents; i.e.

\[ P[j]_{or} = 1 - (1 - P[j - 1]) * \left( \prod_{2}^{\text{or p}[j]} (1 - P[i]) \right) \]  

From \( P[j] \), we compute \( P[j]_{or} \) by modifying \( P[j] \) with two factors: one for the odds of not entering into an inferring loop, and one for the odds of not causing contradictions:

\[ P[j]_{no contradiction} = \left( 1 - \frac{n}{f} \right)^{n[j]_{or f}} \]
diet(fatty).
diet(light).

happy :- tranquility(hi); rich, healthy.
healthy :- diet(light).
satiated :- diet(fatty).
tranquillity(hi) :- satiated; conscience(clear).

Figure 8. A NAYO graph (shown right) connecting terms within some theory (shown left).

\[ P[j]_{\text{no loop}} = \left(1 - \frac{1}{V}\right)^{n[j]_{\text{orf}}} \]  
where \( n[j] \) is a guesstimate of the size of the proof tree to depth \( j \). Observe the use of \( n[j] \) * orf \ in Equation 5 and Equation 6. And-nodes contradict no other nodes; hence we only need to consider contradictions for or f of the system. Also, since every and-node has an or-node as a parent, then we need only check for loops amongst the or-nodes.

The probability \( P[j] \) of reaching any node is hence the sum of \( P[j]_{\text{or}} \) and \( P[j]_{\text{and}} \) weighted by the frequencies of and-nodes and or-nodes; i.e.

\[ P[j] = \text{and}f \ast P[j]_{\text{and}} + \text{orf} \ast P[j]_{\text{or}} \]  
\[ P[j]_{\text{or}} = P[j]_{\text{or}} \ast P[j]_{\text{no loop}} \ast P[j]_{\text{no contradicti}} \]  

A simulation of the above system of equations is around 200 lines of Prolog. This model can be executed to generate \( P[j] \). From this figure, we find the number of tests \( N \) required to be \( C = 99\% \) percent certain of reaching a random node in a dependency graph using Equation 1.

The above model was run for a wide range of input parameters; e.g. up to \( 10^6 \) nodes, up to 1000 inputs, wildly varying the frequency and skew of and-nodes, or-nodes, and no-edges, etc. The frequency distribution of the generated \( N \) values is shown in Figure 9 divided according to the \( j \) (proof height) value. The simulation results shows that HT0’s success was not a quirk of the models in its domains. Rather, if we explore a NAYO graph to more than a shallow depth \( j > 50 \) then in the usual case, we can reach most parts of that theory with small number of random inputs.

The results of the simulation of the reachability model reproduce the HT0 results: despite the presences of contradictory options in a theory, the odds of reaching a goal can be quite high. However, the reachability analysis raises more questions than in answers. A strange feature of the reachability model is that its conclusions are barely effected by the number of contradictions in a theory. This lack-of-effect was detected using a machine learner (C4.5 [30])

and the following sensitivity analysis. Outputs from the reachability model can be classified using Equation 1 as follows:

\[
\text{Class} = \begin{cases} 
\text{fast and cheap} & \text{if } N < 10^2, \\
\text{fast and moderately expensive} & \text{if } N < 10^4, \\
\text{slow and expensive} & \text{if } N < 10^6, \\
\text{impossible} & \text{otherwise}.
\end{cases}
\]

Decision trees to predict these classifications were built using three different subsets of the model parameters (see Figure 10). For each subset, learners were given example sets of different sizes: 150 examples, 1500 examples, and

Figure 9. Some frequency distributions of the number of tests required to be 99\% sure of reaching a node at height \( j \) generated from the Menzies-Cukic-Singh reachability model.
Some Least

Table: Parameters used in the reachability model, divided into three sets: All, Some, Least.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>All</th>
<th>Some</th>
<th>Least</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</td>
<td>height</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>depth</td>
<td>mean relative parent height ($\beta$)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\alpha p$</td>
<td>$\gamma(\alpha p)$ mean</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\alpha d$</td>
<td>$\gamma(\alpha d)$ mean</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\alpha r$</td>
<td>$\gamma(\alpha r)$ mean</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\alpha n$</td>
<td>$\gamma(\alpha n)$ skew</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\alpha o$</td>
<td>$\gamma(\alpha o)$ skew</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\alpha s$</td>
<td>$\gamma(\alpha s)$ skew</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$t_1$</td>
<td>number of inputs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$v$</td>
<td>number of nodes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>classes</td>
<td>1...10 or 10^2...10^3 or 10^4...10^6 or 10^6...∞</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 10. Parameters used in the reachability model, divided into three sets: All, Some, Least.

150000 examples. A baseline for classifier accuracy was generated by building a classifier using All 13 model parameters. A nearly similar classifier could be built by ignoring numerous parameters (evidence: compare the All curve to the Some curve in Figure 11). However, if we blocked access by the machine learner to andf, the accuracy of the classifier fell by 15-20% (evidence: compare the Some curve to the Least curve in Figure 11). Hence, the parameters included in All-Some are clearly not the main determiners of reachability. Note that the set All-Some includes information about the frequency of contradictions in theory ($n\alpha o_1, n\alpha o_2$).

The lack of sensitivity of reachability to contradictions is most strange. The pre-experimental intuition was that as the number of contradictory possibilities increase, then the odds of reaching a particular conclusion should decrease. This intuition turns out to be incorrect, both in specific examples shown in §3 and in the general case described by the reachability model. To explain this insensitivity, we must return to funnel theory.

5 Formal Funnel Theory

A formal analysis of funnel theory explains why the odds of reaching some randomly selected part of a theory is barely affected by the number of contradictions in that theory. In this section, a mathematical simulation demonstrates that given the choice of a narrow or a wide funnels to reach a goal, a random search engine will select the narrow funnel. That is, even if a theory supports many arguments, randomized search will favor the less contentious parts of a theory.

Suppose some goal can be reached by a narrow funnel $M$ or a wide funnel $N$ as follows:

$$\begin{align*}
\text{narrow} & = c \prod_{i=1}^{m} a_i \\
\text{wide} & = d \prod_{i=1}^{n} b_i
\end{align*}$$

Under what circumstances will the narrow funnel be favored over the wide funnel? More precisely, when are the odds of reaching goal via the narrow funnel much greater that the odds of reaching goal via the wide funnel? To answer this question, we begin with the following definitions. Let the $M$ funnel use $m$ variables and the $N$ funnel use $n$ variables. Each member of $M$ is reached via a path with probability $a_i$ while each member of $N$ is reached via a path with probability $b_i$. Two paths exist from the funnels to this goal: one from the narrow neck with probability $c$ and one from the wide neck with probability $d$. The probability of reaching the goal via the two pathways is:

$$\begin{align*}
\text{narrow} = c \prod_{i=1}^{m} a_i \\
\text{wide} = d \prod_{i=1}^{n} b_i
\end{align*}$$

For comparison purposes, we express the size of the wider funnel as a ratio $\alpha$ of the narrower funnel; i.e.

Figure 11. Accuracy of classifiers learnt using the All, Some, Least sets defined in Figure 10. Note that the error in the classifier is not changed much by using the Some set.
Assuming that the goal is reached, then there are three ways to do so. Firstly, we can reach the goal using both funnels:

\[ n = \alpha m \quad (11) \]

Secondly, we can reach the goal using the narrow funnel and not the wider funnel:

\[ \text{narrow} \land \neg \text{wide} = \text{narrow}(1 - \text{wide}) \quad (13) \]

Thirdly, we can reach the goal using the wider funnel and not the narrow funnel.

\[ \neg \text{narrow} \land \text{wide} = (1 - \text{narrow})\text{wide} \quad (14) \]

Let \( g \) be probability of reaching \( \text{goal}_i \). Clearly, \( g \) is the sum of Equation 12, and Equation 13, Equation 14; i.e.

\[ g = \text{narrow} + \text{wide} - \text{narrow.wide} \quad (15) \]

The odds of an event with probability \( P(X) \) is the ratio of that event to its complement; i.e. \( \frac{P(X)}{1 - P(X)} \). Hence, the odds of Equation 16 is:

\[ \text{Odds (narrow|g)} = \frac{\text{narrow}}{1 - \left( \frac{\text{narrow}}{\text{narrow + wide - narrow.wide}} \right)} \]

\[ = \frac{\text{narrow}}{\text{wide}(1 - \text{narrow})} \quad (18) \]

Similarly, the odds of Equation 17 is:

\[ \text{Odds (wide|g)} = \frac{\text{wide}}{\text{narrow}(1 - \text{wide})} \quad (19) \]

We divide Equation 18 by Equation 19 to compute the ratio \( R \) of the conditional odds of reaching \( \text{goal}_i \) via the narrow or the wide funnel:

\[ R = \frac{(\text{narrow})^2(1 - \text{wide})}{(\text{wide})^2(1 - \text{narrow})} \quad (20) \]

Our pre-condition for use of the narrow funnel is:

\[ R > 1 \quad (21) \]

In general, using the narrow funnel is much more likely if \( R \) is very large, i.e. bigger than some threshold value \( t \)

\[ R > t \quad (22) \]

where \( t \) is some number much larger than 1.

We can now define a procedure for finding situations when a random search engine will favor narrow funnels over wide funnels:

- For a wide range of values of \( a_i, b_i, c, d, m, \alpha, \ldots \)
- Look for situations when Equation 22 is satisfied.

We apply this procedure below, twice:

- In the first application, we make some simplifying assumptions such as \( a_i \) and \( b_i \) come from uniform probability distributions. These simplifying assumptions let us derive expressions for the ratios of \( c \) and \( d \) that would satisfy Equation 22.
- In the second application, we reject the simplifying assumptions and describe a simulation that handles a wider range of cases.

In both applications, it is clear that if we grow the wide funnel wider, then Equation 22 is often satisfied.

### 5.1 The Uniform Case

Consider the simple case that \( a_i \) and \( b_i \) come from uniform probability distributions, i.e.

\[ \sum_{i=1}^{m} a_i = 1 \]

\[ \therefore a_i = \frac{1}{m} \]

\[ \therefore \text{narrow} = c \left( \frac{1}{m} \right)^m \quad (23) \]

Similarly

\[ \text{wide} = d \left( \frac{1}{n} \right)^n \quad (24) \]

Thus, by Equation 21, narrow funnel is more likely when:

\[ \text{narrow}^2(1 - \text{wide}) > \text{wide}^2(1 - \text{narrow}) \]

which we can rearrange to

\[ (\text{narrow} - \text{wide})(\text{narrow} + \text{wide} - \text{narrow.wide}) > 0 \quad (25) \]

Equation 25 contains two terms, the second of which is Equation 15 which is always positive. Hence, Equation 25
is positive when \( \frac{n_{\text{narrow}}}{n_{\text{wide}}} > 1 \). Substituting in Equation 23 and Equation 24, yields:

\[
\frac{n_{\text{narrow}}}{n_{\text{wide}}} = \frac{c \left( \frac{1}{m} \right)^m}{d \left( \frac{1}{n} \right)^n} \tag{26}
\]

Recall that \( n = \alpha m \), i.e. Equation 26 will hold when:

\[
(\alpha m)^m m^{-m} > \frac{d}{c} \tag{27}
\]

Consider the case of two funnels, one twice as big as the other; i.e. \( \alpha = 2 \). Equation 27 can be rearranged to show that \( \frac{n_{\text{narrow}}}{n_{\text{wide}}} > 1 \) is true when

\[
(4m)^m > \frac{d}{c} \tag{28}
\]

At \( m = 2 \), Equation 28 becomes \( d < 64c \). That is, to access \( \text{goal}_i \) from the wider funnel, the pathway \( d \) must be 64 times more likely than the pathway \( c \). This is not highly likely and this becomes less likely as the narrower funnel grows. By the same reasoning, at \( m = 3 \), to access \( \text{goal}_i \) from the wider funnel, the pathway \( d \) must be 1728 times more likely than the narrower pathway \( c \). That is, under the assumptions of this uniform case, as the wide funnel gets wider, it becomes less and less likely that it will be used.

5.2 The Non-Uniform Case

We have seen that the two assumptions of

1. low threshold value of \( t = 1 \) and
2. uniform probability distributions for the funnel pre-conditions

means that the narrow funnel is far more likely than the wider funnel. This section relaxes these two assumptions to use very large values of \( t \) and wildly varying values for \( a_i \) and \( b_i \). A small simulator is used to compute Equation 22 as follows. The mean \( \mu \) and standard deviation \( \sigma \) of the logarithm of the variables \( a_i, b_i, c, d \) were picked at random from the following ranges:

\[
\mu \in \{1, 2, \ldots 10\} \tag{29}
\]

\[
\text{spread} \in \{0.05, 0.1, 0.2, 0.4, 0.8\} \tag{30}
\]

\( \mu \) and \( \text{spread} \) where then converted into probability as follows:

\[
\sigma = \text{spread} \times \mu
\]

\[
\text{probability} = 10^{-1 \cdot \text{normDist}(\mu, \sigma)} \tag{31}
\]

Figure 12. Outputs from 100000 runs of the funnel simulator. The Y-axis shows what percentage of the runs satisfies Equation 22 as \( \alpha \) increases. On the plot, \( \alpha \) is shown as “alpha”.

Note that this method produces non-uniform probabilities for \( a_i \) and \( b_i \). Next, \( m \) and \( \alpha \) were picked at random from the ranges:

\[
m \in \{1, 2, \ldots 10\} \tag{32}
\]

\[
\alpha \in \{1, 1.25, 1.5, \ldots 10\} \tag{33}
\]

\( R \) was then calculated and the number of times \( R \) exceeded different values for \( t \) is shown in Figure 12. As might be expected, at \( t = 1, \alpha = 1 \) the funnels are the same size and the odds of using one of them is 50%. As \( \alpha \) increases, then increasingly Equation 22 is satisfied and the narrower funnel will be preferred to the wider funnel. The effect is quite pronounced. For example, in 82% of our simulated runs, random search will be 10,000,000,000 times as likely as to use funnels \( \frac{1}{3} \) smaller than alternate wider funnels (see the \( \alpha = 3 \) results).

In summary, in both the uniform and non-uniform case, many maybes mostly mean the same thing. Perhaps the reason for this is as a funnel widens, it becomes exponentially less likely that a random search engine will find all the members of the wider funnel. What ever the underlying cause, the effect is clear: the narrow funnel will usually be favored and the number of arguments that can effect the reachable goals will be reduced.

6 Some Details

This section clarifies some details of this discussion. Our case has been that most maybes mean the same thing, not that all maybes mean the same thing. As shown above in Figure 9, there exist discrete systems for which many maybes do not mean the same thing.
Also, the argument described here relates to the properties of discrete systems containing contradictions. Such an argument may not apply to continuous systems with feedback loops. Continuous systems with feedback loops can generate wildly varying behavior if that system moves into a chaotic region of its state space. Clearly, in systems experiencing such chaos, many maybes will not mostly mean the same thing.

Our emphasis on discrete systems does not preclude the application of this analysis to conventional procedural software. Much research has been devoted to the extraction of discrete models (in the form of finite state machines) from procedural code. For example, the BANDERA system [10] automatically extracts (slices) the minimum portions of a JAVA program’s bytecodes which are relevant to proving particular properties models. These minimal portions are then converted into the finite state machine required for automatic formal analysis. Also, in domains where tools like BANDERA are unavailable, finite state machines can be generated from the high-level documentation describing a procedural system [34].

This article suggests that we can reason about a theory that contains inconsistencies. Such a suggestion might be foreign to students of classical deductive logic in which a contradiction implies anything at all. Classical deduction was a useful tool but in the late twentieth century, many researchers found that non-standard logics were required for inconsistency-tolerant reasoning about (e.g.) model-based diagnosis [9], conflicting requirements [16], or overrides in inheritance hierarchies [15].

The argument made here was that the average number of reachable goal literals are not affected greatly by the presence of contradictory inferences in a theory. This is a statement about where inference pathways end and not about the route taken to a goal. Hence, even when most maybes mean the same thing (i.e. the same number of goals are being reached), an indeterminate device (i.e. one containing contradictions) can take many different pathways to those goals. Consequently, the side-effects of reaching a goal can be very different. If the negation of undesirable side-effects (e.g. not reactor melt down) are added to the goal set, then the argument of this paper will apply and we can quickly check if we can/ cannot reach undesirable side effects.

This analysis assumes a set-covering semantics; i.e. we only consider literals that exist on proof trees between input and goal literals. The opposite to set-covering semantics is consistency-based semantics in which inference spreads out to find all literals consistent with inputs and goals, regardless of whether or not those literals are required for accessing some goals. The debate between set-covering and consistency-based semantics has occurred elsewhere (e.g. [9, 21]). This study favor set-covering semantics since if we are interested in literals outside the goal-finding proof trees, we can add them to our goal set.

7 Conclusion

As theory size or complexity grows, we will become less and less sure about the assertions in that theory. Contradictory options (the “maybes”) will often be entered into theories, particularly if that theory is generated from designers with different views about a domain or the purpose of a program.

An often repeated experimental observation is that a fast random exploration of a program will reach as many interesting goals as a larger number of considered probes. The mathematics of reachability shows us that these observations are not some quirk of particular domains. Rather these observations are examples of a general principle: on average, the way we resolve contradictions does not affect the overall number of reachable goals provided that we are probing into our theories to a non-trivial depth.

These mathematical and experimental results can be explained using funnel theory. Given a choice of $M$ arguments or $N$ arguments ($M < N$) to reach a goal, random search will usually favor the smaller set of arguments. Hence, fewer critical factors will change the number of goals we can reach and most maybes will mean the same thing.

Acknowledgements

The arguments in this article benefitted greatly from numerous lively discussions with Helen Burgess and Steve Easterbrook.

References


Modeling and Reuse
Coping with Uncertainty in Software Retrieval Systems

Abdelhamid Bouchachia, Roland T. Mittermeir
Universität Klagenfurt, Institut für Informatik-Systeme
Universitätsstraße 65-67, 9020 Klagenfurt, Austria
{hamid, roland}@ifi.uni-klu.ac.at

Abstract

We present a software retrieval scheme based on textual documentation written in natural language. The distinguishing feature of this approach is to use the fuzziness inherent in natural language utterances as a feature to smooth differences of descriptions of like concepts made by different individuals. The theoretical basis of this approach, systemic functional theory, is adopted from linguistic research. Concepts thus obtained are related by means of fuzzy sets.

1. Introduction

There is no doubt that software reuse is an important issue in software engineering. Reusing components provides advantages over traditional design from scratch. Increasing productivity, reducing costs for future projects, improving system quality by reusing components that have been successfully used, and increasing reliability by using components already proven correct are some motivations for software reuse. Reuse as a technique can be applied at different steps of the software development life cycle. Much work was done on specification, design and code reuse respectively [5]. In such a context, the problem of software representation, classification, and retrieval turns out to be decisive for the success of the reuse process. This is not only true for early life cycle artifacts (i.e. specifications and design) but also for code, the focus of this paper.

A retrieval system is based on three elements: a model for representing the software components expressed in terms of their functionality, structure, and content, a model for the user’s query which seeking a software component, and a matching function between the component description, and the users’ needs expressed by queries. Therefore, the problem of representation is a very crucial issue. Many methods for representing software components for reuse have been proposed [21]. We mainly distinguish two classes of methods: formal and non-formal ones. The first category includes formal algebraic specifications, signatures and sampling [23]. This category of methods is valued for its precision. To retrieve software components, the user is obliged to provide the system with a precise and well structured information. The first characteristic follows the nature of the technique that serves to describe a software component. In an algebraic framework for example, the user’s need should be well defined after what the retrieval system performs a systematic matching (generally deduction based) between the user’s need and the content of the software library. This task allows only for exact matching. In contrast, informal methods rely on the use of natural language. Representatives include hypertext [12], knowledge bases [26] and information retrieval [19] techniques. Such techniques are more familiar to most users and are more flexible. Although the use of natural language poses problems, it is the means that can be applied indistinctly within all levels of reuse (specifications, design, and code).

Software retrieval is a special case of the general information retrieval problem. We have to consider both, the textuality of software as well as its executional properties [2]. However, the strict textuality of code written in some programming language seems inadequate. Besides being too low at the level of representation, it represents already a tight intertwining of the problem (functionality) and its solution (structural and algorithmical properties). As it is generally the case in the reuse context, if we focus mainly on functionality and some non-functional properties not on algorithmicity, high level descriptions that match in their expressivity thoughts of potential users are more adequate. Software documentation, represented as natural language text, serves well for this purpose. It is rather
free of representational or algorithmic details and it is presented at a high level of abstraction. It also has the benefit that the unspecificity of natural language might in fact serve the purpose of smoothing out different abstraction strategies of developer(s) and users describing the same functionality. Thus, a vice can be turned into a benefit, a benefit that no representation based on a formal language would allow.

With a rich library, the user cannot oversee all its contents. Hence, to perform exact matching would be off the point. On the other side, the user does not always seek for a perfect component, s/he might adapt software components that are close to her/his need or s/he might adapt her/his needs to available components. In software integration, the user disposes two options: black-box reuse where s/he integrates the retrieved component in the current project without any modification and white-box reuse where s/he may bring necessary modifications to the retrieved component before its integration. Therefore, the system should perform partial matching based on similarity. Following this line of arguments, a software retrieval system based on natural language must support the imprecision inherent in user need's specification, and the ambiguity related to natural language itself.

The paper is structured as follows: In section 2, we review the use of natural language for software retrieval. The significance of uncertainty in such a domain is presented in section 3. Section 4.1 highlights the problem of document representation. The systemic functional theory is explained in section 4.2. Uncertainty handling is discussed in section 4.3 with respect to the suggested theory. Then, the notion of “hierarchical fuzzy sets”, how they support the retrieval process, and how they are implemented are presented.

2 Natural language for Software Retrieval

A natural language based system is a system that exclusively relies on linguistic knowledge to describe software components. Narrative software descriptions are treated as documents in an information retrieval system. We have already outlined some motivations for the use of natural language to manipulate software components. Furthermore, people in general prefer to communicate their needs using their native language. Formal languages are always tiresome to learn. To communicate with a retrieval system, the user would be constrained by rigid syntax. Expressing oneself in a formal language is generally a difficult task demanding precision. In contrast, natural language offers the user more freedom to express her/his need. Furthermore, even if the users are familiar with formal descriptions, the problem of adequately organizing the content of a software library is quite hard for evolving libraries.

The basic problem with software retrieval is the accuracy in identifying the information that serves to index the library. The simplest approach is to use just keywords for indexing. The related literature in this domain favours two indexing methods: controlled and uncontrolled. With controlled (or manual) indexing, software descriptions are understood before index terms are assigned. This idea was generalized by Prieto-Díaz under the name of faceted classification. A facet is a set of concepts from which the components are viewed. Each facet includes a list of keywords. With controlled indexing, a constrained terminology is required; otherwise inconsistencies will be produced especially if the library is large and many persons are responsible for the indexing operation. Maintenance of this terminology is too cumbersome though for an evolving area. Uncontrolled (or free-text) indexing is based on the vector space model and uses the frequency of words in natural language descriptions as indicator of relevance of the content to a given topic without any semantic interpretation. The indexing terms are determined automatically in four steps: All the words in the document are tokenized, filtered using a stop list (which contains “meaningless” words such as articles, prepositions, and common verbs), then stemmed using some stemmer, and weighed using the frequency of occurrence of each stem.

Considering efficiency, free-text indexing technique has been proven to be more efficient than the controlled indexing vocabulary technique. This conclusion has had an important impact on the current research in this area.

3 Significance of Uncertainty in Software Retrieval

Uncertainty is an important aspect of any system dedicated to complex human interaction, hence also to software retrieval systems. When interacting with the system, users tend to expect the machine understands their requests in a flexible way. Dealing with human abilities is an ultimate goal for current interactive man-machine systems. Thus, the discussion about uncertainty becomes increasingly relevant whenever we want to model the human linguistic and mental reasoning abilities.

Uncertainty mainly appears under two forms [14]:

---

1In the rest of the paper, we use “software description” and “document” indistinguishably.
ambiguity and fuzziness. Ambiguity is related to the fact of being uncertain and doubtful about meaning or intention of the statement. From the software retrieval perspective, a user has generally a vague idea about the content of the library. Although s/he searches for some component fitting to a given need, initially s/he does not require high precision. The user generally tends only to approach the pre-conceived solution in a converging process.

Ambiguity is also related to the fact of admitting more than one meaning. For the same query, many alternatives (i.e. components) turn out to satisfy the user's need. Mathematically, this reflects a "one to many"-relation. This leads to constrain the system's matching behavior, so that it should perform non-exact matching (or best matching) based on some similarity measure.

The concept of fuzziness is identified with the lack of a sharp distinction between elements of a population. It also refers to the notion of "unclear", "ill-defined" or "confused". If projected to the software retrieval context, we can interpret it from two angles. First, the system is generally faced with some unclear or incomplete query specification. Therefore, it is constrained to perform adequate inferences in order to clarify or to complete the user's need. Second, to alleviate the problem of distinction and to seek for a more friendly system, the problem of software component organization in the library has to be examined. The classification of software components is a crucial issue. It conditions the activity of the whole retrieval system, and aims at putting similar components in the same category (they are about the same topic). The classical theory of categorization defines categories as containers of components with common features or properties. A component can be either inside or outside a certain category depending on having all the properties or not. This classical view does not deal with components having only a subset of required properties. To overcome this problem, prototype theory is to be used [16, 24]. It relies on the principle of the degree of representativeness, meaning that some components are considered better representatives of a category than others. Therefore the notion of belongingness of a component to a category is based on how much features it shares with the category in question. This view will permit to perform a polythetic classification of components retaining that a component has a "non-sharp" identity.

From the above discussion, it follows that a software retrieval system fundamentally requires inexact matching and fuzzy classification mechanisms. These requirements are supported by the fuzzy set theory of Zadeh [30]. Within fuzzy set theory, a natural language description of a software component is represented by a fuzzy set of features [6]. Each feature is an indexing term with an associated membership value, called weight. The membership value expresses the variable degree of significance the feature has in representing the content of the description. In the vector space model, it is directly computed from the number of occurrences of this feature in the description. During the retrieval process, a distance is computed in terms of the number and weights of common features between the user's query and the representation of each component in the library. These components that allow for small distance, generally expressed in terms of a threshold, are retrieved. Since the fuzzy set theory provides a selection of relational operators [8, 17], it is possible to use triangular norms (t-norms), triangular conorms (t-conorms) and similarity measures (equality, subsethood) to compute the distance between the query and the components' descriptions.

The literature shows that many people have used techniques derived from fuzzy sets to support the classification of documents. These techniques range from clustering algorithms to neural networks [17]. Using such mechanisms allows to reduce the search space for a fitting component, since we measure the distance only between the query representation and the prototype of each category. This way of reasoning is very plausible since the two criteria characterizing a retrieval system, namely fuzzy matching and fuzzy classification, are satisfied. Using fuzzy sets independently of any other technique has raised some critics [16, 24]. The critics show that logic connectives based on t-norm and t-conorm operators cannot conveniently account for the prototypicality of the elements when looking for creating complex categories.

4 Towards an Intelligent Retrieval System

4.1 Problem of Representation

The intermediate representation of software description (i.e. indexing) is the crucial part in a software retrieval system. Low recall (the ratio of retrieved relevant assets to the total number of relevant assets) and precision (the ratio of retrieved relevant assets to the total number of retrieved assets in software/information retrieval are due to the problems of properly representing the documents content. We mentioned that there are two techniques which are widely used, namely keywords and the vector space model. Both of them suffer different limitations.
Keywords, are difficult to extract and cannot express the whole semantic content of a document. The problem between string matching and concept matching is most clearly seen in the synonym-/homonym problem. However, this is just an extreme situation. In general, words express concepts only to a certain degree but might also be related to several concepts to a certain degree. To properly interpret words, we have to consider them in the broader (semantic) context of the text where they appear. Keyword matches do not take such relations into account.

On the other hand, the vector space model was developed to eliminate many of the problems associated with exact matching techniques. It considers documents as vectors of terms in a space where coordinates of each vector are weights assigned to each of the terms. Despite using stop list and stemming, techniques to reduce the number of terms, the size of a repository’s document vector is usually large and the vector contains some terms that are not important for the semantic content of the document. Further, if machine learning techniques are used then both, the performance and the training time, will be negatively affected by the vector size. The retrieval performance depends largely on the quality of the features used to represent the documents.

To improve the accuracy of the indexing operation, two solutions were suggested in the literature. They are, knowledge base integration and feature selection. By using some knowledge base (i.e. thesaurus), it is possible to take into consideration linguistic phenomena such as synonymy and hypernymy. This integration has substantially improved the retrieval effectiveness expressed in terms of recall and precision [9]. Also, some retrieval systems have used syntactic patterns such as compound words and nominal groups as indexing elements [4]. On the other side, many methods have been used to reduce the features embedded in the documents [28]. Using such algorithms, it is not easy to avoid eliminating features that are important. Consequently, the representation of the whole semantic content of the document will be seriously affected. Hence, it sounds interesting to perform such operation more intelligently and without extra cost of such procedures.

From the above discussion, it is clear that an efficient retrieval model should incorporate term relationships and focus on significant features in order to circumvent the limitations in the accuracy and generality of the indexing process.

To overcome the drawbacks outlined, we have suggested in [2] the use of systemic functional theory for software/information retrieval.

### 4.2 Systemic Functional Theory-based System

By taking advantage of the cognitive aspect inherent in the organization of texts, our aim is to find the location of important information. To satisfy this goal, we exploit the theme-rheme theory, the basis of the systemic functional theory [7].

From a structural point of view, a proposition is divided into a *topic (theme)* and a *comment (rheme)*. Halliday [13, p.36] defined theme as: “...a function in the clause as a message. It is what the message is concerned with, the point of departure for what the speaker is going to say.”

A theme identifies something known by both speaker and listener. It is the part of the sentence from which the speaker proceeds by presenting the information to be commented in the remainder of the sentence. The *rheme* is defined as the remainder of the message, where the theme is developed [13, p.38]. It introduces new information, as a comment, to further explain the topic. Danes [7] noted that the important information lies in the theme part. Writers or speakers place their concerns within the message as thematic content. Thus, the content of the rheme tends to represent the details of the theme.

Since the theme occurs in most languages at the beginning of a sentence, it can take forms such as a nominal, adverbial, or prepositional groups. Furthermore, one can distinguish marked from unmarked themes. If it is unmarked, it occurs in the form of the subject (nominal group) of the sentence, whereas the marked theme is in the form of an adjunct (adverbial group and prepositional phrase). (For more details see [13]).

To perform text analysis efficiently, we exploit both structure and texture; meaning that our attention is put on what and how something is said in the document. Therefore, we focus on passages where meaningful content is likely to appear. We discussed two approaches for representing the semantics of documents [2]. In the first approach, called meaningful sentences approach, we assume that the main topic of a text is clearly stated by a few summarizing sentences. These topical sentences are determined by computing the number of themes appearing in each sentence. If used as index, they certainly increase precision and recall. Since the index is extracted from the content of the document and includes basic elements of the meaning it carries the appropriate contents.

In the present paper, we focus on the second, the stratified abstractions approach. Here, words or phrases are considered as semantic units. They describe the concerns of texts are main features. They represent the set of items, appearing as themes that
are explained. Items representing details of the main ones and appear as rhemes are called secondary features. We depart from the idea that an important concept is usually commented. These comments introduce details related to the topic of a given concept and represent facets of that concept. Therefore, to compute the impact of each concept, we introduce the notion of weight. The weight of a thematic concept reflects the significance of that concept in carrying the content of the document. In other words, the weight shows the contribution of the concept in representing the document. In other words, the weight shows the central role of a concept in defining thematic concepts. It is the number of occurrences of the rhematic concept appearing in the rheme part of a sentence throughout the text.

**Definition 1:** Topicality power $P_T$ measures the centrality of a concept and provides the ability to determine the main objects of the document. It is computed by summing up the number of occurrences of rhemes explaining the given concept.

$$P_T = \sum_{ij} r_{ij}$$ (1)

Likewise, the rhematic concepts are identified and weighed. The weight in this case reflects the contribution of a rhematic concept in defining thematic concepts. It is computed using the explanatory measure defined as follows:

**Definition 2:** Explanatory power $P_E$ measures how well a rhematic concept serves to explain the thematic concepts. It is the number of occurrences of the rhematic concept appearing in the rheme part of a sentence throughout the text.

$$P_E = \sum_{i} r_{ij}$$ (2)

An illustrative example (Tab. 2) shows the analysis results of the document describing the UNIX command “MKDIR”(Tab. 1).

To bind words to concepts (i.e. to connect words to each other) a thesaurus allowing for lexical cohesion devices is necessary. Note that, there is a similarity between this method and the co-occurrence analysis when considering the sentence as the “look-up” window and the thematic themes as the concepts to which we want to measure the collocation degree with the rhematic concepts. Therefore, our method differs from the co-occurrence analysis. Our processing does not take care of the size of the sentences, whereas co-occurrence analysis is limited to the length of the window. Also, our approach allows unusual co-occurrence because we filter out the concepts not by their co-occurrence value but by their explanatory and topicality powers.

### Table 1. Excerpt of MKDIR description

<table>
<thead>
<tr>
<th>THEME</th>
<th>RHHEME</th>
<th>Creation</th>
<th>Directory</th>
<th>File</th>
<th>Owner</th>
<th>Group-id</th>
<th>Change</th>
<th>Topicality Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mkdir 1,3,6</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Entry 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Directory 2,4,5,7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>File 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Parent 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Group-id 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Change 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table 2. Thematic and rhematic concepts analysis. Themes are represented by the sentence number they appear in. Rhemes are represented as columns. An element $r_{ij}$ indicates the number of occurrences of the rhematic concept $R_i$ that “explains” the thematic concept $T_j$ along the text.

### 4.3 Uncertainty-centered System

Central to our retrieval system is the concept of uncertainty. This is drawn directly from the two measures we described earlier. The notion of “explained by” is actually a fuzzy association between rhemes and themes. First, we will show how fuzzy sets can serve to model the problem of retrieval. Then we present a neural network model capable of supporting the reasoning developed within fuzzy sets. Formally each thematic concept is a fuzzy set and can be defined as:

**Definition 3:** Fuzzy theme representation is a fuzzy set $(R, h)$ where $h: R \rightarrow [0,1]$ is associated to a thematic concept $(T_i)$ with $R$, the set of rhemes serving...
as base set (or support). The membership function $h$ is given as:

$$h = \frac{\text{No. of occurrences of the rheme } R_{ij} \text{ explaining } T_i}{\text{the topicality power of theme } T_i} = \frac{r_{ij}}{P_{T_i}}$$  \hspace{1cm} (3)$$

The function $h$ is a local weighting function. After the topicality value of a theme is determined, $h$ estimates the contribution (a value between 0 and 1) of each rheme (that contributes in specifying the context of the theme in question) in producing topicality power. One might note that this computation of $h$ has a certain normalizing property.

Finally, we want to represent documents as fuzzy representations of the themes they contain in relation to themes occurring in the archive in question. The membership value of some theme is either 0, if that theme is not present in a given document or it is a value between 0 and 1, normalized for the number of themes occurring in this particular document. This membership value smoothes differences in the length of documents by downgrading the number of occurrences of a theme in rather complex documents. Thematic complexity is not computed as the sheer length of a document, since the wordiness of explaining the themes is taken care of earlier. Thematic complexity is computed rather as the presence and degree of explanation of a certain theme in relationship to other themes present in the same document. This leads to the following two definitions:

**Definition 4:** Thematic complexity is the sum over the topicality power of all themes mentioned.

$$C_T = \sum_i P_{T_i} = \sum_i \sum_j r_{ij}$$ \hspace{1cm} (4)$$

**Definition 5:** Fuzzy document representation is a fuzzy set $(T, g)$ where $g: T \rightarrow [0,1]$ is associated to a document $D$ with $T$, the set of all themes mentioned in some document kept in the archive serving as base set (or support). The membership function $g$ is given as:

$$g = \frac{\text{the topicality of the thematic concept } T_i \text{ in } D}{\text{the thematic complexity of document } D} = \frac{P_{T_i}}{C_T}$$ \hspace{1cm} (5)$$

This definition tells that a document is represented not as a simple fuzzy set but as a “complex” fuzzy set. It is a fuzzy set where each element is a fuzzy set. This type of fuzzy sets is known as “type-2 fuzzy sets”. To keep the relationship to the hierarchical nature of description representation, we refer to it as “hierarchical fuzzy sets”. Figure 1 shows this notion of hierarchy more clearly.

![Figure 1. Weights gathering from individual documents](image)

### 4.3.1 Comparing Documents

For retrieval we need to assess the similarity between a natural language query and the natural language description of components kept in the archive. Similarity (or the degree of equality) between two fuzzy sets $A$ and $B$ is usually measured by:

$$S(A, B) \triangleq \frac{|A \cap B|}{|A \cup B|}$$ \hspace{1cm} (6)$$

where $||$, \( \cap \), and \( \cup \) are defined as:

$$\begin{align*}
|A| &= \sum_i \mu_A(x_i) \\
\mu_{A \cap B}(x_i) &= \min\{\mu_A(x_i), \mu_B(x_i)\} \\
\mu_{A \cup B}(x_i) &= \max\{\mu_A(x_i), \mu_B(x_i)\}
\end{align*}$$

In our work, similarity can be based on the information contributed by themes of two documents or by assessing the similarity of rhemes. Since the rhemes and themes are related by the relation “explain”, we propose to use a combination of both similarities. Thus, the similarity function will be the sum of two similarity degrees and is given as:

$$\text{Sim} = \lambda s_1 + (1 - \lambda)s_2$$ \hspace{1cm} (7)$$

where:

- $s_1$ is the similarity degree between the sets of the themes of the first document and those of the second document, and $s_2$ is the similarity degree between the sets of rhemes of the first document and
Neural networks have been used in the domain of software descriptions. Some authors have used simple neural networks \[1/0, 2/0\]. Others \[1/9, 2/9\] have exploited hierarchical retrieval. Some authors have used simple neural networks, precisely self-organizing maps \[6/\].

Meanings are defined entirely by other words. Thereby, it becomes clear that neural networks are able to fully use in order to identify relations among documents. Neural network approaches may be successfully used in order to identify relations among documents and to keep contextual usage of words (word meanings are defined entirely by other words). Therefore, it becomes clear that neural networks are able to perform some implicit inferences (see section 3), such as extending the meaning of the query. We rely on neural networks based on unsupervised learning for matching natural language software description.

In the sequel, we try to show how the notion of "hierarchical fuzzy sets" can be supported using a cascade of two neural networks: fuzzy associative memories (FAM) for mapping explaining concepts to explained concepts and fuzzy adaptive resonance theory (Fuzzy ART) to perform the classification of documents using explained concepts (see Fig. 2).

Dealing with semantics assumes dealing with some crucial questions such as how word meanings are related to concepts, and how word combinations are supported. Therefore, to simulate the human interpretation of words, the context of words has to be properly understood. This process allows naturally to perform a unique interpretation of a given word and thus resolves the ambiguity attached to multiple-meaning words. From this perspective neural networks are a very appropriate tool.

The choice of using FAMs to perform the first level of mapping in the whole process, namely the association of rhemes to themes (but also to realize "hierarchical fuzzy sets"), is motivated by the power of the FAMs in knowledge modeling. In fact, they allow to build a complex concept (theme) from fine-grained pieces of knowledge (rhemes). FAMs can simulate a knowledge base as a long term memory expressed in terms of weights (matrix). The propagation of activations along the network enables not only to determine the meaning of a given concept in given contextual conditions it also enables some implicit operations such as inference and logical connections (since weights are computed by fuzzy logic operators). More clearly, if a concept is activated by a pattern, then, whenever a similar pattern is introduced, the concept is activated. Using FAMs will allow to trigger a concept with a certain degree of evidence because they map fuzzy sets to fuzzy sets \[15, 17\]. Given a fuzzy set \( R \) and its corresponding fuzzy set \( T \), their representations are given as:

\[
R : (R_1/\mu_R(R_1), R_2/\mu_R(R_2), \ldots, R_m/\mu_R(R_m))
\]

\[
T : (T_1/\mu_T(T_1), T_2/\mu_T(T_2), \ldots, T_n/\mu_T(T_n))
\]

with \( \mu_R \) and \( \mu_T \) being the membership functions that map an element \( R_j \) of \( R \) and \( T_i \) of \( T \) and interpret the semantic significance of \( R_j \) for \( R \) and \( T_i \) for \( T \). In fact, for a given document, \( \mu_T(T_i) \) is the function \( g \) computed in (eq 5) and represents the averaged computed topicality power of the thematic concept \( T_i \) which is given by:

\[
\mu_T(T_i) = \frac{\text{topicality power of } T_i}{\text{sum of the topicality powers}} = \frac{P_{T_i}}{\sum_i P_{T_i}} = \frac{P_{T_i}}{C_T}
\]
\( \mu_R(R_j) \) represents the averaged computed explanatory power of the thematic concept \( R_j \) and is given by:

\[
\mu_R(R_j) = \frac{\text{explanatory power of } R_j}{\text{sum of explanatory powers}} = \frac{P_{E_j}}{\sum_j P_{E_j}}
\]

whereas the mapping from \( R \) to \( T \) plays the role of memorizing the association between both sets. Consequently, the long-term memory (the set of associations) are given by a correlation-minimum encoding matrix\(^2\):

\[
W = R^T \circ T
\]

The long-term memory can also be given by a correlation-product matrix, where the matrix composition operation is replaced by matrix multiplication.

\[
W = R^T T
\]

The equations (12) and (13) serve to perform the recall procedure of the themes \( T \) and the rhemes \( R \) with respect to the equations (10) and (11):

\[
T = R \circ W
\]

\[
R = T \circ W^T
\]

\[
T = RW
\]

\[
R = TW^T
\]

We conclude this idea by saying that using FAMs can efficiently deal with hidden patterns. This implies that, if a set of rhemes is present and these rhemes are supposed to explain a given concept but this latter is not explicitly present in the input, the net will activate it automatically. This is another level of handling vagueness and to tackle the problem of unclear or incomplete specifications provided by the users. It is also a way to handle situations where users abstract highly related facts into different abstract terms.

According to Fig. 2, the learning process is performed in two stages. Given a collection of documents, first the relation theme-rheme is learned using the fuzzy associative memories. Then the relation theme-category is performed, using the fuzzy adaptive resonance net. Therefore, during the recall process for a given document, the set of rhemes are presented to the FAM which outputs a set of thematic concepts. Those activated themes will serve as input to the second net (Fuzzy ART) which activates the corresponding category.

Fuzzy adaptive nets are based on unsupervised learning and are able to classify patterns in overlapping

\(^2\)The symbol ‘\( \circ \)’ stands for the fuzzy matrix composition (min-max rule) and \( R^T \) is the transposition of \( R \). categories by performing a many-to-many mapping [3].

As mentioned earlier, we are looking for mechanisms that perform a clustering based on the prototypicality principle. Fuzzy ART not only satisfies this requirement but is also able to perform a hierarchy by simply increasing (or decreasing) a parameter called vigilance. The network is thus capable of discrimination [1]. The processing cycle of Fuzzy ART mainly consists of three steps: category choice, vigilance test, and resonance. During the first step, the input \( X \) representing the set of themes \( T \) is presented to the net. Then a choice function \( H_j \) is computed for each category \( j \) defined by a set of adaptive weights \( W_j \).

\[
H_j = \frac{|X \cap W_j|}{\alpha + |W_j|}
\]

\( \cap \) denotes the fuzzy AND operator and \( \alpha (> 0) \) is a user-defined parameter called the choice parameter. The category \( J \) with the maximum value of \( H_j \) is chosen. During the second step (i.e., the vigilance test), the similarity between \( W_j \) and the input \( X \) is compared to a parameter \( \rho \) called vigilance.

\[
\frac{|X \cap W_j|}{|X|} \geq \rho
\]

If the test is passed, resonance occurs and learning takes place. If not, the category \( J \) is excluded and a new category whose \( H_k \) is the largest value is chosen. Then the vigilance test is repeated until the condition is satisfied. During the last step, the weights \( W_j \) of the selected category are updated according to the rule:

\[
W_j^{(new)} = \beta(X \cap W_j^{(old)}) + (1 - \beta)W_j^{(old)}
\]

5 Overview of the System

The system under construction (see fig. 3) consists mainly of two modules: the natural language processor and the neural networks. The natural language processor performs the analysis of the software description (resp. the query specification). Three steps are required to do that. First the software description is tagged by associating to each word a syntactical tag. This step aims at considering only semantically meaningful words (prepositions, articles, adjectives, auxiliary verbs, etc., are excluded). The second step, stemming, reduces the words to their roots (by eliminating the endings of the words). It aims to enhance matching and to reduce the size of feature space. To avoid common verbs or nouns, we also make use of a stop list which has to be fixed after a collection of documents have been analyzed. The third step computes
the topicality and the explanatory powers by considering the list of meaningful words that appear as themes or rhemes in each sentence of the given description according to eq. 1 and 2. To determine themes and rhemes, an algorithm based on details provided in [13] is used. Using a thesaurus, lexical relations (i.e. cohesive links, e.g. synonymy) between themes (resp. rhemes) are supported. The second module in this ar-

![Diagram](https://example.com/diagram)

**Figure 3. A general overview of the system**

chitecture consists of a cascade of neural nets shown in fig. 2. Having either a document to classify or a query to match, the nets perform a learning or a recognition process. The nets are built after fixing the number of themes and rhemes extracted from a given collection of documents. The learning process uses equations (8,9) to compute the inputs to the nets.

Preliminary results showed promising results, however a final judgment cannot be given at the moment. To provide empirical results, we need to perform experiments on real collection of software descriptions.

6 Conclusion

We presented a new technique for software/information retrieval that is able to intelligently perform document analysis. This technique is based on the notion of systemic functional grammar of natural languages. It takes into account both, structure and texture of the document and tackles the problem of reducing irrelevant information which was treated widely in the literature [28]. It has to be mentioned that a variety of techniques (e.g. the vector space model, probabilistic models, symbolic models, neural nets, genetic algorithms, etc.) have been used in the general framework of information retrieval. But all these techniques circumvented the issue of intelligently understanding documents.

On the other side, using soft computing for software engineering generally and for software retrieval particularly becomes increasingly an important issue. Especially when the description of software components is based on natural language, the uncertainty phenomenon turns out to be a vital aspect of the retrieval system. The main fuzzy techniques which have been applied in the domain of software retrieval are fuzzy sets and neural networks. In this paper, we tried to show how to apply both techniques in order to satisfy the main requirements of a software retrieval, namely shallow concept comprehension by fuzzy matching and fuzzy clustering. This technique exploits the problem of fuzziness and imprecision of natural language texts in a positive way. Coping with fuzziness is aimed at improving at least the degree of recall. Showing a further extension to improve precision is beyond the scope of this paper. It will be reported at another occasion.

References


Reasoning with Design Knowledge for Interactively Supporting Framework Reuse

Wolfgang De Meuter¹, Maja D’Hondt², Sofie Goderis¹, and Theo D’Hondt¹
¹Programming Technology Lab, ²System and Software Engineering Lab
Vrije Universiteit Brussel
(wdmeuter | mjdhondt | sgoderis | tjdhondt)@vub.ac.be

Abstract

The paper explains our efforts in using a rule-based reasoning engine to actively support reuse of object-oriented frameworks. The engine is embedded in the programming environment and is able to reason about the developed source code. By means of explicitly encoded design knowledge, it is able to actively guide the reuse process by intelligently assisting the programmer in taking the appropriate reuse steps. Reuse thus becomes an interaction between the programmer and the reasoning engine.

Two conducted experiments are explained in this paper, illustrating the applicability of such an embedded rule-based inference engine to explicitly guide both black box and white box reuse.

1. Introduction and Hypothesis

One of the major consequences of the popularity of object technology is the advent of object-oriented frameworks [11] [2] [9]. As opposed to a single software application, a framework is a reusable software system that offers a solution to a family of related problems. Repeatedly reusing a framework results in a number of similar software applications that only vary in well-defined places. These variations on a common theme are achieved when the framework is constructed as a well-designed skeleton of code that is instantiated by filling in the so-called hot spots.

Currently, a distinction is made between white box reuse and black box reuse of a framework [3]. In the case of the former the framework's hot spots are implemented as abstract classes or classes with default behaviour, and reuse of the framework requires techniques such as overriding and subclassing. Hence the reuser needs to know the details of the framework design and implementation in order to make controlled changes. Black box reuse involves configuring highly parameterised existing parts of the framework rather than adding new behaviour. This requires techniques such as polymorphism and parameterisation. In this case design and implementation details do not need to be exposed to the reuser, who nevertheless needs to be aware of the existing parts in order to select the correct ones and compose them.

The aforementioned reuse processes indicate that good framework reuse depends on the availability of documentation [4]. Whether white box reuse or black box reuse is applied to a framework, the documentation should clearly describe the intended behaviour and use of the framework, and provide a thorough explanation of its design and implementation. The documentation has to be informal enough to be easily understood by framework reusers, without becoming ambiguous.

However, current state of the art does not provide satisfactory techniques for framework reuse documentation. One technique is documenting frameworks with patterns [4] [1], which state the purpose of the framework, describe how to reuse it, and explain its design and implementation. Another way framework reuse is supported, is by means of a cookbook [6] [5] which provides standard recipes offering a step by step explanation of how the framework can be reused. In both techniques, several problems arise:

- The documentation is written down in a variant of natural language, even if some kind structuring is applied. This results in informal and thus ambiguous documentation.
- It is not always clear what pattern or recipe to use, or the reuser does not even realise there exists a suitable one. A mechanism, similar to a wizard, is missing to interactively guide the reuser through the correct reuse process.

We observe that documentation to guide framework reuse is knowledge. In artificial intelligence numerous successful knowledge representation schemes have been developed, accompanied by powerful inference engines that are able to reason with this knowledge [8]. Therefore, our hypothesis is that an expert system whose knowledge base consists of the design and reuse
knowledge of a specific framework, can interactively guide the reuser and enhance the quality of the reuse process. During the review process of this paper, we came across similar work that was conducted independently from ours [14], which shows that this research track is indeed relevant.

The next section explores the requirements of an interactive guiding system for framework reuse. Section 3 describes the technological choices we made corresponding to these requirements in order to develop a concrete system. Section 5 sketches two concrete experiments we conducted with this system regarding black box and white box reuse in order to prove our hypothesis. Section 6 concludes.

2. Requirements for Supporting Interactive Framework Reuse

A first requirement is that framework development should still be performed in a standard object-oriented programming language, since existing techniques and methods ought to remain applicable. The expert system with design and reuse knowledge should be integrated in the development environment of this language. We elaborate on the language of choice, Squeak, for proving our hypothesis in Sec. 3.1.

Knowledge about framework reuse involves information on how a specific framework should be reused and on framework reuse in general. Because we want this knowledge to be easily maintainable upon evolution of the framework, the knowledge should be explicitly represented and separated from the reasoning algorithm that uses it to interactively guide the reuser. This reasoning algorithm should be a forward chainer because inferring the steps in the process of framework reuse, is a form of goal-less reasoning. Indeed, no goal is known beforehand that the reasoning algorithm needs to prove, as is the case with a backward chainer. The forward chainer will instead try to generate possible solutions starting from initial facts. Another crucial property of forward chainers is that access to their state is indeed maintained. Therefore, the reuser can interactively guide the system to generate possible reuse recipes. When certain steps have been performed by the reuser, the system is able to take control for further instructions, remember what still needs to be done, and adapt its advice to the concrete situation the reuser is in. This illustrates another requirement for our system: it has to be coupled to the framework implementation, in other words as a meta-system that reasons about the code level. How we incorporated this coupling mechanism in our system is explained in Sec. 4.

3. Experimental Setup

The following describes the ingredients of a system for interactively supporting framework reuse that are necessary to fulfill the requirements listed in the previous section. The choices Squeak, KAN and the coupling mechanism discussed above are explained and motivated.

3.1. Squeak

The reasons for choosing Squeak [7], an open-source Smalltalk environment developed at Disney Imagineering, are:
• We decided to use Smalltalk because we consider it to be the pearl of object-oriented programming. This is especially important in framework development as it requires extreme flexibility in order to achieve the highest possible reusability. The Squeak Smalltalk implementation is particularly interesting because it has a very active user community.
• The expert system needs to be plugged into the programming language in such a way that it can take control after certain actions of the reuser. Currently, only Smalltalk with its successful tradition of adaptable programming tools such as browsers and finders meets these high standards of openness. Although development environments for other languages (notably Java) are catching up swiftly, we still feel that there is currently no widely used environment that can stand the comparison with Smalltalk environments.

3.2. The KAN Forward Chainer

Since artificial intelligence has a multitude of knowledge technologies at hand, we made it our explicit goal not to try to find something new, but instead to select an existing solution. We opted for KAN, a "lean and mean' expert system shell, and implemented it on top of Squeak. We will not give a full account of KAN, but refer the interested reader to [10] for the specification of KAN as a language and to [13] for an elaboration of the version we implemented.

Figure 1 contains some code excerpts supporting our discourse. The code comes from a small expert system implementation.

---

1 We do not suppose that reuse in itself is goal-less, only that it is difficult to express the goal of reuse in predicate calculus. Moreover, the reasoning process needed here should not attempt to prove one postulated goal (as does a backward chainer), but perform inference based on input from the reuser instead.

2 As a matter of fact, KAN was implemented in Squeak, but the implementation details are outside the scope of this paper.
whose task is to classify finches based on their external properties.

\[
\textsf{(define (ruleset finch-type) finch-rules)}
\]

\[
\textsf{(define (rule finch-rules) diamond-firetail-rule (if (beak red) ...) (then (conclude (species diamond-firetail)) (communicate "species is diamond firetail") (investigate identity-determination-rules)))}
\]

\[
\textsf{(define problemsolver finch-classification-solver (object (a finch-type)) (ruleset finch-rules) (goal species))}
\]

**Figure 1: Some KAN code**

KAN is an expert system shell whose entities (fact base, rules, ... are internally organised as frames of slots. For example, a rule is a frame containing an `if` slot and a `then` slot. Every slot needs a filler for that slot. The filler for an `if` slot is a conditional expression. For the `then` slot of a rule, the filler is a sequence of actions. The general syntax for a slot \(s\) with a filler \(f\) is \((s\ f)\).

The main KAN frame is called a problem solver. A problem solver consists of a ruleset slot, an object slot and a goal slot. As an example, Figure 1 shows the declaration of a problem solver `finch-classification-solver`. A goal is a monitor on some fact that becomes satisfied when the fact is indeed concluded to be true. The rule set the problem solver will use for reasoning is the `finch-rules` set. The idea of rule sets is that the forward chainer cycles through the rules in the set until the goal is reached or until no more new facts are derived. Figure 1 shows one exemplary rule `diamond-firetail-rule` of the `finch-rules` rule set. The rule concludes that a finch is a diamond firetail finch (and informs the user about this) whenever the finch has a red beak and a few other properties.

One of the actions the conclusion of a rule can contain is investigate, permitting rule sets to be hierarchically organised. In [10] this hierarchical organisation of rule sets is aligned with the approach of organising expert systems around ‘tasks’ to be solved. A rule set corresponds to such a task and an investigate action corresponds to the spawning of a sub task. Rules can also contain communicate actions to display a message to the user, and conclude actions that conclude a new fact in the fact base of the problem solver. In Figure 1 we can see that when the `diamond-firetail-rule` succeeds, it will conclude the species of the observed finch, communicate this finding to the user and call the `identity-determination-rules` rule set to investigate whether the observed finch was already encountered before and if so, which specimen is being observed.

4. Coupling KAN to Smalltalk

We extended the original KAN design with facilities to access the underlying framework implementation written in Smalltalk from within a running expert system. Two constructions where added to the original KAN language:

- The first construction is the `(smalltalk <name>)` primitive. This expression looks up a name in the Smalltalk runtime and injects the corresponding Smalltalk object into KAN. This is achieved by simply wrapping the Smalltalk object as a KAN value at the level of the KAN evaluator.

- The second construction is a primitive `(send o selector [o1 ... ok])`, which can appear in conditionals expressions as well as in the conclusion actions of rules. It requires that all sub expressions \(o, o_1, ..., o_k\) evaluate to Smalltalk objects as described above. When this is the case, the message `selector` is effectively sent to the Smalltalk counterpart of \(o\) with \(o_1\) to \(ok\) as arguments. This is accomplished by unwrapping the receiver and the arguments in order to get actual Smalltalk objects, which in turn are used as parameters in the `perform:withArguments:` primitive from the Smalltalk meta object protocol. Finally, the resulting Smalltalk object is wrapped again in order to inject it back into KAN.

These two constructions grant full control over Smalltalk from within KAN, as in Smalltalk everything is done using messages (even making classes e.g.).

5. Experiments

We employed the technology described in the previous section to support our hypothesis that rule-based expert system technology indeed enables active framework documentation. We attempted to confirm the hypothesis by designing two tiny expert systems that guide a programmer in the reuse process.

5.1. Experiment 1: Black Box Reuse

![What is the value of store for theCollection? operations characters integers floatingPoints signedIntegers unsignedIntegers](image)

**Figure 2: A pop-up menu for multi-valued questions**

The first experiment involves the applicability of our approach to guide a black box reuse process. To confirm

---

3. In KAN terminology, an object is just a fact base. This has nothing to do with the term ‘object’ in the object-oriented sense.

4. More precisely, the name is looked up in the global dictionary Smalltalk which contains all the global variables.
5.2. Experiment 2: White Box Reuse

In our second experiment the expert system shell was used to actively assist white box reuse of a framework. Before moving on to the experiment itself, we first elaborate on the framework on which the reuse experiment is based. We chose a LAN framework [12] we developed to serve as didactic artefact in the courses on reuse we teach at our university. This framework simulates a simple circular LAN network. The main elements of the framework are nodes (with subclasses for workstations and printers) and packets. Nodes work together to make packets go around the LAN. Packets have the responsibility to ensure that they are addressed to those nodes. There are different kinds of reuse we could think of. The existing family can be extended with new kinds of nodes, output-servers, packets, packet delivery systems (e.g. broadcasting), addressing schemes, and so on. However, in order to make these extensions correctly, the reuser has to follow explicit reuse recipes that reflect the design of the framework. These reuse recipes are encoded in the active cookbook we conceived. The idea of the active cookbook is that there is a problem solver telling the user gradually which classes or methods to create, after which it is suspended to allow the user to effectively perform these tasks. At this point the user has full control. He can program whatever he wants, but as soon as these tasks have been carried out, the problem solver restarts. As such, the problem solver is continuously being suspended (by itself) and restarted (by the programming environment in which the user is operating).

In order to comfortably express this interactive behaviour, we had to enrich the theoretical coupling of Sec. 4. First, we extended the KAN formalism with a new kind of action (suspend) to be used in rules. This suspends the problem solver, thereby keeping its internal state for later resumption. Second, we extended the Smalltalk programming environment with a so-called task pool containing ‘tasks the programmer still has to do’. From the side of the problem solver, the send construction can be used to post tasks in the task pool. Currently, we use two kinds of tasks, one to express that a certain method has to be added to a class, and the other one to express that a subclass has to be made of a certain class. Furthermore, both tasks have a flag associated to them which expresses whether or not the problem solver has to be resumed after the task has been executed. From the side of the programming environment, each time the programmer writes a new method or class, the task pool is consulted. If it appears to contain a task that matches the method or class just added, the task is removed from the pool and if its flag requires so, the problem solver is resumed. Based on the knowledge that the task has been fulfilled, it can continue the reuse process it was guiding by posting new tasks in the pool. Notice that it is thanks to Smalltalk’s openness that we were able to change the programming environment such that it calls our problem solver when needed.

The problem solver we implemented is started by sending reuse to any class of the framework. It then works in two phases. In the first phase, a rule set interrogates the user and determines which reuse recipe the reuser has to use. Of course, if there is no recipe that fits the need of the programmer, the problem solver stops. When the right recipe is determined however, a rule set that goes through the different reuse steps (adding methods and classes) is executed. This rule set posts the reuse tasks yet to be performed in the task pool and subsequently suspends itself.

\[
\begin{align*}
\text{(define \ (rule \ printerrules) \ addNewPrinter)} \\
\text{\ (if \ (not \ printerclassdone))} \\
\text{\ (then}} \end{align*}
\]

This extension is not fundamental to our approach. The construction explained in Sec. 4 allows us to send any message to any Smalltalk object from within KAN. So, in theory, we can do everything in KAN. However, KAN as a paradigm is unsuitable to implement things which are extremely sequential in nature.

\[\text{For the technical details we refer to [13]. The essence of the adaptations consists of changing the browser at the point of class and method accepts.}\]
(send (smalltalk TaskPool) classNeeded:solve: [[smalltalk Printserver] true]) (conclude printerClassDone) (suspend))

(define (rule printerrules) printerAdded
  (if (printerClassDone)
      (not printerMethodsDone))
  (then
    (send (smalltalk TaskPool) methodNeeded:forClass:solve: [#isDestinationFor:]
      (send (smalltalk TaskPool) lastClassAdded [] true)
    )
    (conclude printerMethodsDone)
    (suspend)))

(define (rule printerrules) newDocument
  (if (printerMethodsDone)
      (not documentClassDone))
  (then
    (send (smalltalk TaskPool) classNeeded:solve: [[smalltalk Document] true])
    (conclude documentClassDone)
    (suspend)))

Figure 3: A part of the ‘add new printer’ reuse recipe

An excerpt of the problem solver guiding reuse of the LAN framework is shown in Figure 3. It concerns three rules from the recipe that describes how to add a new kind of printer. To do this the user needs to create a new subclass of Printserver with a method isDestinationFor: that will be called by the framework. Furthermore, a new subclass of AbstractDocument is needed. This is clearly stated in the rules of Figure 3. The first rule simply says that the task pool needs to know that a new subclass of PrintServer is needed and that the problem solver has to resume when this is done. It subsequently suspends the solver. After the solver has been restarted, the second rule dictates that when this class was indeed added, the task pool has to be informed a new method has to be added to that class. A reference to the newly added class is asked from the task pool using lastClassAdded. The final rule tells the reuser to create a new subclass of AbstractDocument.

Notice that, because the problem solver’s state is preserved, properties like printerClassDone are still known. Thus, the first rule will no longer fire but the second rule will, since one of the preconditions of this rule is printerClassDone.

Future work in this research area consists of making the reuse documentation active so that it automatically evolves together with the framework. Currently, the reuse documentation - represented in the knowledge base of the expert system shell - is able to dictate where and how the framework should be adapted for reuse, but unable to adapt itself accordingly.

6. Conclusion

Although software reusability is spectacularly enhanced as a result of the use of object-oriented framework technology, good documentation is still indispensable to help it attaining its full potential. The fundamental problem is that documentation has to be unambiguous and interactive, the latter because it is often not clear what framework features to reuse, or that there exists a suitable feature in the first place.

We argued that expert system technology is a good stepping stone to alleviate this problem. To prove this hypothesis, we built an expert system shell on top of a Smalltalk programming environment, designed in such a way that it allows the construction of interactive support for framework reuse to guide the reuser through the reuse process. We conducted two experiments that are representative for two typical reuse processes, black box and white box reuse.

Although impossible to prove in a mathematical sense, we feel that our experiments convincingly show how both black box and white box reuse of frameworks are supported and enhanced by this blend of state-of-the-art object technology, meta-programming and artificial intelligence.

7. References

[3] Ralph E. Johnson and Brian Foote: Designing Reusable Classes. In Object-Oriented Programming
Minimally Adequate Synthesizer Tolerates Inaccurate Information during Behavioral Modeling

Johannes Koskinen  
Tampere University of Technology  
Software Systems Laboratory  
P.O.Box 553  
FIN-33101 Tampere  
Finland  
jomppa@cs.tut.fi

Erkki Mäkinen  
University of Tampere  
Dept. of Computer and Information Sciences  
P.O.Box 607  
FIN-3314 University of Tampere, Finland  
em@cs.uta.fi

Tarja Systä  
Tampere University of Technology  
Software Systems Laboratory  
P.O.Box 553  
FIN-33101 Tampere  
Finland  
tysta@cs.tut.fi

Abstract

The human-centered, interactive software engineering tools have to manage vague and inaccurate information. The user should be able to give incomplete input and change her mind when desired. In this paper we sketch tool support to master such situations in behavioral modeling and dynamic reverse engineering.

Our starting point is Minimally Adequate Synthesizer (MAS), an interactive algorithm that synthesizes UML statechart diagrams from sequence diagrams. It follows Angluin's framework of minimally adequate teacher to infer the target statechart diagram by consulting the user. In this paper we generalize MAS to allow inaccurate answers to the queries it poses to the user.

We consider different kinds of inaccurate answers the user can provide. We allow Probably yes and Probably no answers, i.e. weak Yes and No answers. The information obtained from these answers is considered less significant than that obtained from normal, definite answers. Hence, the user is given a possibility to change this information later. Furthermore, the user can postpone answering by saying Later. Finally, we discuss the possibility to add I don't know answers to the system. These can be handled by simulating all computations obtained by replacing them with Yes and No answers.

1. Introduction

UML [13,15] has become an industrial standard for specifying object-oriented software systems and for presenting and documenting various design decisions during the software development. UML provides different diagram types that can be used to view a system from different perspectives and/or at different levels of abstraction. The various UML models of the same system share information and are thus strongly dependent. Hence, meaningful transformation operations between several pairs of diagram types in UML can be defined. When supported by appropriate tools, such operations can automate a substantial part of both forward and reverse engineering.

In UML, both interactions diagram types, namely sequence diagrams and collaboration diagrams, are used to visualize object interactions. Although these two diagram types emphasize different aspects of object interactions, they are very close to each other. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways [15]. Automated transformation between them is supported, for example, in Rational Rose [14]. Usually, only a certain part of the diagram can be transformed. In some cases, the shared information is used just for checking consistency between the models. For instance, Rose maintains method lists across class diagrams and sequence diagrams to keep them consistent during editions.

Several tools for synthesizing state machines from variations of Message Sequence Charts (MSCs) [5] are currently available [8,9,10,16,17,20]. The UML notation (sequence and statechart diagrams) is used in methods introduced by Systä [18] and Whittle and Schumann [20]. These tools support behavioral modeling of software systems by helping the designer to quickly shift from constructing example scenarios to tuning the final specification of the behavior as a state machine. In [18], the synthesis algorithm is used for dynamic reverse engineering purposes: it can be used to examine the overall run-time behavior of an object as a state machine.

In UML based behavioral modeling, sequence and collaboration diagrams are used for showing examples of interaction between several objects, statechart diagrams are used for specifying the full dynamic behavior of a single class of objects. Sequence diagrams constructed during the system analysis are gradually refined in the design phase. When a sufficiently complete set of sequence diagrams exists, the diagrams are transformed
(usually manually) to a final specification of the system. Sequence diagrams, especially those constructed at an early phase of the software construction process, are often incomplete and inaccurate. Thus, a fully automated process for transforming them to a statechart diagram may generate faults or inaccuracies in the resulting diagram. Furthermore, a synthesized statechart diagram typically generalizes the information given as sequence diagrams: in addition to the example scenarios given as an input, it accepts various other paths. Thus, totally automatic synthesis algorithms may result in a state machine that contains undesired generalizations. To overcome these problems, we have introduced [11,12] an interactive synthesis algorithm Minimally Adequate Synthesizer (MAS).

In interactive systems, the user’s behavior can be unpredictable and inconsistent and she can provide inaccurate information to the system. Especially reverse engineering would benefit the possibility to deal with vague information, since “incomplete”, “imprecise”, “ambiguous”, and “inconsistent” are typical adjectives to describe its intermediate stages [7]. Interactivity of MAS requires that user’s mind changes have to be taken into account. Moreover, the user should have a possibility to give inaccurate answers and the algorithm should be able to handle them. In this paper we consider different kinds of inaccurate answers the user can provide to membership queries (i.e., whether a certain generalization is allowed or not). We allow Probably yes and Probably no answers (i.e., weak Yes and No answers). The information obtained from these answers is considered less significant than that obtained from normal, definite answers. Furthermore, the user can postpone answering by saying Later. Finally, we discuss the possibility to add I don’t know answers to the system. A formal treatment of user’s inaccurate answers would require the use of a version of modal logic. However, we restrict ourselves to an elementary and very informal manipulation of the modalities Probably yes and Probably no.

The factor of inaccuracy is present also when using state machine synthesis algorithms for dynamic reverse engineering purposes. The run-time behavior of the target system is often visualized as variations of MSCs (e.g., sequence diagrams). The information included in these MSCs depends on the level of granularity the information is extracted, the selection of parts of the system examined, and the actual usage of the software. Thus, MSCs rarely contain the complete event trace with all possible inputs. An interactive synthesis algorithm could provide considerable help for the reverse engineer to understand the behavior of the system. The user consultancy draws her attention to the key points in which the generated information is not sufficient. Moreover, it helps her to construct a higher level model of the behavior (a state machine) that corresponds to her understanding of the overall behavior of the system.

The rest of the paper is organized as follows. Chapter 2 gives an overview of MAS. The next chapter further deals with MAS by concentrating on the interaction between the user and the algorithm. In Chapter 4 we survey previous work on inaccurate answers in inference algorithms following the learning paradigm of minimally adequate teacher. Moreover, we briefly discuss approximate reasoning methods needed in the connection with inaccurate answers. Chapter 5 sketches different ways to handle inaccurate answers in MAS. The proposed method is then illustrated in the next chapter by considering an example. Finally in Chapter 7, we discuss the use of our extended algorithm and sketch an additional method (I don’t know answers) to express the user’s inability to give an exact answer.

2. An overview of MAS

MAS infers a UML statechart diagram from the sequence diagrams and other information provided by the designer following Angluin’s framework of minimally adequate teacher [1]. In what follows, we shortly describe MAS (for further details, the reader is referred to [11,12]).

The synthesis algorithms generalize information given in MSCs, i.e., the resulting statechart machine accepts more paths through the modeled system than represented as MSCs. The generalization is usually the desired effect. However, in some cases a synthesized statechart machine might also accept unwanted or erroneous paths and thus be “overgeneralized”. Applying the synthesis algorithm to an incomplete set of sequence diagrams may result in an overgeneralized state machine that does not meet the user’s intentions. MAS avoids undesired results, due to overgeneralization. To achieve this goal, the algorithm asks the designer for guidance during the synthesis process, when needed.

The input of MAS is constructed from a UML sequence diagram. A sequence diagram consists of participating objects and messages occurring between these objects. Objects are shown as vertical lines called lifelines and messages as horizontal arrows extending from a sender object to a receiver object. Given a sequence diagram describing a scenario, the trace originating from the diagram with respect to an instance I of class C is obtained as follows. Consider the lifeline corresponding to I. Starting from the top, for successive sent message e_i and received message e_i, add item (e_i, e_j) into the trace. If either of e_i or e_j is missing, then add NULL instead. If the explicit deletion of the object is not shown at the end of the sequence diagram, let the right hand side of the last pair be VOID.

A trace item (e_i, e_j) implies that at a certain point during the execution of the system, C sends a message e_i to some other object and then reacts to message e_j sent by
another object. Thus, we map each sent message with an action performed in a state and each received message with an event that causes a state change (i.e., fires a transition). This mapping is shown, e.g., in Figures 4 and 5.

The input of MAS is a set of traces, and MAS tries to construct a statechart diagram consistent with the input traces. However, this is usually not possible without a help from the user. MAS can pose membership queries and equivalence queries. In a membership query, MAS asks the user whether a given sequence of operations is acceptable. Based on its application specific knowledge, MAS can conclude the correct answer to most of the membership queries. This is essential for the usability and efficiency of the use. (In its original form, Angluin’s minimally adequate teacher makes far too many queries to be practically usable.)

When the data structures of MAS fulfill a certain condition (are “closed” and “consistent” in terms of [1]), it makes an equivalence query (a conjecture). This means that MAS outputs a statechart diagram, and the user can accept or reject it. If the user accepts the conjecture, the algorithm halts. In the case of rejection, the user is expected to give a counterexample to guide the further execution of the algorithm. MAS allows the user to change her mind during the synthesis process. A mind change may cause a major re-organization of the data structures. This is discussed in [12]. Figure 1 shows an overview of MAS.

![Figure 1. An overview of MAS](image)

We have a practical implementation of MAS, integrated to a real-world UML modeling tool, the Nokia TED [21]. TED is a multi-user software development environment that has been implemented at Nokia Research Center and is currently used at Nokia. It supports most of the UML diagram types.

### 3. User interactions in MAS

Algorithms that require user’s consultancy should pose questions that are meaningful and interesting from the user’s perspective. In the ideal case, the questions draw user’s attention to critical points in the underlying problem to be solved. Since answering many simple and/or similar questions is often frustrating, the questions posed to the user should guide, instead of bother her. The quality of a question depends on both the content and the presentation of the question: the question should be presented in a form that is easy to understand. Furthermore, a proper user interface is essential in such interactive systems.

Interaction with the user is the main advantage of MAS over previously known synthesis algorithms. Totally automatic synthesis algorithms, e.g., the algorithm used in SCED [8], may result in a state machine that contains undesired generalizations. Because MAS consults the user during the synthesis process, the user can be confident that such generalizations do not appear in the resulting statechart diagram.

MAS gives a membership query in a form of a (sub)path. The user needs to decide whether the resulting statechart diagram should accept the (sub)path or not. Currently, MAS poses the membership queries in a form of a simple sequence diagram with two participants: the object of interest and a participant that represents any other participant (inside or outside the system border) the object send messages to or receives messages from. Another approach would be to visualize them as a simple state machine. Note that it is not possible to highlight the corresponding path in the synthesized statechart diagram, since no statechart diagram exists at that point.

The task of providing counterexamples is the most difficult part of using MAS. Hence, the user interface should support the user to find proper counterexamples and to check their consistency with the other information available. A counterexample can be positive or negative. In a case of a positive counterexample, MAS should change the conjecture so that it also accepts the given counterexample. A negative counterexample, in turn, describes a generalization in the conjecture that should be prohibited.

MAS allows both positive and negative counterexamples to be given in a form of a sequence diagram. However, it is more natural to give positive counterexamples as sequence diagrams and negative ones by editing the statechart diagram (which is also possible in MAS). For example, deleting a transition from the statechart diagram is equivalent with giving a set of
negative counterexamples, which are no longer accepted by the statechart diagram when the transition is missing.

An easy method to define a very general type of negative counterexamples is to allow the user to select paths from the conjectured statechart diagram by clicking its states on the screen. Suppose the user clicks a pair of states $s1$ and $s2$ one after another. This can be interpreted so that all paths from state $s1$ to state $s2$ are forbidden. Allowing the user to give such a family of negative counterexamples decreases membership queries in the future.

4. Incomplete user inputs

Angluin’s original framework [1] of minimally adequate teacher assumes that all queries are answered correctly. However, in various practical situations the information provided by the teacher is inaccurate. Behavioral modeling is a typical example of such situations. Jahnke and Walestein [7] even advocate the view that the key factor in reverse engineering is the inaccuracy of the information available. In this chapter we first recall some of the essential concepts from the literature on minimally adequate teachers allowing inaccurate answers. After that we very briefly discuss the approximate reasoning methods needed when handling inaccurate information.

It is often supposed that the answers to equivalence queries are always correct although the teacher might otherwise be unreliable. This assumption is in accordance with our approach, since we consider all information related to equivalence queries and user's edit operations to be primary and correct. When considering the status of an individual path (i.e., when answering a membership query), the user may have difficulties to give an exact answer because the path is chosen by the algorithm. On the other hand, when answering an equivalence query and possibly searching for a counterexample, the user has the freedom to choose any path (string) she herself considers relevant.

Most of the previous work on inaccurate answers is related to the PAC learning model [19]. We work with exact learning algorithms since it is unnatural to consider user’s answers and their possible inaccuracy as a random process. On the contrary, we hope that the user indeed learns something when running MAS.

Inaccurate answers to membership queries can be persistent or non-persistent [2]. If an inaccurate answer is persistent, it does not change during the inference (synthesis) process. A special form of persistent answers is considered by Frazier et al. [3]. Their consistently ignorant teacher is not able to give answers from which the exact value of any inaccurate answer could be concluded. It is obvious that in our application the answers should be non-persistent, i.e., the user can change her mind also in the case of inaccurate answers. In the matter of fact, this is precisely what we are aiming at: the user should get a better and better understanding of the system so that her answers became more and more exact. Hence, we hope that the teacher (i.e., the user) is not consistently ignorant.

Using the terminology of inductive inference, we can say that our system follows the framework of exact learning and that our inaccurate answers are non-persistent. As far as we know, such systems are not presented earlier.

Angluin et al. [2] make difference between limited and malicious membership queries. A limited membership query of [2] accepts persistent $I$ don’t know answers, while a malicious membership query may give the wrong answer. In the limited membership query model a conjecture is “nonstrictly” correct if it agrees the target except possibly on samples for which an $I$ don’t know answer was given.

In the case of limited membership queries, the answers that are not omissions ($I$ don’t know) are guaranteed to be correct. In a malicious membership query, the answer may be an error. In our model, malicious answers are not considered, but all Yes and No answers are treated as correct. However, we allow the user to change her mind: a No answer may turn to be Yes and vice versa. This implies backtracking operations in the data structures of MAS as discussed in [12].

In all “theoretical” models for inductive inference allowing inaccurate answers (see [2,3] and the references given there), a formal criterion for success is essential in order to able to prove, for example, complexity results concerning the model. Our criterion for success depends on the user: whether or not she is satisfied with the resulting state machine.

We end this chapter by briefly discussing the methods of approximate reasoning needed in MAS. Hajek [4] associates the term uncertainty with degree of belief while vagueness is associated with degree of truth. When a user of MAS gives an inaccurate answer to a membership query, she expresses her degree of belief concerning the status of the path (string) in question in the conjecture to be given. Indeed, we do not think inaccurate answers as fuzzy. We are aiming at a deterministic and crisp model, but in the case of inaccurate answers, the user is, for some reason, unable to give a crisp answer. In other words, by giving an inaccurate answer the user is not saying that the path belongs to the model with weight (say) 0.8. Instead, she means that, at least at the moment, she is not sure whether or not the path should belong to the model.

Hajek [4] argues that systems dealing with uncertainty (i.e., the degree of belief) are related to various generalizations of modal logic. Hence, such systems could form the underlying theory of handling inaccurate information in MAS. However, we shall not give any formal treatment
of our modalities (notably *Probably yes* and *Probably no*). Our goals here are purely practical: we are aiming at a tool for supporting software engineers, and for that purpose, it is enough if the simple rules for implications to be given in the next chapter are useful in OO design.

Jahneke and Walenstein [7] carefully analyze the nature of inaccurate information in reverse engineering. They consider inaccurate information as fuzzy, following the decisions made in the Varlet environment [6].

5. Managing inaccurate answers in MAS

As described in [11,12], MAS allows only *Yes* and *No* answers to membership queries. User’s mind changes cannot be considered as incomplete or inaccurate answers, since at any moment MAS handles all information as “certainly true”. A better approach is to allow the user to give genuinely inaccurate answers. This, however, makes the structure of MAS considerably more complicated. The purpose of this chapter is to sketch ways to equip MAS with a possibility to handle incomplete and inaccurate answers. We consider here only membership queries, since we suppose that user’s answers to equivalence queries are always correct and definite.

When constructing a conjecture, MAS requires exact information indicating whether or not certain paths are possible in the system considered (or in the grammatical inference terminology: “whether or not certain strings belong to the target language”). Hence, in order to allow the algorithm to accept inaccurate answers, we have to have some kind of mapping from inaccurate answers to normal *Yes* and *No* answers. This is pictured in Figure 2 where an arrow stands for a membership query phase and a circle stands for a conjecture.

Figure 2 shows that we have to maintain the mapping from the inaccurate level to the exact level during the whole membership query phase; otherwise MAS would be unable to know when it is time to make the conjecture. However, we can allow some answers to be changed later, also after a conjecture. This can be enabled by maintaining information about those answers and their effects. Our task here is to define reasonable modalities in the inaccurate level and a meaningful mapping from the inaccurate level to the exact level, and to maintain it during the whole synthesis process. MAS will then do the rest of the job.

A simple way to support the user in cases of uncertainty is to allow her to answer *Later* to a membership query. In some cases, a question that is difficult to answer might become obvious, when the synthesis process gets further and the user learns more about the overall behavior. Furthermore, a difficult membership query might even get automatically solved later during the synthesis process. This happens if another query contains information from which the answer can be concluded. Implementing *Later* answers is straightforward. The algorithm just needs to keep track of postponed questions and to pose them later (if needed) when other membership queries have been asked. Eventually, the user has to give answers to all the questions that cannot be solved by MAS, since MAS is not able to give conjectures otherwise.

*Later* answers only postpone user’s decisions. She needs also stronger methods for expressing her ignorance concerning a specific query. It is natural to expect that although the user is not sure how to answer a membership query, she usually has some kind of assumption of the correct answer. A suitable approach then is to allow *Probably yes* and *Probably no* answers. In what follows, we refer the information obtained from *Probably yes* and *Probably no* as indefinite. In the data structures of MAS an indefinite answer is mapped to the corresponding crisp answer, e.g. *Probably yes* is mapped to *Yes*.

MAS tries to conclude answers to the membership queries without consulting the user. Its primary goal is to use definite information, i.e., information obtained as input traces, normal *Yes* and *No* answers, counterexamples, or information which is concluded from these or from the results of user’s edit operation on a conjecture. If

![Figure 2. A mapping from the inaccurate level to the level of MAS](image-url)
this is not possible, MAS tries to use indefinite information. Information concluded from indefinite information is also indefinite.

A Probably yes (resp. Probably no) is interpreted as a “Weak yes” (resp. “Weak no”). This means that the user is allowed to change a Probably yes (resp. Probably no) answer to be normal Yes or No later on. Since MAS uses also inaccurate information for automatically answering membership queries, the user cannot give contradicting information in form of answers to membership queries. Namely, a membership query that would allow contradicting answer is not posed to the user; MAS has already concluded the answer by itself. Furthermore, we consider automatic changes to be suspect. For instance, consider the behavior of an alarm clock and the following two subpaths: “setting the alarm on” (A) and “the alarm clock starts buzzing” (B). Suppose then that the designer has answered Probably yes to the membership query AB, which is a concatenation of membership queries A and B. Furthermore, she has answered No to a membership query B, since the alarm clock should not start buzzing if the alarm is not set on. The designer can then answer No to ABB (for the above reason), yet being unconfident whether the subquery AB should be allowed or not. Since (sub)paths depend on other (sub)paths, automatic changes based on additional information is questionable.

In order to give the user a flexible way to express her mind changes concerning the status of a piece of inaccurate information, MAS provides a window in which all the inaccurate information is presented. The user can browse the questions and modify the answers simply by clicking the mouse button. If the user now gives an accurate answer, the question disappears. Figure 3 shows the present implementation of such a window.

In the case of Figure 3, there are four pieces of inaccurate information of which one is shown in the window. The user has given (or MAS has concluded) a Probably yes answer to a query concerning an object called Control unit. The rest of the systems (all the other objects) is considered as System.

The status of a piece of information may change from indefinite to definite (but not to the opposite direction), and from inaccurate to inaccurate (e.g., Probably yes may change to Probably no). The change of status is naturally transitive: if a piece of indefinite information changes to be definite, it changes the status of all pieces of (indefinite) information concluded from the changed information. This requires a similar bookkeeping system which is sketched in [12] for handling user's mind changes.

The use of indefinite information allows flexible mind changes. When giving definite information, which is in contradiction with prior indefinite information, the user causes a change in the indefinite information in question and in all information possibly concluded from it. Moreover, the use of indefinite information encourages the user to make intelligent guesses. Instead of the neutral Later (or I don't know), the user can choose the side she is favoring without being absolutely sure. As a consequence, all her knowledge about the system will benefit the execution of the algorithm.

We end this chapter by briefly discussing some user interaction issues concerning the management of inaccurate information. Note that the following ideas are not yet implemented in our system.

It may be difficult for the user to keep track of the form of the information maintained in the system. Thus, the system should support the user by reminding her of the inaccurate nature of information and by informing her when the table entries are changed. The former task can be done, e.g., by using different colors when drawing the paths of a conjecture based on different form of information (paths inferred from definite information by one color and paths inferred from inaccurate information by another color). Another possibility is to animate these paths. This would focus user's attention to the questions that are still unsolved in her design effort.

It is also important that the user is informed when the (indefinite) value of a table entry is changed. This gives her a deeper understanding of the effects of her design decisions.

When implementing inaccuracy to MAS, the main question is when and how the user should be able to change her prior inaccurate answers. In our (current) implementation we allow non-automatic change of reaction only when the algorithm normally interacts with the user (e.g. in membership query). This method does not limit the user’s ability to change her mind, because the execution of the algorithm is too fast for the user to interrupt it at the specific point in any case.
Changing one’s mind after outputting the conjecture allows the user to test the influence of the different inaccurate answers. If the user doesn’t exactly know whether she should allow a certain path in the final statechart, she can give indefinite information (say Probably yes) and after looking at the generated statechart, possibly change her mind to the opposite and give a new answer (Probably no). MAS then generates a new statechart reflecting this new information.

6. Example

In this chapter we give an example to illustrate how MAS handles inaccurate information. The example system to be modeled is an elevator. Figure 4 shows a sequence diagram that depicts the usage of the elevator when going down. The Controller of the elevator first gets a request of going down from the user interface (UI) and delegates the Elevator. In the middle of the floors, the user presses “Stop” button, which stops the elevator. The Controller then gets another request of going down (continuation). After the Elevator has reached a floor, the user asks it to open the door.

To get a more interesting synthesis example, another sequence diagram is constructed. It is otherwise similar to the one in Figure 4, but the requests of going down are replaced with requests of going up. When synthesizing a statechart diagram for the Controller, these two sequence diagrams are considered. The trace constructed from the sequence diagram in Figure 4, for instance, consists of the following seven pairs:


During the synthesis, MAS asks questions that deals with the following cases:
1. Going down and/or up without stopping in the middle, and opening the door
2. Going down and/or up, stopping between the floors (perhaps several times), and opening the door after reaching a certain floor
3. Opening the door while being stopped in the middle of the floors
4. Changing the direction after stopping in the middle of the floors.

It is obvious for the user that cases 1 and 2 must be allowed. Furthermore, for the security of the passengers, case 3 cannot be allowed. The case 4, instead, is not as clear as the other cases. Assume then that during the execution of MAS the user allows case 4 by answering Probably yes. Her answer is mapped to Yes, and the synthesized statechart diagram is shown in Figure 5.

![Figure 4. An example sequence diagram](image)

![Figure 5. A statechart diagram for the Controller of an elevator](image)

The fact that the sequence diagrams are not correctly constructed is a source of most of the questions. For instance, different conditions hold when stopping between the floors and stopping when reaching a floor. That piece of information is not included in the sequence diagrams. When synthesizing a statechart diagram, MAS draws user’s attention to such design flaws.

Assume then that the user afterwards changes her mind in questions of type 4. Since her previous answer was indefinite, MAS can change it. If the new piece of information is indefinite, more similar changes can take place during the execution of the algorithm; this piece of
new information can be further changed later on. Irrespective of its type (i.e., is it definite or indefinite), a statechart diagram in Figure 6 would result.

![Figure 6. A statechart diagram that does not allow changing the direction after stopping in between the floors](image)

7. Discussion

In this paper we have discussed an interactive algorithm MAS, which synthesizes UML statechart diagrams from sequence diagrams. A synthesized statechart diagram typically generalizes the information given as sequence diagrams: in addition to the example scenarios given as an input, it accepts various other paths. MAS consults the user whenever it is not certain whether a certain generalization is allowed or not. Thus, the resulting statechart diagram does not contain automatically constructed undesired generalizations. The focus in this paper has been on the user interactions of MAS and possibilities to allow the user to give inaccurate answers during the synthesis process.

The set of sequence diagram is rarely covering enough for synthesizing a statechart diagram complete. Furthermore, the sequence diagrams may contain incomplete and even fault information. This may happen in forward engineering when the sequence diagrams are constructed manually. It may also happen in dynamic reverse engineering when prefiltering mechanisms are used for the information extracted or the run-time usage of the target system is not covering enough. MAS draws user’s attention to such critical points during the synthesis process by asking questions.

In this paper we have discussed two different kinds of inaccurate answers the user can to a membership query (i.e., if a certain generalization is allowed or not). We allow Probably yes and Probably no answers, i.e. weak Yes and No answers. These answers can be changed if the user gives contradicting information later. The user can also postpone answering by saying Later. In some cases, however, the user might be completely unable to choose between Yes and No, even when the Probably answers are available. In such cases, the user might want to answer I don’t know. When answering I don’t know the user tells to the algorithm that she is not only postponing her answer, but that she also believes not to be able to give it in the near future. The algorithm must now branch in order to simulate its function after possible answers Yes and No. The algorithm continues its execution separately in the two branches. Depending on the user’s forthcoming answers, further branching is possible. As a result, we may obtain a tree of simultaneous executions. When the user now answers a membership query, the answer can be used in all the simultaneous executions. On the other hand, different executions may pose different queries. Each execution in the tree may result a different conjecture. The user should be able to choose the one she likes best. It is likely that the size of the simulation tree, and hence, the number of alternative conjectures, does not cause problems. Namely, the tree is maintained between consecutive conjectures only. When a conjecture is given (or the best one out of the alternatives is picked up), the user edits the conjecture as she likes. The result of user’s edit operations is considered as her “true will”, and the algorithm continues with a single execution.

An obvious refinement of the system is to combine indefinite information and I don’t know answers. This gives the user five different possibilities to answer a membership query (or five different modalities): Yes, Probably yes, I don’t know, Probably no, and No. Moreover, she can postpone her answer by saying Later.

Future work on the approach in general include further decreasing the number of membership queries and the development of the user interface. So far, we have learnt that the system is efficient on tasks up to medium size, but it slows down when the observation table becomes large.

Acknowledgements. We wish to thank Jens Jahnke for his valuable comments. This research has been financially supported by the Academy of Finland (Project 35025), the Center for Technological Development in Finland (TEKES), Nokia, Neles Automation, Sensor Software Consulting, Acura Systems, and Plenware.

References


Metrics
Abstract

The research literature contains various models for estimating programming effort. Traditional, well understood, mature approaches include Putnam’s SLIM [15], Albrect’s function point method of estimation [2], as well as COCOMO [3] and COCOMO II [1, 4]. Besides these traditional approaches, various machine learning techniques, including neural networks, [16, 17, 12] have evolved. At the foundation of these models is a set of cost drivers based upon process (e.g. process maturity), project (e.g., reuse, platform), personnel (e.g. team cohesion, personnel experience), and/or product measures (e.g. size and interface). Historically, SLOC (Source Lines of Code) metric is the most popular product metric used in the formulation of the various models. It is simple to calculate and provides a quick answer. However, it is an oversimplification of the product measure contribution to what characterizes programming effort. Using only SLOC ignores the contribution of other internal product metrics, such as complexity and vocabulary, in determining programming effort. What is needed is a more representative product metric which is both simple to calculate and provides a quick answer. This paper describes a neural network approach for characterizing programming effort based on internal product measures. Over thirty-three thousand different neural network experiments were performed upon data derived from a corporate repository. Four different simple metrics (size, vocabulary, complexity, and object) are assessed in terms of their individual contribution to programming effort. Afterwards these simple metrics are combined and assessed to determine the synergistic impact of each of the combinations. Finally, a cross-validation is performed on a second corporate repository.

Keywords

Metrics, measurement, empirical analysis, effort estimation, neural networks

1. Introduction

The research literature contains various models for estimating programming effort. Traditional, well understood, mature approaches include Putnam’s SLIM [15], Albrect’s function point method of estimation [2], as well as COCOMO [3] and COCOMO II [1, 4]. Besides these traditional approaches, various machine learning techniques, including neural networks, [16, 17, 12] have evolved. At the foundation of these models is a set of cost drivers based upon process (e.g. process maturity), project (e.g., reuse, platform), personnel (e.g. team cohesion, personnel experience), and/or product measures (e.g. size and interface). Effort estimation models include some or all of these types of measures. However, product-based metrics are the most prevalent in effort estimation models. Since the inception of software metrics in the 1970s, the predominant product metric used in measuring programming effort is a Source Lines of Code (SLOC) metric. The appeal lies in SLOC’s ability to provide a quick and simple answer applicable at both the developer or project level. The appeal of simplicity is also one of the shortcomings of SLOC since it ignores other common code characteristics such as McCabe’s cyclomatic complexity [14] and Halstead’s vocabulary [11] metrics.

One of the goals of all effort estimation models is to accurately and reliably predict programming effort. Creating a highly accurate and reliable model is a challenging endeavor. One of the more accurate research models predicts effort estimates of 25% more than 75% of the time [17]. The difficulty is determining which are the answers outside the 25% accuracy. A natural research direction is to improve upon the accuracy and reliability of the models produced.

Regarding product metrics, this raises an important question, “How do various product metrics contribute to the formation of an effort estimator?” Understanding this question will lead to the improvement in the creation of effort estimation models.

In [5] and [6] it was demonstrated for simple well-understood metrics that a neural network approach could generate a network that produces results comparable to that of a traditional polynomial formulation. This validation of the neural network approach against known benchmarks (McCabe and Halstead) show the technique is sound. The
initial research was extended in a commercial setting by employing neural networks to predict areas likely to receive frequent reuses[7]. Identifying highly-reusable objects allows an organization to focus their testing efforts. Collectively all previous works demonstrate that neural networks can be applied to software metric problems in both research and commercial environments. Furthermore, comparative studies [9, 10, 13] show neural networks as a reasonable option for formulating an effort estimation model.

This paper builds upon this previous research and describes a process of applying a neural network approach to the developing of an effort estimating metric. Using data from a company which specializes in business-to-business petrochemical-industry software, over 33,000 experiments were conducted varying test suites, neural network architectures, and input parameters. Different product-based metric combinations are assessed in terms of their contribution to the formulation of an effort estimating tool. The approach cross-validated against data collected from a second corporation’s software repository.

2. General framework

The measures for building the various neural network models were extracted from a software project which allows petrochemical-based corporations to conduct complete procurement life-cycle processes over the Internet.

All data derived from the programming units were based on sole-authorship allowing for very precise measurement of effort and minimizes the noise resulting from assessing different programming abilities.

The data suite contains 104 different units. Each unit corresponds to a program written in Delphi and may contain object, variable, type, and constant definitions along with programming logic in the form of functions or procedures. In neural network terminology, each program is considered to be a vector.

For each vector, nine input measures are collected reflecting a program’s size, vocabulary, number of objects, and complexity. The output is the effort, in hours, needed to create the unit.

**Size.** The size category includes two metrics. The SLOC metric is defined as any line of code which is neither blank nor a comment, and a count of the total number of procedures and functions within the unit.

**Vocabulary.** The vocabulary category includes the four initial metrics defined by Halstead [11], total operands, total operators, unique operands and unique operators.

**Objects.** The object category includes two metrics, the total number and the unique number of objects in a unit.

**Complexity.** Complexity refers to the total complexity within a unit. It is based on McCabe’s [14] definition of complexity.

Figure 1 shows the general layout of the input and output parameters for a neural network model.

Figure 1: General layout for a Neural Network

By using a powerset combination of the input metric categories it is possible to create 15 different input configurations. These categories are:

- Size,
- Vocabulary,
- Objects,
- Complexity,
- Size and Vocabulary,
- Size and Objects,
- Size and Complexity,
- Vocabulary and Objects,
- Vocabulary and Complexity,
- Objects and Complexity,
- Size, Vocabulary, and Objects,
- Size, Vocabulary, and Complexity,
- Size, Objects, and Complexity,
- Vocabulary, Objects, and Complexity, and
- Size, Vocabulary, Objects and Complexity.

Besides grouping individual vectors into specific metric categories, 100 of the 104 vectors are distributed into ten distinct groups of ten vectors each. For a given neural network architecture and a given number of inputs, each one of the ten groups acts as a neural network test set. The vectors were sorted by effort and evenly distributed amongst the groups with the intent of creating balanced sets. Setting up different testing scenarios serves to validate the integrity of the modeling process.

The remaining four vectors contain minimum and/or maximum values for some of the inputs or output. These vectors always remained in the training set in order to avoid extrapolation problems.

The cross-product of the 15 input categories with the 10 groups yields 150 combinations for building a neural network model. The next question is what type of neural network architectures to apply. It is possible to vary the number of hidden layers along with the number of nodes per hidden layer. A decision was made to limit the maximum number of hidden layers to 3 and to limit the number of
hidden nodes per hidden layer to twice the number of inputs. One method to speed the modeling process was to increase node additions by two. To illustrate these concepts, consider a neural network with 3 inputs and 1 output. The following neural network architectures are some of the possible architectures that can be used to build various models: 3-1 (three inputs, no hidden, one output); 3-1-1 (three inputs, one hidden, one output); 3-3-1 (three inputs, three hidden, one output); 3-5-1 (three inputs, five hidden, one output); 3-1-1-1 (three inputs, one node on first hidden layer, one node on the second hidden layer, one output); 3-3-1-1; 3-5-1-1; 3-1-3-1; 3-3-3-1; 3-5-3-1; 3-1-5-1; 3-3-5-1; 3-5-5-1; 3-1-1-1-1 (three inputs, three hidden layers of one node each, one output); 3-3-1-1-1; ... ; 3-3-5-5-1; and 3-5-5-5-1.

For \( N \) inputs, the number of neural network architecture permutations equals: \( I + N + N^2 + N^3 \) where

\( I \) means there is only one neural network architecture with zero hidden layers,

\( N \) is the number of ways of creating a neural network architecture with one layer,

\( N^2 \) is the number of ways of creating a neural network architecture with two layers, and

\( N^3 \) is the number of ways of creating a neural network architecture with three layers.

Table 1 shows the total number of permutations of neural network architectures, metric categories, and group configurations.

<table>
<thead>
<tr>
<th>Metric Category</th>
<th>Number of Inputs</th>
<th>Number of Neural Network Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>Vocabulary</td>
<td>4</td>
<td>85</td>
</tr>
<tr>
<td>Objects</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>Complexity</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Size and Vocabulary</td>
<td>6</td>
<td>259</td>
</tr>
<tr>
<td>Size and Objects</td>
<td>4</td>
<td>85</td>
</tr>
<tr>
<td>Size and Complexity</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>Vocabulary and Objects</td>
<td>6</td>
<td>259</td>
</tr>
<tr>
<td>Vocabulary and Complexity</td>
<td>5</td>
<td>156</td>
</tr>
<tr>
<td>Objects and Complexity</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>Size, Vocabulary and Objects</td>
<td>8</td>
<td>585</td>
</tr>
<tr>
<td>Size, Vocabulary and Complexity</td>
<td>7</td>
<td>400</td>
</tr>
<tr>
<td>Size, Objects and Complexity</td>
<td>5</td>
<td>156</td>
</tr>
<tr>
<td>Vocabulary, Objects and Complexity</td>
<td>7</td>
<td>400</td>
</tr>
<tr>
<td>All metrics</td>
<td>9</td>
<td>820</td>
</tr>
</tbody>
</table>

Table 1: Total number of Neural Network Models

In order to build and train 33,190 neural networks, an automated neural network program was used. This program is based on Fahlman’s [8] quickprop algorithm.

In order to reduce variations between neural network models, several parameters remain constant. These include alpha, the learning rate, which is one; mu, the momentum, which also remains at one; unit type is always asymmetrical; and tolerance which is set to 30 percent. Tolerance is Mean Absolute Relative Error, or MARE. It is typically written as pred(30).

As data is read, it is normalized to values between zero and one for processing. For reporting purposes, all results are mapped back to the original ranges of a specific network.

As a way to speed up the experimentation process, the initial approach limits every neural network to 1000 epochs. Using this constraint allows all training to be completed within weeks, rather than months or years. The only unanswered question is whether this would provide enough epochs to sufficiently train the neural network. Experience in [6] and [7] has shown that most neural networks train in less than 1000 epochs, so using a longer training period is often fruitless. In the event that all the neural networks produce low correlation results, then more training will be performed.

There are many ways of measuring “successful” neural network training. Rather than commit to only one method of measuring success, several different forms of “success” are described. For every group within every metric category over all the different neural network architectures, the following seven types of “success” measures are captured:

**Training RMS error.** This is lowest error value with respect to training.

**Test RMS error.** This is lowest error value with respect to testing.

**Training Correlation.** This is the highest correlation value between the calculated and actual training outputs.

**Test Correlation.** This is the highest correlation value between the calculated and actual test outputs.

**Combined Training and Test correlation.** This case adds the training and test correlations and keeps track of the highest combined total.

**Total train correct.** This is the number of training vectors, out of a maximum of 94, that produce a value within the 30 percent threshold range, written as pred(30), of the actual training output.

**Total test correct.** This is the number of test vectors, out of a
maximum of 10, that produce a value within the threshold range, pred(30), of the actual test output.

In order to illuminate the neural network modeling process, a pseudocode representation follows. This code shows all the key loops for building all of the neural network models.

(* Pseudocode algorithm of the process *)
Alpha := 1.0;
Momentum := 1.0;
UnitType := Asym.;
Tolerance := 30%;
MaximumEpochs := 1000;
Loop through all 15 possible metric configurations
Loop through all 10 groups (* 'A' to 'J' *)
(* Loop through neural net architectures *)
HiddenLayers = 0 to 3
NodesPerLayer = 1 to 2*NumberOfInputs (in net)
(* Initialize Best Cases *)
(* This following inner loop will be *)
(* executed over 33,000 times. *)
For Epochs := 1 to MaximumEpochs do begin
  Train One Epoch;
  Test One Epoch;
  Update best case values if necessary.
  End Loop;
Save Best Case and statistical information
NEXT NodesPerLayer
NEXT HiddenLayers
End Loop (* for all 10 groups *)
End Loop (* for Metric Configurations *)

This code serves as the foundation for the actual modeling program. Running the program in batch mode generates 150 different files (15 different input configurations times the 10 different groups). Each file contains results of the 7 best cases.

3. Results: general assessment

The statistical results from analyzing a test suite are of more interest and value than from assessing training data. The test suite results provide a benchmark of how well a neural network model performs on novel data. Therefore, all the results in this section focus on the test results.

Table 2 shows the average of all 15 input configurations for each data group. The average correlation exceeds 0.9500 for 7 of the 10 groups. The average correct, pred(30), exceeds 50% for 5 of the 10 groups. A typical group, based upon median results, would have an RMS error around 0.0045, a Test Correlation around 0.9742, and an accuracy of 48%.

<table>
<thead>
<tr>
<th>Data Group</th>
<th>Ave. Test RMS Error</th>
<th>Ave. Test Correlation</th>
<th>Ave. Accuracy pred(30)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.00251</td>
<td>0.9504</td>
<td>43%</td>
</tr>
<tr>
<td>B</td>
<td>0.00380</td>
<td>0.8960</td>
<td>38%</td>
</tr>
<tr>
<td>C</td>
<td>0.00249</td>
<td>0.9028</td>
<td>59%</td>
</tr>
<tr>
<td>D</td>
<td>0.00194</td>
<td>0.9192</td>
<td>45%</td>
</tr>
<tr>
<td>E</td>
<td>0.00252</td>
<td>0.9734</td>
<td>41%</td>
</tr>
<tr>
<td>F</td>
<td>0.01052</td>
<td>0.9873</td>
<td>43%</td>
</tr>
</tbody>
</table>

Ranking each group by column and adding up the rankings shows that groups C, H, and J had the best results and group B the poorest.

Group B had median values for RMS Error, Test Correlation, and Accuracy of 0.000209, 0.9130 and 40% respectively.

<table>
<thead>
<tr>
<th>Metric Category</th>
<th>Test RMS Error</th>
<th>Test Corr.</th>
<th>Test Accuracy pred(30)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>0.000195</td>
<td>0.8288</td>
<td>30%</td>
</tr>
<tr>
<td>Objects</td>
<td>0.001240</td>
<td>0.8387</td>
<td>50%</td>
</tr>
<tr>
<td>Complexity</td>
<td>0.051631</td>
<td>0.8337</td>
<td>10%</td>
</tr>
<tr>
<td>Vocabulary</td>
<td>0.000445</td>
<td>0.9359</td>
<td>30%</td>
</tr>
<tr>
<td>Size, Object</td>
<td>0.000180</td>
<td>0.8965</td>
<td>40%</td>
</tr>
<tr>
<td>Size, Complexity</td>
<td>0.000194</td>
<td>0.8346</td>
<td>30%</td>
</tr>
<tr>
<td>Size, Vocabulary</td>
<td>0.000209</td>
<td>0.9575</td>
<td>40%</td>
</tr>
<tr>
<td>Object, Complexity</td>
<td>0.000955</td>
<td>0.8740</td>
<td>50%</td>
</tr>
<tr>
<td>Object, Vocabulary</td>
<td>0.000376</td>
<td>0.9237</td>
<td>40%</td>
</tr>
<tr>
<td>Complexity, Vocabulary</td>
<td>0.000450</td>
<td>0.9322</td>
<td>30%</td>
</tr>
<tr>
<td>Size, Object, Complexity</td>
<td>0.000178</td>
<td>0.9075</td>
<td>50%</td>
</tr>
<tr>
<td>Size, Object, Vocabulary</td>
<td>0.000184</td>
<td>0.9198</td>
<td>50%</td>
</tr>
<tr>
<td>Size, Complexity, Vocabulary</td>
<td>0.000178</td>
<td>0.9130</td>
<td>40%</td>
</tr>
<tr>
<td>All</td>
<td>0.000177</td>
<td>0.9141</td>
<td>50%</td>
</tr>
</tbody>
</table>

Group C had median values for RMS Error, Test Correlation, and Accuracy of 0.00015, 0.9254 and 60% respectively.
Group H had median values for RMS Error, Test Correlation, and Accuracy of 0.000025, 0.9956 and 60% respectively.

Group J had median values for RMS Error, Test Correlation, and Accuracy of 0.000025, 0.9956 and 60% respectively.

It is interesting to note that Group B, the lowest ranked group, did not produce the lowest individual values. The range of values for each result ranged from 0.000014 to 0.11563 (Group J) for the Test RMS Error, 0.7458 to 0.9765 for the Test Correlation (Group C), and 0 to 80 percent for the Test Accuracy (Groups H and J).

4. Metric contribution to effort

This section describes the contribution of various syntax metrics to effort. Each of the 10 groups mentioned in the previous section were sorted by Test RMS Error then ranked from 1 to 15. This process was repeated for the Test Correlation along with Test Accuracy. As a consequence of this process each of the 15 inputs (metric configurations) had 3 ratings for each of the 10 groups. Averaging the 30 ratings produced the following results presented in table 7.

<table>
<thead>
<tr>
<th>Metric Configuration</th>
<th>Average Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>2.37</td>
</tr>
<tr>
<td>Size, Object, Vocabulary</td>
<td>2.47</td>
</tr>
<tr>
<td>Size, Complexity, Vocabulary</td>
<td>4.70</td>
</tr>
<tr>
<td>Object, Complexity, Vocabulary</td>
<td>4.70</td>
</tr>
</tbody>
</table>
Table 7: Metric Contribution (3 Categories)

<table>
<thead>
<tr>
<th>Metric Configuration</th>
<th>Average Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size, Vocabulary</td>
<td>5.03</td>
</tr>
<tr>
<td>Object, Vocabulary</td>
<td>5.20</td>
</tr>
<tr>
<td>Size, Object, Complexity</td>
<td>5.27</td>
</tr>
<tr>
<td>Size, Object</td>
<td>6.23</td>
</tr>
<tr>
<td>Size, Complexity</td>
<td>8.63</td>
</tr>
<tr>
<td>Complexity, Vocabulary</td>
<td>8.93</td>
</tr>
<tr>
<td>Object, Complexity</td>
<td>9.20</td>
</tr>
<tr>
<td>Vocabulary</td>
<td>9.83</td>
</tr>
<tr>
<td>Size</td>
<td>10.43</td>
</tr>
<tr>
<td>Objects</td>
<td>11.63</td>
</tr>
<tr>
<td>Complexity</td>
<td>14.17</td>
</tr>
</tbody>
</table>

Table 7 shows that using all the syntax metrics and the Size, Object, and Vocabulary produce the best ratings for the Test data. The individual metrics did not fare as well as any combination of metrics. The general pattern suggests a very good synergy among syntax metrics for estimating effort. Ironically Complexity, as an individual metric, did not fare well at estimating effort. However, when added to any other combination it improved the average rating.

Focusing on the accuracy results only, the two best input configurations are all the metrics and Size, Object, and Vocabulary. Once again the individual metrics were not as accurate as compound metrics. Table 8 shows the summary for the accuracy ratings.

Table 8: Metric Contribution (Accuracy Only)

<table>
<thead>
<tr>
<th>Metric Configuration</th>
<th>Average Accuracy Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>1.50</td>
</tr>
<tr>
<td>Size, Object, Vocabulary</td>
<td>1.50</td>
</tr>
<tr>
<td>Object, Vocabulary</td>
<td>2.30</td>
</tr>
<tr>
<td>Size, Vocabulary</td>
<td>2.60</td>
</tr>
<tr>
<td>Size, Complexity, Vocabulary</td>
<td>2.60</td>
</tr>
<tr>
<td>Object, Complexity, Vocabulary</td>
<td>2.80</td>
</tr>
<tr>
<td>Object, Complexity</td>
<td>4.60</td>
</tr>
<tr>
<td>Size, Object, Complexity</td>
<td>5.10</td>
</tr>
<tr>
<td>Size, Object</td>
<td>5.80</td>
</tr>
<tr>
<td>Size, Complexity</td>
<td>7.20</td>
</tr>
<tr>
<td>Objects</td>
<td>7.30</td>
</tr>
<tr>
<td>Size</td>
<td>8.30</td>
</tr>
<tr>
<td>Complexity, Vocabulary</td>
<td>9.60</td>
</tr>
<tr>
<td>Vocabulary</td>
<td>10.20</td>
</tr>
<tr>
<td>Complexity</td>
<td>15.00</td>
</tr>
</tbody>
</table>

5. Cross validation

A natural question is how well the neural network predicts for a new software development project. To address this issue, 433 observations were collected from a completely different project from a completely different corporation. These observations serve as the test data for the following neural network experiments.

The input set consisted of all the syntax metrics used in the experiments described in the previous sections. Ten different neural network models are constructed. Each corresponds to the training data used in the previous sections. In essence, each training set consists of nine of the ten groups of data along with the permanent data items.

Each model is trained using the 10 different datasets. This validation approach provides a true measure of the model’s predictive abilities. This test yields an average PRED(.30) of 73.26% indicating than on average, the validation results produce estimates within 30% of the actuals 73.26% of the time.

Furthermore, there were extrapolation issues for every input and the effort output. As a consequence, the neural network could not accurately predict effort for those observations outside of the bounds of the training data. Even with the extrapolation issues, it is concluded that the neural network model has reasonably good predictive qualities.

6. Conclusions

This work describes a process of building a neural network-based model for measuring software effort. It demonstrates a process for extracting a set of software metrics from a program, associating the metrics with a program’s effort, and constructing a neural network model. An automated process creates over 33,000 different neural network models and collects data for the “best” cases. Product metrics are grouped into four categories: size, vocabulary, objects, and complexity, then analyzed for training and test cases.

Compound metrics, in the form of combinations of individual measures, generally outperformed individual measures. The results from all the experiments justify the inclusion of other product-based measures, such as cyclomatic complexity and vocabulary, into the formation of an effort estimation equation. Overall, the input configuration which included all the syntax measures produced the most accurate model against the test data than any subset combination.

The cross-validation experiments generated reasonable results and warrant further research in this area.

Using a neural network-based approach to formulate an effort estimation model allows for automated construction and assessment of effort estimation models. The actual effort
metric is embedded within the neural network architecture. At present, there is no mechanism for interpreting the architecture of the neural network. This may be perceived as a weakness of a neural network approach. The trend in Software Engineering is towards more sophisticated software processes. This implies that effort estimation models, which mirror these processes, will naturally become more complicated. As a consequence, it will be harder to create and interpret a “white-box” effort estimation model.

7. Future directions

This work focused upon product metric contribution to the formation of an effort estimation model. It raises several questions which spawn further research.

The tendency for adding more product metrics to improve results suggests adding more product metrics to the experimentation process. This could include coupling and cohesion metrics.

This research focused only on product metrics in the formulation of an effort estimation model. This research could be extended by adding process, project, and personnel type metrics. A synthesis with another model, e.g. COCOMO II, might be a logical step.

Neural network modeling is only one of several approaches for constructing an effort estimation model. Applying different modeling approaches, e.g. Case-Based Reasoning, might provide more accurate models.

8. References


Scheduling Multiprocessor Tasks in Presence of the Correlated Failures

Piotr Jędrzejowicz, Ireneusz Czarnowski, Ewa Ratajczak, Izabela Wierzbowska
Department of Information Systems, Gdynia Maritime Academy, Gdynia, Poland
{pj, irek, ewra, iza}@wsm.gdynia.pl

Walter J. Gutjahr
Department of Statistics and Decision Support Systems, University of Vienna, Austria
walter.gutjahr@univie.ac.at

Aleksander Skakowski
Department of Computer Science, Gdynia Maritime Academy, Gdynia, Poland
askakow@wsm.gdynia.pl

Abstract

The paper deals with scheduling multiple processor (m-p) tasks on multiple processors to maximize overall schedule reliability under the requirement that all tasks meet their deadlines. An m-p task can be represented in a schedule by one or more independently developed variants. We allow correlation of variant failures within a task and model the correlation using a Beta distribution. Because the considered problem belongs to NP-hard class four different approximation algorithms based on soft-computing methods have been proposed to solve the problem. The algorithms include island-based evolutionary, neural network, population learning, and tabu search approaches. Experiment results show that the proposed algorithms find good quality solutions in a reasonable time.

1. Introduction

As it was pointed out by [7] and [6], there exist scheduling problems where tasks are processed on more than one processor at a time. During the execution of these multiprocessor (m-p) tasks communication among processors executing the same task is implicitly hidden in a „black box” denoting an assignment of this task to a subset of processors during some time interval. The paper deals with scheduling multiple variant software on multiple processors to maximize schedule reliability. Multiple variant software is modeled by m-p tasks. The proposed approach addresses one of the frequently encountered conflicts involving high dependability and safety standards required versus system performance and cost.

One of the potential application areas for multiprocessor task scheduling is the fault-tolerant computing under time constraints. Among several techniques used to achieve software fault tolerance, some best known include N-version programming (NVP) [3], recovery blocks (RB) [23], [20], and N-version self-checking programming (NSCP) [29], [21]. A hybrid solution integrating NVP and recovery blocks is known as the consensus recovery blocks [26]. In all the above listed techniques, the required fault tolerance is achieved by increasing the number of independently developed program variants, which in turn leads to higher reliability at a cost of the additional resources used. It should be noted that the m-p task concept could be used to model variety of the fault-tolerant structures, since all require processing of redundant variants, internal communication and execution of an adjudication algorithm. In such a model, the size of a task would correspond to the number of its redundant variants.

Scheduling m-p tasks is understood as assigning processing elements to tasks and, at the same time, deciding on task size and structure, in such a way that all constraints are satisfied and some overall performance goal is optimized. Scheduling m-p tasks differs from traditional, single-variant task scheduling problems, by the extended solution space. It includes not only an assignment of tasks to processors but also a decision as to which combination of redundant variants, from a set of the available ones, should be used to construct each task (at least one variant of each task has to be included within a schedule). Algorithms for scheduling multiprocessor tasks constructed from multiple program variants have been proposed in [19], [8]. The discussed algorithms are based on the assumption that failures of multiple program variants are statistically independent.
The present approach differs by allowing correlation of failures within a task. Hence, the earlier assumption is not needed any longer.

In the paper global optimization criterion is schedule reliability understood as a probability that all scheduled tasks will be executed without failures. Schedule reliability is to be maximized under the requirement that all tasks meet their respective deadlines. Unfortunately the discussed scheduling problem belongs to the NP-hard class. To enable obtaining solutions within reasonable time the following approximation algorithms, based on soft-computing methods, have been designed and tested:

- island-based evolution algorithm (IBEA);
- neural network algorithm (NNA);
- hybrid 3opt-tabu search algorithm (TSA);
- population learning scheme (PLS).

The paper is organized as follows. Section 2 contains task description and its reliability model. Section 3 contains problem formulation. Section 4 gives short description of the proposed approximation algorithms. Section 5 contains numerical example and the results of computational experiments carried. Finally, in Section 6 suggestions for future work and conclusions are made.

2. Multiprocessor task and its reliability model.

A set $N$ of $n$ of multiprocessor tasks to be processed under time constraint is considered. A multiprocessor (m-p) task is built from a finite set of possibly independently developed program variants. It is assumed that scheduling such a task requires two types of resources – processors and time. Number of processors needed is equal to the number of program variants used to construct an m-p task. Time required to process an m-p task can be calculated from its variant processing times $p_{ji}$, $i = 1, \ldots, n$, $j = 1, \ldots, N V_j$, where $N V_j$ is a number of available variants. Processing time of the task $j$ is calculated as:

$$p_j = \max_i \{p_{ji}\} + o_j; i \in G_j \text{ if } |G_j| \geq 2 \quad p_{ji} \text{ otherwise}$$

where: $o_j$ is an overhead time and $G_j$ is a set of variant indexes indicating variants used to construct task $j$.

From the problem assumptions it is clear that tasks are allocated to processors only after their size and the internal structure, i.e. combination of variants to be processed, have been chosen. It also follows from these assumptions that when constructing a task from, say, $x$ available variants there are $2^x - 1$ possible structures to be considered.

To reduce the solution space a concept of an effective task structure is introduced. For a task $j$ of the size equal to $x$ ($1 \leq x \leq N V_j$) there are at most $N V_j + 1 - x$ effective structures. A structure of a given size is considered effective if there exists no other combination of program variants with the same (or shorter) processing time and higher reliability. Generating effective structures for a given task size is straightforward and requires polynomial time. Altogether, for a task $j$ with the number of available variants equal to $N V_j$, there are, at most, $N V_j(N V_j + 1)/2$ effective structures.

A nontrivial aspect of our investigation is the computation of the task reliability $R_j$ from the variant reliabilities $r_i$, $i \in G_j$. As stated in Section 1, we want to overcome the idealized assumption that redundant variants fail independently from each other. Models for the reliability computation of fault-tolerant structures with variants the failures of which are correlated have been developed in [10], [25] and [27]. These approaches, however, assume that the estimated reliability values of the single variants of a task are all equal, which is a too severe restriction within our context. We need an extended model where these estimates may vary. Nevertheless, in the special case of equal single reliability estimates, the model should be consistent with the framework proposed in the articles cited above.

In the following, we describe an extension of the Nicola-Goyal approach serving this purpose. The formal background of this extension is presented in more detail in [16]. Here, we concentrate on the basic ideas. For simplicity, let us assume that $G_j = \{1, \ldots, k\}$ with $1 \leq k \leq N V_j$ (which can always be achieved by a suitable re-assignment of the variant indices).

The key for an appropriate modeling of failure dependencies between redundant program variants has been given in [10]. There is no reason why independently developed variants should not fail independently on some fixed program input $y$; nevertheless, on a random program input $Y$, the failure events for different variants get correlated simply by the fact that not all inputs $y$ have the same probability leading to a program failure: some of them are very easy to process, such that they are most probably not failure-causing, while others are difficult to process, such that all variants may have a high probability of failing on them. This induces a positive correlation between the failures of the variants in an execution of the task.

Conceptually, we may consider a population $Pop$ of
programs potentially written to the given specification, and assume that the program variants 1, ..., k are (randomly) selected from subsets Popi, ..., Popk of Pop, respectively. By \( p(y) \) and \( p_i(y) \), \( 1 \leq i \leq k \), we denote the fraction of programs in Pop in Popi that fail on input \( y \). Now let us suppose that also the input is selected randomly according to some usage distribution \( Q \), i.e., the input is a random variable \( Y \). Then, the fractions of failing programs in Pop resp. in Popi, \( p(Y) \) resp. \( p_i(Y) \), are random variables as well. The random variable \( p(Y) \) can be interpreted as the probability that during operational usage, a program variant selected arbitrarily from Pop will fail, and \( p_i(Y) \) has analogous interpretation.

Nicola’s and Goyal’s approach is based on the assumption that \( p(Y) \) has a Beta distribution with parameters \( \alpha \) and \( \beta \), which are chosen as follows:

\[
\alpha = \pi / \theta, \quad \beta = (1-\pi) / \theta.
\]

Therein, \( \pi \) is the estimated unreliability of a variant drawn from Pop (i.e., the expected value of \( p(Y) \)), and \( \theta \geq 0 \) is a correlation level, where the boundary case \( \theta = 0 \) represents failure independence, whereas the other boundary case \( \theta = \infty \) represents identical failure behavior of all versions (one of them fails exactly if all others fail).

Let us say a few words of explanation to the choice of Beta distributions. This type of model distributions is widely used, especially in Bayesian statistics, but also in other areas (such as, e.g., project planning by PERT). They combine the advantage of easy computational tractability with flexibility in application: mean value and variance can be tuned similarly as by using the normal distribution, but contrary to the last, also skew distributions can be modeled, and the range of possible values can be restricted to an interval (in our case, the interval \([0,1]\) is required). In principle, we could adapt our technique also to other families of distributions, but the family of Beta distributions seems to be the most appropriate from the viewpoint of ease of application.

In our extended model, we assume that also the random variables \( p_i(Y) \), \( 1 \leq i \leq k \) have Beta distributions, with parameters \( \alpha_i \) and \( \beta_i \), chosen as follows:

\[
\alpha_i = \pi_i / \theta, \quad \beta_i = (1-\pi_i) / \theta,
\]

where \( \pi_i \) is the estimated unreliability (probability of failing) of variant \( i \) drawn from Popi, and \( \theta \) is the correlation level as explained above.

Furthermore, a certain functional relation between \( p(Y) \) and \( p_i(Y) \) is assumed. We suppose that the fraction \( p(Y) \) of programs in Pop failing on a certain input \( y \) already determines the fractions \( p_i(Y) \) of programs in Popi failing on this input \( y \). Of course, this functional dependency (we denote it by \( \varphi_i \)) cannot be chosen arbitrarily. Instead, it must be established in such a way that the distribution \( \text{Beta}(\alpha_i, \beta_i) \) of \( p(Y) \) is transformed just to the distribution \( \text{Beta}(\alpha, \beta) \) of \( p_i(Y) \).

In [16] it is shown that the following function \( \varphi_i \) satisfies this requirement: Let \( F \) be the distribution function of \( \text{Beta}(\alpha, \beta) \), and let \( F_i \) be the distribution function of \( \text{Beta}(\alpha_i, \beta_i) \). We set

\[
\varphi_i(p) = F_i^{-1}(F(p)).
\]

The relation between \( p(Y) \) and \( p_i(Y) \) is then given by

\[
p_i(Y) = \varphi_i(p(Y)).
\]

On the assumptions above, the unreliability \( unrel = 1 - R_j \) of task \( j \), i.e., the probability that all its variants \( i (i = 1, \ldots, k) \) fail on a random input \( Y \), can be computed by elementary probabilistic calculations. One obtains:

\[
unrel = \frac{1}{0} \int \varphi_1(p) \cdots \varphi_k(p) f(p) dp
\]

where \( f(p) = F'(p) \) is the first derivative of the distribution function \( F \), i.e., the corresponding density function.

As it has been demonstrated in [16], the value \( unrel \) computed from the formula above is independent of the chosen basic distribution \( F \); another \( F \) yields another density \( f(p) \), but also other transformation functions \( \varphi_i(p) \), leading finally to the same computation result.

It is well-known that the distribution function \( F \) of a Beta(\( \alpha \), \( \beta \)) distribution is given by

\[
F(p) = p^{\beta-1} (1-p)^{\alpha-1} / \text{Beta}(\alpha, \beta),
\]

where \( \text{B}(\alpha, \beta) \) is the Beta function. Using this fact and a Taylor expansion of the Beta function, we obtain the following numerical procedures for the computation of \( unrel \):

Function \( \text{B}(\alpha, \beta, x) \)

Begin

value := 0;
for \( j := 0, 1, \ldots, \beta-1 \) do
value := value + \((-1)^{j} (\beta-1)! x^{\alpha+j} / (j! (\beta-1-j)! (\alpha+j))\);
return value;
end.
Function $\phi(\alpha, \beta, \alpha_0, \beta_0, p)$

Begin

$c := B(\alpha_0, \beta_0, p) * B(\alpha, \beta, 1) / B(\alpha_0, \beta_0, 1);$  
$x := 0.1;$

while $(B(\alpha, \beta, x) > c)$ do

$x := x / 0;$

repeat until accuracy suffices

$x := x - (B(\alpha, \beta, x) - c) / (x^{\alpha-1} * (1-x)^{\beta-1});$

return $x$;

end.

Function integrand $(k, \alpha[], \beta[], \alpha_0, \beta_0, p)$

Begin

$q := 1;$

for $i := 1$ to $k$ do

$q := q * \phi(\alpha[i], \beta[i], \alpha_0, \beta_0, p);$  
return $q * p^{n-1} * (1-p) \beta_0-1 / B(\alpha_0, \beta_0, 1);$  
end.

end.

Function unrel $(k, \alpha[], \beta[], \alpha_0, \beta_0, p)$

Begin

return result of numerical integration of integrand $(k \alpha[], \beta[], \alpha_0, \beta_0, p)$ between $p = 0$ and $p = 1$, e.g., by Simpson’s rule;

end.

Therein, $\alpha_0$ and $\beta_0$ are the parameters chosen for the basic Beta distribution $F$, and $\alpha[]$ and $\beta[]$ are the arrays of the numbers $\alpha_i$ and $\beta_i$. Since $\beta_i = (1 - \pi_i) / \theta_i$, and since $\pi_i$ and $\theta$ are small in practical applications, the values $\beta_i$ can be rounded to integers. The procedures above are based on the assumption that this has been done.

3. Problem formulation.

A set $N$ of $n$ of multiprocessor tasks to be processed under time constraint is considered. It is assumed that the following information with respect to each task in $N$ is available:

- ready time - $a_j$, $j = 1, \ldots, n$
- deadline - $d_j$, $j = 1, \ldots, n$
- number of available variants - $NV_j$, $j = 1, \ldots, n$
- variant processing times - $p_{ji}$, $j = 1, \ldots, n$, $i = 1, \ldots, NV_j$
- variant reliabilities - $r_{ji}$, $j = 1, \ldots, n$, $i = 1, \ldots, NV_j$
- level of failure correlation between variants - $0_j$, $j = 1, \ldots, n$

The considered problem of scheduling multiprocessor tasks, denoted using Graham’s [15] notation as $P|\alpha_m p|\beta R$, is characterized by a set of multiple, identical processors $P$, and a set of multiple-variant tasks $N$. Each task has the maximum size $NV_j$, and the minimum size equal to 1. Tasks are independent and non-preemptable with ready times and deadlines differing per task. Tasks have processing times, which may differ per combination of variants chosen for execution. Task $j$ of the size $|G_j| = m$ requires $m$ parallel processors for processing. Optimization criterion is schedule reliability calculated as $R = \Pi R_i$. Task reliabilities are calculated using reliability model described in Section 2. In general failures of variants within a task can be statistically correlated. Decision variables include assignment of tasks to processors and combination of variants to be executed for each task. Tasks can not be delayed.

4. Scheduling algorithms.

4.1. Island based evolution algorithm – IBEA.

An island based evolution algorithm (IBEA) belongs to the distributed algorithm class. To improve efficiency of genetic algorithms (GA) several distributed GA’s were proposed in [14], [4], [2]. Their ideas included an island-based approach where a set of independent populations of individuals evolves on “islands” cooperating with each other. The island-based approach brings two benefits: a model that maps easily onto the parallel hardware and extended search area (due to multiplicity of islands) preventing from sticking in local optima. Promising results of the island-based approach motivated the authors to design the IBEA for scheduling multiple-variant tasks.

The proposed island-based evolutionary algorithm (IBEA) works on two levels with two corresponding types of individuals. To evolve individuals of the lower level a population-based evolutionary algorithm (PBEA) is proposed. On the higher level the following assumptions are made:

- an individual is an island $I_k$, $k = 1, 2, \ldots, K$, where $K$ is the number of islands;
- an island is represented by a set of the lower level individuals;
- all islands are located on the directed ring;
- populations of lower level individuals evolve on each island independently;
- each island $I_k$ regularly sends its best solution to the successor $I_{k mod K+1}$ in the ring;
- the algorithm stops when an optimality criterion is satisfied or the preset number of generations on each island have been generated;
- when IBEA stops the best overall solution is the final one.
On the lower level the following assumptions are made:

- an individual is represented by an n-element vector \( S_a = \{ c^j_i \mid i = 1, 2, \ldots, n; 1 \leq j \leq n \} \), used to construct a list of tasks, where \( i \) is an index describing a place of the task \( j \) on the list, and \( c^j_i \) is a code representing the combination of variants used to construct task \( j \);
- all \( S_a \) representing feasible solutions are potential individuals;
- an initial population is composed in part from the potential individuals for whom combination and order of tasks on the list is random, and in part from the potential individuals for whom combination of tasks is random with order determined by a non-decreasing ready time as the first criterion and a non-decreasing deadlines as the second;
- each individual can be transformed into a solution by applying LSG-\( P(a, m-p|R) \), which is a specially designed algorithm for list-scheduling \( m-p \) tasks;
- each solution produced by the LSG-\( P(a, m-p|R) \) can be directly evaluated in terms of its fitness;
- new population is formed by applying several evolution operators: selection and transfer of some more "fit" individuals, random generation of individuals, crossover, and mutation.

The following pseudo-code shows main stages of the IBEA-\( P(a, m-p|R) \) algorithm:

Procedure IBEA-\( P(a, m-p|R) \)

Begin

Set number of islands \( K \), number of generations \( PN \) for each island, size of the population for each island \( PS \). For each island \( I_o \) generate an initial population \( PP_o \).

While no stopping criteria is met do

For each island \( I_o \) do

Evolve \( PN \) generations using PBEA.

Send the best solution to \( l_{(k \mod K) + 1} \).

Incorporate the best solution from \( l_{(K(k - 2) \mod K) + 1} \) instead of the worst one.

EndWhile

Find the best solution across all islands and save it as the final one.

End.

PBEA-\( P(a, m-p|R) \) algorithm is shown in the following pseudo-code:

Procedure PBEA-\( P(a, p, size)|R \)

Begin

Set population size \( PS \), generate a set of \( PS \) individuals to form an initial population \( PP_o \);

Set \( ic := 0 \); (\( ic \) - iteration counter);

While no stopping criteria is met do

Set \( ic := ic + 1 \)

Calculate fitness factor for each individual in \( PP_{ic+1} \) using LSG-\( P(a, m-p|R) \);

Form new population \( PP_{ic+1} \):

- Select randomly a quarter of \( PS \) individuals from \( PP_{ic+1} \) (probability of selection depends on fitness of an individual);
- Produce a quarter of \( PS \) individuals by applying crossover operator to previously selected individuals from \( PP_{ic+1} \);
- Produce a quarter of \( PS \) individuals by applying mutation operators to previously selected individuals from \( PP_{ic+1} \);
- Generate half of a quarter of \( PS \) individuals from set of potential individuals (random size, and order);
- Generate half of a quarter of \( PS \) individuals from set of potential individuals (random size, and fixed order).

EndWhile

End.

LSG-\( P(a, m-p|R) \) algorithm used within PBEA-\( P(a, m-p|R) \) is carried in the three steps:

Procedure LSG-\( P(a, m-p|R \)

Begin

Step 1. Construct a list of tasks from the code representing individuals. Set loop over tasks on the list.

Step 2. Within the loop, allocate current task to multiple processors minimizing the beginning time of its processing. Continue with tasks until all have been allocated.

Step 3. If the resulting schedule has task delays, a fitness of the individual \( S_a \) is calculated as \( R_a = -(1 - \Pi R_j) \) where \( j \) belongs to a set of the delayed tasks. Otherwise, \( R_a = \Pi R_j, j=1, \ldots, n \).

End.

4.2. Neural network algorithm – NNA.

Neural Networks have been used to obtain solutions of problems belonging to NP-hard class (see for example [17]). NNA have been also applied to job-shop scheduling [30], [18], [5], and [1]. The NNA proposed in this paper is based on a dedicated neural network. To train the network an evolution algorithm is used. The architecture of the NNA - \( P(a, m-p|R) \) depends on a number and properties of \( m-p \) tasks, which are to be scheduled. The set of tasks is partitioned to form subsets of tasks with
identical ready times. The number of such subsets corresponds to the number of layers in the neural network. Each layer consists of neurons representing the tasks to be scheduled. The described architecture is additionally extended and includes elements known as adders. These elements are situated in each layer. Number of adders at each layer corresponds to the number of available processors. The neurons of each layer send their signals to adders. The neurons are connected to adders and values of their weights define neuron activity. Connection weights between neurons and adders can have binary values \{0,1\} that mean subsequently connection absence or presence. At the moment of algorithm initiation weights values assume random values. The neurons of each layer send their signals to adders. Signals represent variant processing time. Adders realise heuristic function of activation, which sums input signals and produce signals representing deadlines of the tasks to be executed on processors. Signals between layers pass through decision blocks, which are equivalent to decoders. The decision blocks take decisions with respect to correcting (or not) signals from adders. They may restructure connections between neurons and adders in a layer if signals from adders require such a correction. If the change of the connections does not yield a positive result then a decision block can simulate an action of the decision block in earlier layer if it exists. Decision blocks adjust connection weights in learning process using a classical genetic algorithm similar to [24] and [28]. Weights are adjusted after the end of the subsequent epoch. This method is known as batch training procedure [9].

During the learning process a genetic algorithm is searching for suitable weights of connections. Each neuron must send a signal to at least one adder. Chromosome for a layer is a string of bits whose length depends on number of tasks and processors in this layer. In searching for weights of connections the genetic algorithm uses standard genetic operators: one-point crossover and mutation. Crossover is based on exchanging parts of bit strings between two chromosomes. Operation of mutation is realized by random changes of weight values. In selection process elitist method is used.

The proposed Neural Network Algorithm involves executing steps shown in the following pseudo-code:

```plaintext
Procedure NNA-P[a, m-p|R
Begin
Set number of layers k;
While no stopping criteria is met do
  For i :=1 to k do
    Train network using GA;
  endfor
End
```

Evaluate the objective function for the obtained network structure;
If the objective function is improved then best network structure := obtained structure;
endwhile
Treat the best network structure as the final one and decode problem solution;

4.3. Hybrid 3opt-tabu search algorithm – TSA.

The proposed approach is an extension of the tabu-search meta-heuristic [12], [13], [11]. The algorithm TSA-P[a, m-p|R is based on the following assumptions and notations:

- a solution \( Q \) is a list of objects, each representing a single variant of a task. An object has attributes including task number \( j, \) \( j=1, ..., n \) and variant number \( i, \) \( i=1, ..., NV_j \). At least one variant of each task has to be scheduled;
- for each processor \( b \) a list of objects is created. For \( k \) processors there are \( k \) such lists. Originally lists are generated by consecutively allocating objects to the least loaded processor;
- objects on the lists are ordered using the 3-opt algorithm (i.e. searching for the best schedule for the first three objects, then 2\textsuperscript{nd}, 3\textsuperscript{rd}, and 4\textsuperscript{th}, etc.);
- a move involves adding or erasing a variant \( i \) of the task \( j \) from the list owned by a processor \( b \);
- attributes of the move are denoted as \( m(b, j, i, h, ea, mr) \), where \( b, j, i \) are the respective indexes, \( h \) is a position of task \( j \) on the respective list, \( ea \) is a move status (+1) for adding, and (-1) for erasing, and \( mr \) is a move value;
- \( N(Q) \) denotes a set of possible moves from \( Q \), called neighbourhood of \( Q \);
- \( N'(Q) \) denotes a modified neighbourhood of \( Q \) that is neighbourhood without all tabu-active moves and moves of which it is easy to say that they will not improve the current solution;
- \( TAM \) denotes list of tabu-active moves;
- \( SM \) denotes a short term memory of the tabu algorithm, where executed moves are remembered.
TSA-P[a, m-p|R involves executing steps shown in the following pseudo-code:

```plaintext
Procedure TSA-P[a, m-p|R
Begin
SM=Ø;
TAM=Ø;
Generate two initial solutions:
```
one shortest variant for each task order by their non-decreasing ready times as the first criterion and non-decreasing deadlines as the second; $Q_2$ - one shortest variant for each task order by their non-decreasing deadlines as the first criterion and non-decreasing ready times as the second; Calculate total tardiness; Try to improve both solutions using the 3-opt approach providing the total tardiness value remains zero; Set number of iterations $it$; If total tardiness > 0 then scheduling problem has no solution; else For both initial solutions $Q_1$ and $Q_2$ do $\max = \text{reliability of the initial solution}$; For 1 to $it$ do While best move is not chosen do Consider all moves in $N'(Q_i)$, select and remember the best move (with the best move value) $m_{best}\left(b, j, i, h, ea, mr\right)$; If the best move cannot be found then delete from TAM all moves with the least $tabu_{tenure}$; endwhile Add the new solution to the SM; Update TAM: Add $m_i\left(b_{best}, 0, 0, h_{best}, 0, 0\right)$, $it/4$; Add $m_k\left(b_{best}, 0, l_{best}, 0, -1, 0\right)$, $it/6$; Add $m_t\left(b_{best}, l_{best} - ea_{best}, 0, 0\right)$, $it/2$; Delete moves with $tabu_{tenure} = 0$; If $mr_{best} > \max$ then $\max = mr_{best}$ endif endfor Choose the best solution from the solutions found for $Q_1$ and $Q_2$; endif endif

4.4. Population learning scheme - PLS.

To obtain a solution of the $P[a, m-p]R$ problem we have adopted the Lin-Kerninghan algorithm [22] originally applied to solving travelling salesman problems, and used it as a population learning tool. The approach is based on the improvement procedure, which can be applied to the existing schedule, possibly not even a feasible one. The procedure aims at improving a schedule by reducing the number of the delayed tasks and, at the same time, increasing schedule reliability. Within the proposed PLS the improvement procedure is applied to a population of solutions. Constructing a list of tasks, sorting it according to non-decreasing deadlines and allocating tasks from this list to least loaded processors generates an initial population. Tasks are allocated to processors in order of their appearance on the list with some probability $q$ and randomly with probability $1-q$. For each task a number of program variants to be executed is randomly drawn.

The improvement procedure aims at maximizing schedule reliability minus the number of delayed tasks. Its idea is to consider one by one the delayed tasks. For each task $z$ a set $K$ of other tasks, such that inserting task $k \in K$ instead of task $z$ best improves value of the goal function, is constructed. As the result a schedule with a double task $k$ appears. Improvement is again applied to the second appearance of the task $k$. If set $K$ can not be constructed a schedule is upgraded in such a way that each task is scheduled exactly once and the result compared with the best schedule achieved so far. At the first two levels of the recursion set $K$ contains 5 elements, later on, only 1 element.

5. Numerical example and the results of computational experiments.

As an example the $P[a, m-p]R$ problem with 11 tasks and 5 processors has been solved using all of the proposed approaches. The respective data is shown in Table 1.

All algorithms have generated feasible schedules – that is schedules without delayed tasks, with at least one variant of each task scheduled, and all tasks having the required resources (processor and time) allocated. The respective probabilities of the successful completion of all tasks are:

- island based evolution algorithm – 0.917186;
- neural network algorithm – 0.89425;
- hybrid 3opt-tabu search algorithm - 0.915393;
- population learning scheme - 0.917186.

Schedules generated by the respective algorithms are shown in Figure. 1.

To evaluate the proposed algorithms computational experiment has been carried. It has involved 20 randomly generated problems. Numbers of tasks per problem have been generated as random integers from the uniformly distributed interval $U[10,28]$, numbers of available processors from $U[2,12]$, numbers of variants per task from $U[2,5]$, variant processing times from $U[1,15]$. Variant reliabilities have been generated as random real numbers from $U[0.8999,0.9999]$, and variants failure
Table 1. Example $P|\alpha\mu$, $m$-$p|R$ problem

<table>
<thead>
<tr>
<th>task number</th>
<th>variants per task</th>
<th>variant number</th>
<th>arrival time $a_i$</th>
<th>variant processing time $p_{ij}$</th>
<th>deadline time $d_j$</th>
<th>variant reliability $r_{ij}$</th>
<th>correlation between variant failures</th>
<th>time overhead $o_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>0.925</td>
<td>0.1705</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>0.9301</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>0.9365</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>0.9873</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>9</td>
<td>0.9998</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>16</td>
<td>0.945</td>
<td>0.1533</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>16</td>
<td>0.9569</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>16</td>
<td>0.96</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>16</td>
<td>0.9723</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>4</td>
<td>16</td>
<td>0.9825</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>21</td>
<td>0.9029</td>
<td>0.1579</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>21</td>
<td>0.9563</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
<td>21</td>
<td>0.9762</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>7</td>
<td>21</td>
<td>0.9835</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>25</td>
<td>0.9441</td>
<td>0.1289</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>8</td>
<td>25</td>
<td>0.9619</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td>25</td>
<td>0.9558</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>9</td>
<td>25</td>
<td>0.9608</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>19</td>
<td>0.9143</td>
<td>0.1301</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>19</td>
<td>0.923</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>19</td>
<td>0.9312</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>19</td>
<td>0.9337</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>19</td>
<td>0.9801</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
<td>9</td>
<td>8</td>
<td>25</td>
<td>0.906</td>
<td>0.1774</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>8</td>
<td>25</td>
<td>0.9092</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>10</td>
<td>25</td>
<td>0.9237</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>11</td>
<td>25</td>
<td>0.9254</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>11</td>
<td>25</td>
<td>0.9367</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
<td>13</td>
<td>5</td>
<td>26</td>
<td>0.9074</td>
<td>0.1014</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>5</td>
<td>26</td>
<td>0.9497</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>6</td>
<td>26</td>
<td>0.9633</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>1</td>
<td>14</td>
<td>6</td>
<td>30</td>
<td>0.9029</td>
<td>0.176</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>7</td>
<td>30</td>
<td>0.9448</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>7</td>
<td>30</td>
<td>0.984</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>8</td>
<td>30</td>
<td>0.9896</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>1</td>
<td>18</td>
<td>5</td>
<td>27</td>
<td>0.9274</td>
<td>0.1814</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>6</td>
<td>27</td>
<td>0.9794</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
<td>23</td>
<td>5</td>
<td>32</td>
<td>0.9187</td>
<td>0.1709</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>6</td>
<td>32</td>
<td>0.9621</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>1</td>
<td>26</td>
<td>2</td>
<td>32</td>
<td>0.9047</td>
<td>0.1045</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>26</td>
<td>2</td>
<td>32</td>
<td>0.9749</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>2</td>
<td>32</td>
<td>0.9831</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>2</td>
<td>32</td>
<td>0.9915</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
a) island based evolution algorithm and population learning scheme

b) neural network algorithm

c) hybrid 3opt-tabu search algorithm

Figure 1. Schedules generated by the proposed algorithms
correlation from U[0,1,0.2]. The following measures have been used to evaluate the algorithms:

- mean relative error related to the best known solution - MRE;
- maximum relative error encountered - Max RE;
- minimum relative error encountered - Min RE;
- best result ratio - BRR.

Results of the experiment are shown in Table 2.

Experiment results show that the proposed algorithms generate good quality solutions. The hybrid 3opt-tabu search algorithm outperforms the remaining ones in terms of both - mean relative error and best result ratio. It also has a very low maximum relative error level. Island based evolution algorithm also offers good performance and a low maximum relative error level. However both approaches require more computational resources than the remaining two. Population learning scheme has been faster than IBEA by the factor of 2 to 3. (It has taken IBEA about 9-15 minutes to find a solution on Pentium II-MMX CPU, 300MHz).

<table>
<thead>
<tr>
<th>Measure</th>
<th>IBEA</th>
<th>NNA</th>
<th>TSA</th>
<th>PLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRE</td>
<td>0.371%</td>
<td>6.501%</td>
<td>0.243%</td>
<td>1.255%</td>
</tr>
<tr>
<td>Max RE</td>
<td>2.21%</td>
<td>18.01%</td>
<td>2.21%</td>
<td>3.34%</td>
</tr>
<tr>
<td>Min RE</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>BRR</td>
<td>31.6%</td>
<td>10.5%</td>
<td>47.4%</td>
<td>15.8%</td>
</tr>
</tbody>
</table>

6. Conclusion.

Main contribution of the paper is a successful application of several soft computing approaches to solving some difficult scheduling problems including scheduling software components with correlated failures. Computational experiments have shown that even quite basic and simple computational schemes based on generic metaheuristics can produce competitive results. It has been also shown that novel modelling approach overcoming the idealized assumption that redundant variants fail independently from each other lends itself to optimization and could be useful in generating realistic schedules for systems involving fault-tolerant program structures run under hard time constraints. The presented approach has been limited to a static case where information on set of tasks to be run is known beforehand. Further research should lead into investigating construction of dynamic schedulers where the suggested approach (static one) could be used as a base for an online scheduler. Another direction of research is further improvement of the soft computing algorithms and techniques resulting in improving either quality of result or reducing the demand for computational resources.

References

Acknowledgment: This research has been supported by the KBN grant nr 8T11 02019.
Program Understanding
Expressing Similarity in Software Engineering:
A Neural Model

W. Pedrycz, G. Succi, M. Reformat, P. Musilek, X. Bai
Dept. of Electrical & Computer Engineering
University of Alberta, Edmonton, Canada

Abstract
One of the key objectives of Quantitative Software Engineering is to develop efficient models of software products and software processes. Revealing essential and data-legitimized relationships hidden in software data and refining them in a form of detailed models have triggered interest in pursuing advanced techniques of Computational Intelligence in Software Engineering. Neural networks, and self-organizing maps, in particular, have been used in a number of Software Engineering studies devoted to the visualization of complexity of software artifacts and quantization of relationships between software entities (modules, classes, etc.). In this paper, we are interested in expressing similarity (proximity) between software artifacts. Owing to the abstract nature of software constructs, the intent is to depart from “standard” and explicit instruments such as distances (Euclidean, Hamming, etc.) and focus on implicit and experiment-driven models of similarity designed in the framework of neural networks. We propose a new referential topology of a neural network and reveal that its learning gives rise to a detailed model of similarity. Next, we show how to augment self-organizing maps by incorporating the neural proximity model. In this setting, we elaborate on the issue of duality of supervised – unsupervised learning and discuss an important matter of data – knowledge reconciliation.

The study is illustrated with the use of numerical examples. We consider synthetic data as well as highly dimensional data coming from software projects. The material is organized in 5 sections. First, in Section 2 we introduce a neural model of similarity and motivate its origin. Section 3 contains a detailed description of a hybrid neural architecture for supervised and unsupervised learning. In Section 4 the results of experimental studies are presented, where the phase of supervised learning is followed by the hybrid version of the SOM learning. Section 5 ends the study with some conclusions.

2. The neural model of similarity

The problem of expressing similarity (proximity) between two entities (say, two software modules, two software projects, etc.) arises as a fundamental issue, no matter what specific task (classification, prediction, description, data visualization) one is interested in. When the complexity of the problem increases, the choice of a proper similarity measure becomes critical. The difficulties are inherently associated with the complexity of the entities whose similarity one has to determine. This complexity facet permeates the entire area of Software Engineering and is quite commonly elevated by the abstract nature of its constructs [1][2][10][11][12]. Software projects, software modules or classes are excellent examples of artifacts where similarity determination remains an open issue.

Various distance functions (say Hamming, Euclidean, Tchebyschev) have been regarded as a common way of quantification of similarity. Evidently, the lower the distance between two entities, the higher the similarity between them. While this is a sound assertion, the point of debate is about a selection of a suitable distance. Two signals are similar if their Euclidian distance is small. Two filters perform in a similar way (they are close each other)
if their impulse responses are close in terms of the Tchebyschev distance. On the other hand, why should the similarity of two texts be expressed by Hamming distance? To which extent is the Euclidean distance suitable to measure similarity in the space of software measures (metrics)? By and large, the similarity in Software Engineering could be a matter of “perceptual” similarity expressed by software system architect, system designer, testing engineer, etc. In what we propose, is a principle of knowledge – data reconciliation whose realization is carried out in the framework of neural networks and neurocomputing. In a nutshell, there are few situations (design options, software processes, software components) whose similarity (proximity) has been carefully investigated and quantified numerically. For instance, we may state that two classes are similar with a similarity degree equal to 0.7 (on the unit scale). These cases give rise to a labeled data set. It is used afterwards to train a neural network and in this way quantify implicitly the level of similarity. Once trained, the neural network determines a degree of similarity between two software artifacts treated as its inputs. The similarity measure is used to guide unsupervised learning. One should stress that the notion of similarity is crucial to all techniques of unsupervised learning as being driven by the notion of resemblance of data points. The data set in unsupervised learning is usually much bigger than the one used for supervised learning. Put it in a certain perspective: supervised learning in the form discussed here provides us with some knowledge (quantification of similarity) while unsupervised learning relies on data only and attempts to reveal some structure there [7]. The overall scheme of data – knowledge hybridization is portrayed as shown in Figure 1.

3. The neural architecture: supervised and unsupervised learning

As outlined in the previous section, the problem calls for a hybrid of neural architecture that is capable of addressing the need for supervised as well as unsupervised learning. At the end of supervised learning, we exploit standard feedforward neural networks. The unsupervised end of learning is completed in some standard way such as self-organizing maps. Let us briefly discuss the details of these two modes of learning as being supported by specific neural architectures.

The supervised learning is concerned with an optimization of the similarity level between pairs of input data. The learning data set comprises of triples \((x, x', d)\) where \(d\) is a given level of similarity between \(x\) and \(x'\) as seen in Figure 2(a). A feedforward neural network (usually we confine ourselves to a single hidden layer structure) is a standard way to develop such a mapping using the backpropagation algorithm as a learning scheme. The unsupervised learning dwells on the results of supervised learning in the sense that the similarity measure determined there is used to guide the distribution of the unlabelled data. To elaborate on this in depth, refer to Figure 2(b). In this case, we exploit the standard self-organizing map (SOM) that has been already exploited in Software Engineering [6]. The SOM [3][4] uses a mechanism of competitive learning: a multidimensional data point is mapped onto the two-dimensional map at the location whose neuron exhibits the highest strength of firing that is a minimum distance between the data point and the connections of the neuron. In our case, the distance is generalized to the concept of similarity. The computations of the similarity involves \(x\) and \(w(i,j)\) with the latter being the connections of the neuron at the \((i,j)\)-th coordinates of the map.

Figure 1. A two-phase process of revealing structure in data: neural network constructs a similarity measure over selected data subset (referential supervised learning) that is followed by unsupervised learning (realized e.g., through a self-organizing map)

Figure 2. The paradigm of supervised learning - similarity determination (a) and hybrid supervised -unsupervised learning: SOM augmented by the neural similarity model (NN) (b)
4. Experimental studies

This section shows results of experimental studies dealing with synthetic two-dimensional data and a multivariable data set of software modules. We present a phase of supervised learning that is followed by the hybrid version of the SOM learning.

4.1. The model of supervised learning

Example 1. A synthetic two-dimensional data set exclusive – or (XOR) [9] multivalued function.

The following is an illustrative example highlighting the need for the use of the neural architecture to quantify a concept of similarity manifesting in the problem. Consider a five-dimensional exclusive-or multivalued problem where the data adhere to the following fuzzy function

\[ F: [0,1]^5 \rightarrow [0,1] \text{ such that } F(x) = F(x_1, x_2, \ldots, x_5) = x_1 \oplus x_2 \oplus \ldots \oplus x_5 \]

The function is defined recursively that is we start from two variables

\[ x_1 \oplus x_2 = (\overline{x}_1 t x_2) s(x_1 t \overline{x}_2) \text{ with } \overline{x} = 1 - x \]

Here \( x \) is in the unit interval. The logic operators (t- and s-norms) are realized in the form of the product and probabilistic sum, cf.[8]. Let us recall that by a t-norm we mean a two-argument function \( t: [0,1]^2 \rightarrow [0,1] \) such that (i) monotonically increasing, (ii) associative, (iii) commutative and (iv) satisfying two boundary conditions \( at0 = 0 \text{ and } at1 = a \) where \( a \) is in the unit interval. An s-norm satisfies the same conditions as for t-norms with an exception of the boundary conditions that now read as \( as0 = a \text{ and } as1 = 1 \). In particular we have: \( ab = ab \) (product) and \( asb = a + b - ab \) (probabilistic sum). T- and s-norms are generalizations of the standard logic (and and or) operations encountered in two-valued logic. Interestingly, when considering the distance \( |F(x) - F(x')| \) vis-à-vis the Euclidean distance in the input space (viz. the space of the input variables \( x, d(x, x') \)), the two-dimensional plot of distances reveals a lot of scattering, Figure 3. For the same distance in the input space, we encounter a variety of the corresponding distances in the output space. This points out that the Euclidean distance does not seem to be a suitable instrument to express similarity in the input space. Clearly, it does not coincide with the anticipated similarity of the patterns (data) themselves.

Then the similarity measure was quantified by training a neural network. The network has a single hidden layer with 15 nodes (neurons). Each neuron comes equipped with a sigmoidal nonlinear function. The learning was carried out using a standard backpropagation optimization algorithm with a learning rate \( (\alpha) \) being equal to 1. The plot of the performance index \( Q \) is shown in Figure 4. It becomes evident that the learning was smooth with a rapid reduction in the values of \( Q \). The performance index itself is viewed as a sum of squared errors between the target values and the output values produced by the network, that is

\[ Q = \sum_{k=1}^{N} (targ et(k) - NN(x(k)))^2 \]

where \( \{x(k), target(k)\}, k=1, 2, \ldots, N \) are the input – output data (training set) while \( NN(x(k)) \) denotes an output of the neural network for the given input \( x(k) \).

The scatterplot of the same data as used before is shown in Figure 5. It becomes obvious that the similarity expressed by the neural network is consistent with the similarity in the output space. This points out that the constructed neural model of similarity is more adequate than the previous one.

Example 2. Now we use a well-known MIS data set [5] to illustrate fundamental concepts and discuss all computational facets of the design as well as reveal interesting association patterns in the data set itself. This data set consists of a number of commonly used software measures (such as lines of code, McCabe complexity, bandwidth, etc.) describing a collection of software modules, Table 1. The data set includes also a number of changes made to the individual module during its testing phase.
We pick up 50 data points for supervised learning of the similarity measure. Two neural networks were used: the first has 15 nodes in the hidden layer; in the second version this size was increased to 30 neurons. The learning process is shown in Figure 6. Most of the improvement (reduction on the values of \( Q \)) is noticeable at the beginning of the learning. The increase in the size of the hidden layer does not impact the final results quite significantly. The correspondence between the similarity measure in the input space and the output is illustrated in Figure 7(a) and 7(b).

### Table 1. Software measures

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOC (length of code with comments)</td>
</tr>
<tr>
<td>2</td>
<td>CL (length of code without comments)</td>
</tr>
<tr>
<td>3</td>
<td>Tchar (number of characters)</td>
</tr>
<tr>
<td>4</td>
<td>Tcomn (number of comments)</td>
</tr>
<tr>
<td>5</td>
<td>Mchar (number of comment characters)</td>
</tr>
<tr>
<td>6</td>
<td>Dchar (number of code characters)</td>
</tr>
<tr>
<td>7</td>
<td>( N ) (program length)</td>
</tr>
<tr>
<td>8</td>
<td>( \hat{N} ) (estimated program length)</td>
</tr>
<tr>
<td>9</td>
<td>( N_r ) (Jensen’s program length)</td>
</tr>
<tr>
<td>10</td>
<td>( V(G) ) (McCabe’s cyclomatic number)</td>
</tr>
<tr>
<td>11</td>
<td>BW (Belady’s bandwidth metric)</td>
</tr>
<tr>
<td></td>
<td>output variable</td>
</tr>
<tr>
<td></td>
<td>NC (number of changes)</td>
</tr>
</tbody>
</table>

**Figure 4.** Performance index \( Q \) in successive learning epochs

**Figure 5.** Scatterplot of similarity measures in the input and output space; note the use of the neural network in the determination of the similarity level at the input space (as before 40 data points resulted in 1600 pairs used for training purposes)

**Figure 6.** Performance index in successive learning epochs for 15 nodes in the hidden layer (a) and 30 nodes in the hidden layer (b)
In both case, we encounter a general relationship between the similarities yet these are not easily predictable for any specific similarity on the input space. The “cloud” of points exhibits some scattering, in spite of the general tendency that is quite strong. More scattering is also visible for lower values of the similarity in the input space. For comparison, the same relationship is visualized for the Euclidean distance, Figure 8 where there is far more dispersion than for the neural model of similarity.

4.2. The combined (hybrid) model of supervised and unsupervised learning – data visualization via self-organizing maps

The supervised learning is the first phase of the overall process of data analysis. Next, we use the mechanisms of unsupervised learning governed by the neural similarity measure. The unsupervised learning is completed with the aid of the self-organizing map. The main features of this network are as follows:
- size of the maps: 6 x 6;
- initial condition: randomly initiated connections;
- frequency sensitive learning: no;
- type of neighborhood: Gaussian Function;
- termination criterion: 1000 iterations.

The results are summarized in a series of SOMs; in all of them it becomes evident that the neural model of similarity exhibited a significant (and advantageous) impact on the results of organization of the data. For the XOR problem it resulted in significant condensation of data having the same output while the Euclidean distance gave rise to an excessive spread of the data across the entire map, see Figure 9. The neural similarity measure forced SOM to clearly discriminate between various data points. These that are close in terms of the similarity are immediately placed together while those evidently distinct were pushed away, Figure 10.
Figure 9. Visualization of XOR (exclusive-Or) data in the SOM using Euclidean distance. This map illustrates how data are clustered. The notation used at each node (say, 2 0.45-0.46) identifies the number of data points (here 2) allocated to the location of the map (neuron) and a range of values assumed by the output for these data points (here 0.45-0.46).

Figure 10. Visualization of data in the SOM constructed with the use of the neural network similarity measure. The notation in the SOM is the same as used in the previous figure.

The same effect of condensation is present for the MIS data. The use of the Euclidean distance yielded a substantial level of scattering, Figure 11. This undesired phenomenon has been reduced when applying the neural similarity measure, Figure 12.

5. Conclusions

In this study, we have introduced an idea of a neural network as a model of the similarity measure between data. This model, being more general than standard distances is in rapport with the essence of expressing similarity in Software Engineering. It exhibits two interesting features. First, it comes with a high level of flexibility that can be exercised through learning of the ensuing neural network. Second, it easily incorporates as a front-end of neural models of data visualization such as self-organizing maps. Thus the SOMs provide the designers with a highly user-friendly support vehicle that is particularly helpful in gaining a comprehensive insight into the data.

Acknowledgments

The support from the Natural Sciences and Engineering Research Council of Canada (NSERC) and ASERC (Alberta Software Engineering Research Consortium) is gratefully acknowledged.
Figure 12. Visualization of data in the SOM constructed with the use of the neural similarity measure: 15 nodes in the hidden layer (a) and 30 nodes in the hidden layer (b)

6. References


Recovering UML Diagrams from Java Code using Patterns

Jörg Niere  
Department of Mathematics and Computer Science  
University of Paderborn  
Warburgerstraße 100  
33098 Paderborn, Germany  
nierej@uni-paderborn.de

Jörg P. Wadsack  
Department of Mathematics and Computer Science  
University of Paderborn  
Warburgerstraße 100  
33098 Paderborn, Germany  
maroc@uni-paderborn.de

Albert Zündorf  
Department of Mathematics and Computer Science  
University of Paderborn  
Warburgerstraße 100  
33098 Paderborn, Germany  
zuendorf@uni-paderborn.de

ABSTRACT
Recovering the static structure of legacy source code e.g. as an UML class diagram is quite well understood. In contrast, recovering high-level behaviour diagrams from source code is still an open issue. This paper proposes to use fuzzy pattern recognition techniques for the recovery of UML collaboration diagrams from source code. The approach is based on a knowledge base of basic datatypes and of generic collection classes and of code clichés for Java beans and of fuzzy patterns for object structure look-up and modification clichés. We handle the diversity of existing code clichés by organizing them in an object-oriented hierarchy factorizing important common properties and by relaxing exactness requirements for cliché detection with the help of fuzzy theory. We handle the runtime efforts for cliché detection using a sophisticated inference mechanism based on generic fuzzy reasoning nets (GFRN's). The work is part of the FUJABA case tool aiming to support round-trip engineering for UML and Java.

Keywords
UML, Java, fuzzy logic, reverse-engineering, pattern recognition, round-trip engineering

1 INTRODUCTION
Reverse engineering aims to provide program descriptions on higher levels of abstractions. Such an abstract level could e.g. be a program description using UML diagrams. These program descriptions facilitate the understanding of program structures and program behaviour. State-of-the-art CASE tools like Rational Rose [Ros], TogetherJ [Tog], and Rhapsody [Rha] provide only recovery functions for class diagrams using markers in the code. Notably, Rhapsody supports the recovery of state-charts. The recovery of high-level behaviour descriptions for legacy object-oriented programs is still an open issue.

The work described in this paper is part of the Fujaba project. The Fujaba project aims to develop a round-trip engineering CASE tool for UML. In our previous work, [FNTZ98, JZ98, NNSZ99, KNNZ00], we proposed an execution semantics for UML statecharts, activity diagrams, and collaboration diagrams. Our execution semantics allows to use these UML behaviour diagrams as a visual programming language for object-oriented applications. The code generators of the Fujaba environment translate such executable specifications into fully functional Java classes including method bodies.

Theoretically, with Fujaba no manual coding is necessary any more. Practically, the generated code is frequently modified during debugging. The code may be merged with the contributions of other developers e.g. via a configuration management system. Some system parts may be added by other code generators, e.g. a GUI builder or a database middleware layer, or a distribution layer like CORBA [Vin97].

To deal with such code modifications, the Fujaba environment provides reverse engineering support that analyses Java source code and tries to create the corresponding UML class and behaviour diagrams, cf. [NNWZ00]. So far, the reverse engineering capabilities of Fujaba are limited to round-trip engineering support. This means, Fujaba is merely able to reverse engineer code it has generated itself or that it has been written as if it would have been generated.

Reverse engineering of arbitrary legacy code and third party code is a challenging problem. Most CASE tools are restricted to the analysis of static program structures, i.e. reverse engineering of class diagrams. But even for class diagrams the correct recovery of (structural) associations between classes is not trivial.

To overcome these limitations and to be able to deal with legacy code constructs, this paper proposes the use of fuzzy reasoning technologies, i.e. generic fuzzy reasoning nets, cf. [Jah99]. In this paper we focus on the analysis of code that deals with object structure modifications. We start with a knowledge base about the semantics of pre-defined container classes and their access operations. We analyse attribute declarations to identify (sets of) basic references hold by certain classes. Methods modifying these basic references are classified as access methods with certain degrees of confidence. If access methods are detected with sufficient confidence, their use can be analysed. Specific sequences of access method usages may be turned into collaboration diagrams that describe the look-up of certain object patterns and the modifications of such patterns and the collaboration messages send between the participating objects. The
combination of activity diagrams specifying the control flow and embedded collaboration diagrams in a certain activity is called story-diagrams. Those story-diagrams serve as the behavioural specification for a software system and are linked to method declarations in class diagrams.

The following Section 2 introduces a track based material transportation system as running example. In Section 3 the reconstruction of class diagrams and story-diagrams is described using annotations. The specification of code clichés is introduced in Section 4 and the following Section 5 introduces the corresponding execution formalism, namely generic fuzzy reasoning nets. Section 6 and Section 7 discuss related work and present future work.

2 RUNNING EXAMPLE: SWITCH CONTROL SOFTWARE

In this section, we introduce the switch control software of a track based material transportation system as running example. This example stems from the joint research project ISILEIT funded by the german research foundation (DFG). Within ISILEIT we collaborate with our mechanical and electrical engineering department to set-up an agent based production control system. The building blocks of such a production control system are different, self-acting and computer controlled resources like e.g. switches, shuttles, machines, or robots. Shuttles move on rails and transport goods between various production places. Each production place can be reached using switches in the railway system. Shuttles announce themselves at the switches if they want to visit the corresponding production place. The switch control software keeps track of the targets of the different shuttles and operates the switch accordingly.

Figure 1. shows the structure of a switch as part of a production control system, which we specified by employing Fujaba. The switch has an actor, i.e. the switch drive, which changes its direction. Further it has some sensors, which observe the environment and a Local Operating Network-node, which is connected to a communication network. In our example, the identification unit detects an arriving shuttle and reports the shuttle’s id to the switch control node. Now, the control software decides in which direction the shuttle should be send. If the switch has to change its direction, it activates the stopper in order to let shuttles wait. One has to assure, that no shuttle is in the switching area, when the switch drive is activated, because otherwise the switch drive could be damaged. For that reason, the switch has a pass observer at each exit, which reports every shuttle leaving the area. Note, that we have a one-way driving direction, so that we have one entry and two exits, which means that our example shows a "branching switch".

3 CLASS- AND STORY-DIAGRAM RECONSTRUCTION

Recovering class and behaviour diagrams from Java code is divided into two tasks. First, the static information, the class diagrams, will be reconstructed and in a second task, the behaviour diagrams (here story-diagrams) will be recognized.

![Figure 1. Switch structure](image1.png)

![Figure 2. Rudimentary class diagram recovery](image2.png)

Figure 2. shows a cut-out of the static elements of the Java code of class Switch. From this code fragments a rudimentary class diagram recovery approach could reconstruct the class diagram shown below the Java code. Classes become classes. Attributes of basic types like int or boolean become class attributes. Method declarations become methods of the corresponding UML classes. Inheritance in Java is directly mapped to inheritance relations in the diagram.

We assume, that the class diagram recovery mechanism has already knowledge about all basic Java types. Thus, it may identify types Stopper and OrderedSet as user defined types. Accordingly, the corresponding Java attributes are interpreted as references in the class diagram.
OrderedSet is a pre-defined generic container class from the Java Foundation Class (JFC) library. Equipping our reconstruction mechanism with this additional knowledge, it could turn the announced reference from Switch to OrderedSet into a to-many reference to class Object (the basic class of all classes in Java). In Java, we face the problem that generic container classes do not provide information about the types of the contained entities. To derive such information, our recovery mechanism needs to know the semantics of the access methods of container classes, e.g. method add inserts elements into the container. This allows us to derive the entry type for containers from the usages of the corresponding add method.

Recognizing classes and class members is fairly simple. Analysing method bodies is a more challenging task. One approach to recover the semantics of method bodies is the detection of so-called code clichés, cf. [Wil94]. In Java, very common code clichés are bean properties. A bean property is an attribute with appropriate read and write access methods. A bean property must have the same name as the corresponding attribute plus a `get` (`set`) prefix. In addition, the write method must have exactly one parameter with the same type as the corresponding attribute. Finally, within the body of the write method the parameter value must be assigned to the corresponding attribute. Once an attribute and its access methods have been classified as a bean property, the class diagram recovery mechanism may simplify the corresponding class, accordingly.

In our approach, we use similar code clichés to implement bi-directional associations. Bi-directional associations are implemented using pairs of pointers. These pointers are encapsulated with appropriate read and write access methods. The write access methods guarantee the consistency of the pointer pairs by calling each other, mutually. For to-one associations simple attributes and set- and get-methods are used. For to-many associations we employ generic container classes and methods for iterating through the set of neighbours, adding neighbours, and removing neighbours.

Fujaba employs a flexible cliché detection mechanism the so-called annotation engines. The annotation engines enrich the abstract syntax tree of a parsed program with so-called annotations [HN90]. Annotations are markers for detected occurrences of code clichés. In Figure 3 such annotations are shown as ovals. Annotations enrich the semantics information of abstract syntax trees and allow e.g. to simplify class diagrams. Moreover, such annotations assign a certain semantics to certain methods or code fragments. This semantics may be used for further analysis of other method bodies.

Consider for example Figure 4. Line 35 employs method getIdUnit. Let us assume that method getIdUnit has been annotated as the read access method for an association between class Switch and class IdentificationUnit. This allows us to interpret line 35 as a link look-up operation. In a collaboration diagram such a link look-up operation is shown as a line labelled with the corresponding association name. Such a line connects two boxes representing the source and target variable. In our example these are the variables this and idU, respectively, cf. Figure 4. Similarly, the knowledge about access methods

```java
12: public class Shuttle
13: ...
14: private int shuttleId;
15: ...
16: public int getShuttleId () {
17:   return this.shuttleId;
18: }
19: ...
20: public void setShuttleId (int id) {
21:   this.shuttleId = id;
22: }
23: ...
24: } // class Shuttle
```

**Figure 3. Dealing with bean properties**

However, class OrderedSet is a pre-defined generic container class from the Java Foundation Class (JFC) library. Equipping our reconstruction mechanism with this additional knowledge, it could turn the announced reference from Switch to OrderedSet into a to-many reference to class Object (the basic class of all classes in Java). In Java, we face the problem that generic container classes do not provide information about the types of the contained entities. To derive such information, our recovery mechanism needs to know the semantics of the access methods of container classes, e.g. method add inserts elements into the container. This allows us to derive the entry type for containers from the usages of the corresponding add method.

Recognizing classes and class members is fairly simple. Analysing method bodies is a more challenging task. One approach to recover the semantics of method bodies is the detection of so-called code clichés, cf. [Wil94]. In Java, very common code clichés are bean properties. A bean property is an attribute with appropriate read and write access methods. A bean property must have the same name as the corresponding attribute plus a `get` (`set`) prefix. In addition, the write method must have exactly one parameter with the same type as the corresponding attribute. Finally, within the body of the write method the parameter value must be assigned to the corresponding attribute. Once an attribute and its access methods have been classified as a bean property, the class diagram recovery mechanism may simplify the corresponding class, accordingly.

In our approach, we use similar code clichés to implement bi-directional associations. Bi-directional associations are implemented using pairs of pointers. These pointers are encapsulated with appropriate read and write access methods. The write access methods guarantee the consistency of the pointer pairs by calling each other, mutually. For to-one associations simple attributes and set- and get-methods are used. For to-many associations we employ generic container classes and methods for iterating through the set of neighbours, adding neighbours, and removing neighbours.

Fujaba employs a flexible cliché detection mechanism the so-called annotation engines. The annotation engines enrich the abstract syntax tree of a parsed program with so-called annotations [HN90]. Annotations are markers for detected occurrences of code clichés. In Figure 3 such annotations are shown as ovals. Annotations enrich the semantics information of abstract syntax trees and allow e.g. to simplify class diagrams. Moreover, such annotations assign a certain semantics to certain methods or code fragments. This semantics may be used for further analysis of other method bodies.

Consider for example Figure 4. Line 35 employs method getIdUnit. Let us assume that method getIdUnit has been annotated as the read access method for an association between class Switch and class IdentificationUnit. This allows us to interpret line 35 as a link look-up operation. In a collaboration diagram such a link look-up operation is shown as a line labelled with the corresponding association name. Such a line connects two boxes representing the source and target variable. In our example these are the variables this and idU, respectively, cf. Figure 4. Similarly, the knowledge about access methods
may allow to interpret line 39 and 40 as look-up of a qualified association with cardinality 0..1. Lines 42 to 44 show a typical cliché for the look-up of a to-man y association. In a collaboration diagram such look-ups are shown as lines between appropriate objects, cf. the bottom of Figure 4. Finally, our annotation knowledge allows us to infer, that line 49 creates an at link between object s and idU and line 51 creates a wantsTo link from s to t1. In the collaboration diagram we show link creation using grey colour and the «create» stereotype.

Thus, the detection of Java bean property clichés allows us to assign a dedicated semantics to read- and write-access methods. This knowledge allows us to analyse the usage of such access methods and to recover collaboration diagrams from such code. Note, in more complex situations, a method body may contain several code fragments that correspond to collaboration diagrams. Such code fragments may be mixed with other code (Figure 4 line 37) and control structures that are not covered by such an analysis. To deal with such situations, we embed detected collaboration diagrams into activity diagrams. The activity diagram part shows the top-level control flow and text activities for unrecognized code. Recognized code is turned into collaboration diagram activities.

So far, our annotation engines are able to deal with code that strictly conforms to the Fujaba code generation concepts. Due to our experience, legacy and third party code frequently contains similar code fragments. However, the classification of access methods and especially the detection of collaboration diagram operations is very challenging. For example, there are numerous ways to implement a to-man y association. Accordingly, there are very different ways to enumerate all neighbours of a given object. Similarly, there are various coding clichés for test operations and for dealing with test results. The next section describes how clichés can be specified using the Fujaba environment.

Figure 4. Recognizing object structure changes as collaboration diagrams
4 ANNOTATION-ENGINE SPECIFICATION

In general, one annotation engine is responsible for the
detection of one cliché. The annotation engines work on an
abstract syntax graph constructed out of the code (As parser
we use JavaCC [JCC]). Each annotation engine is embedded
in an architecture which is mainly a combination of a
strategy and a chain of responsibility pattern [GHJV95].
Changed objects of the syntax graph are passed to a
broadcaster which distributes the objects to those engines
registered for the objects’ type. The main part to specify a
code cliché relies on a method implementation for the
corresponding annotation engine. The annotation methods
are illustrated using activity diagrams and collaboration
diagrams i.e. Fujaba story diagrams. Figure 5. shows
collaboration diagrams for the detection of link annotations.
The upper part of Figure 5. shows an activity diagram for the
annotation of a readToOneLink (cf. Figure 4 line 35). If an
assignment in a method body consists of a variable and a
method call, which is annotated as read access of a 'to-one'
association, then we annotate the variable and the method
with an ReadToOneLink node. The annotation of a
readQualifiedLink (cf. Figure 4 line 39) is similar to the
readToOneLink annotation. Only node as:ToOneAssoc in the
collaboration diagram on the top of Figure 5. has to be
replaced by as:QualifiedAssoc. Also the created
annotation rl:ReadToOneLink has to be replaced by a
variable of type ReadQualifiedLink. For the annotation of the
more complex iterateToManyLink story-diagram we have to
check the read access over an iterator construct, cf. bottom of
Figure 5. Basically, a first assignment has to retrieve an
iterator and a second assignment must use this iterator to
assign a value to the actual target variable. Therefore, the
lower part of the annotate_IterateToManyLink collaboration
diagram refers to line 44 and the upper part matches line 42.

To get an overview of the detectable code clichés, we
developed a domain model for the annotations of code
patterns. Figure 6. shows a cut out of the domain model (as
an UML class diagram) used for the round-trip facilities
supported by the Fujaba environment. This model shows the
annotations for the class diagram on the left and the story
diagram annotation parts on the right.

Using inheritance hierarchies allow some simplification in
the specification of the annotation engines. Without the
inheritance hierarchy collecting all look-up links of a method
and aggregate them with a LookupPart annotation, may
result in three annotation methods. I.e. three annotation
engines, one collecting ReadToOneLinks, another
ReadQualifiedLinks and an engine collecting
ReadToManyLinks.
Instead we can use one method (engine) using the superclass LookupLink of the three read link classes. This reduces the number of required methods. Figure 7. shows the annotation for LookupPart containing a set of LookupLink nodes (depicted by two stacked boxes), e.g. ReadToOneLink, ReadQualifiedLink and IterateToManyLink (cf. Figure 4). In general in a collaboration diagram must not contain additional lines like debug outputs (cf. Figure 4 line 37). Such negative application conditions may be specified by negative (crossed-out) nodes. Therefore, the crossed-out node co:ConsoleOutput in Figure 7. assures that a lookup part is only annotated if there is no output printed on the console. For patterns on a high level of abstraction the specification using pure collaboration diagrams is sufficient enough. In general, every time a boolean answer is expected and easily given, collaboration diagrams are sufficient. Looking for clichés (code pattern) is a little bit more difficult, because there are many more variants (e.g. syntax) to express the same semantics. For example, some developers prefer while-loops and other for-loops to solve the same problem.

Such variants highly depend on the education and social background or on affectations of a specific developer. Using collaboration diagrams to specify clichés lead more or less to one rule for one cliché. To deal with this problem, we relax the exactness of cliché detection using fuzzy logic. This allows us to downsize the cliché detection of certain indicators that signal the existence or absence of certain clichés with a certain confidence. For example in Figure 4 the three variants (line 35, line 39 and line 44) of binding an object to a variable annotated with a subclass of LookupLink results in three completely different specifications if we are using collaboration diagrams, only (cf. Figure 5).
This results from the fact, that the abstract syntax trees for the three assignments is different. However, all three rules employ the same indicator, the usage of an association lookup method and an assignment. Thus, we may use only a single rule detecting the usage of a read access method within an assignment, cf. Figure 8.

Also the three rules have many structural similarities, which makes them gluing candidates. ReadToOneLink and ReadQualifiedLink only differ in the type of the variable as. ReadToManyLink seems not to be so close to the other two, e.g. there are more variables used. But, taking the inheritance hierarchy into account and put the common parts into one collaboration diagram forces uncertainty in the detection. To handle the uncertainty coming up with the abstraction of the three collaboration diagrams, we indicate the annotation with a confidence (fuzzy-value) of 80. Collaboration diagrams added with fuzzy-values, are called fuzzy patterns. Figure 8. shows the resulting fuzzy pattern for general lookup-links, which is a combination of the three variants ReadToOneLink, ReadQualifiedLink and ReadToManyLink before. The fuzzy value of the pattern is put in a circle at the top right corner of the variable, here the new annotation. A fuzzy pattern fires if its premise is fulfilled which means in this case, that all variables specified in the pattern must be bound to an object of the abstract syntax graph with a fuzzy value that is higher than the required value. The result value for created annotations is the fuzzy-and of the fuzzy beliefs of the found objects and the fuzzy value of the created object.

If the pattern could be matched, the resulting annotation which is created, gets as fuzzy value positive 80. This reflects that due to our experience such a match is in 80 percent of all cases a correct match and the underlying source code refers to a binding object situation. Having a deeper look in the source code, we find that the fuzzy pattern matches for line 35, line 39, and line 42 instead of line 44. Line 42 is not exactly the line where the object is bound, this is done in line 44, but such uncertainty could be expressed through fuzzy-values and enables us to reduce the number of patterns in the reengineering process.

In those cases, where it is necessary to decide whether or not the automatic decision is correct, the reengineer has to be asked. Typically, the reengineer has to look for nodes with a fuzzy value minor than a defined limit and could overwrite the value to zero or 100. The changes trigger a re-evaluation of the fuzzy values of the depending nodes. Experiences have shown that such a semi-automatic approach causes better results than fully automatic approaches, because a user or reengineer may interact with the tool and contribute her/his knowledge.

5 DETECTION MECHANISM

For detecting clichés in source code we use the formalism of Generic Fuzzy Reasoning Nets (GFRN) [JSZ97, Jah99]. The GFRN formalism has initially been applied in the domain of data reverse engineering. It facilitates the specification and execution of analysis rules and processes and incorporates a notion of uncertainty. In principle, a GFRN is a net of predicates (oval shape) and implications (represented as boxes) which are connected by arcs (cf. Figure 9). Arcs are labelled with formal parameters that can be used to specify constraints for implications. Negation in implications are represented by arcs with black arrow heads. Each implication has an associated confidence value (CV). Based on the theory of possibilistic logic [DLP94], the semantics of a CV is a lower bound of the necessity that the corresponding implication is valid. Note, the CV associated to the implications specify a measure for the degree of certainty. According to typical fuzzy inference operators, an overall valuation for each cliché is based on the difference between the maximum positive CV and the maximum negative CV.

Each specified fuzzy pattern, is canonically mapped to a (part) GFRN. Afterwards, the (part) GFRN’s are merged, i.e. predicates with equal names are glued, resulting in one or more larger nets. Figure 9 shows the GFRN after merging the canonically mapped fuzzy pattern shown in Figure 8 and a fuzzy pattern constructed out of Figure 7.. Variables that occur in the fuzzy pattern are mapped to predicates and are rendered in grey colour while the annotation which represents the goal (created annotations) of the analysis is mapped to a so-called dependent predicate LookupLink.
respectively LookupPart, rendered in black. The relations between variables, i.e. links between variables are mapped to constraints within the corresponding implication (π1(Δ) = ψ). Therefore, the data represented by the variable in a fuzzy pattern is mapped to formal parameters in the GFRN. For example, variable \( a:Assignment \) is mapped to an Assignment predicate and the data, i.e. the variable, the method, the expression and also the assignment itself, is mapped to the formal set parameter \( a \) at the transition from predicate Assignment to implication \( 1 \).

For efficient fuzzy pattern analysis, we propose a two-step process. First, the fragment graph is searched for a match for all positive variables. Subsequently, our detection strategy aims to extend each such match by a match for negative (cancelled) nodes in a fuzzy pattern. The GFRN formalism facilitates the specification of such a strategy by distinguishing between so-called data-driven and goal-driven predicates. Matches for data-driven predicates (represented with solid grey outline) are searched at the beginning of the analysis process (cf. MethodCall, or and Variable in Figure 9). Subsequently, the fragment graph is searched for matching instances of goal-driven predicates (with dashed grey outline). For example, the ConsoleOutput predicate is mapped to a goal-driven predicate and allows us to specify that a console output results in a negative belief of 30 if it occurs within some lookup links of a method. This is different to Figure 7., where a console output must not appear between two lookup-links of a method. This can also be expressed in a corresponding fuzzy pattern (not shown here) and helps us to reduce the number of fuzzy patterns for code clichés. We refer to [JH98] for details on the GFRN inference engine.

6 RELATED WORK
The FUJABA environment and especially its code generators are described in detail in [FNTZ98, KNNZ00]. The underlying technique called story-driven-modelling starting from the first phases of a software development process in general is described in [JZ98] and specializations for production control systems are presented in [NNSZ99].

[HN90] proposes a program analysis based on an Event Base and a Plan Base. First, rudimentary events are constructed from source code. Plans are used to consume one or more events and fire a new event which correspond to the plan’s intention. Annotations visualize the event flow and plan definition. In [PP94] a matching algorithm for syntactic patterns based on a non-deterministic finite automaton is presented. The non-determinism is used to provide dummy variables for special pattern symbols representing syntactical information like variables or function calls. Both [HN90] and [PP94] need one definition for one implementation variant, which lets the approaches fail for at least legacy systems with unknown code-styles.

An automatic approach to extract semantics information from source code is presented by Wills [Wil94]. Similar to our collaboration diagrams, Wills uses graph rewrite rules to specify implementation patterns in terms of so-called program plans. Program plans are structured in a hierarchical design library, i.e., abstract plans consist of aggregations of several more simple plans. A specific library stores program-plans for different domains, e.g., Wills presents a library for sorting algorithms. Wills follows a bottom-up strategy to detect plans which limits the practical usage of the approach to source code about 1000 lines. Quilici [Qui94] proposes an indexing technique and combines top-down and bottom-up detection to overcome this problem.

In [KSRP99], Keller et al. present a semi-automatic approach to find design patterns [GHJV95] in source code. Patterns are represented in UML notation [BRJ99], namely in CDIF format. Matching algorithms are not automatically generated but implemented by hand. [Rad99] employs graph transformations (graph rewrite rules) to extract design patterns automatically and to refactor parts of the source code. Graph rewrite rules are similar to our collaboration diagrams without fuzzyness, so the approach is restricted to the restrictions discussed in section 4.

7 CONCLUSIONS AND FUTURE WORK
This paper introduces the round-trip facilities of the Fujaba environment based on a material flow system specification. The control software for the material flow system is generated out of a Fujaba specification, adapted for speed or debugging reasons and has to be reverse engineered for major modifications. Modifications in the source code should effect in the (re)constructed diagrams. Especially the paper presents a technique using annotations of the syntax graph to gain semantics knowledge. For that reasons, first static information in form of UML class-diagrams is reengineered and afterwards behavioural diagrams namely story-diagrams are recovered. We introduce a solution to inspect Java source code by defining fuzzy patterns. Fuzzy patterns are defined using collaboration diagrams. The huge number of implementation variants for code clichés results in a fuzzy pattern definition for each variant. To catch many variants in one fuzzy pattern definition we introduce fuzzyness into the definition. This fuzzyness let us deal with uncertainty coming from the generalisation of several similar code clichés. As detection mechanism for fuzzy patterns, we introduced Generic Fuzzy Reasoning Nets and presented a mapping of fuzzy pattern to GFRN’s.

We are currently working on a more handy syntax and a first implementation of our approach. Therefore, we enhance the Fujaba system, which already supports the instantiation of design patterns and a rudimentary mechanism to extract design patterns out of Java source code. We plan to enhance our approach by inheritance and polymorphism like in object-oriented languages. For example, a fuzzy pattern detecting aggregations between classes could inherit from a general association detecting fuzzy pattern. Because typically, the difference between them is an explicit deletion of the aggregated objects when using aggregations.

The current prototype of the Fujaba environment is available as free software and comprises about 330 000 lines of pure Java code. Additional information and the current release version of the Fujaba system can be downloaded via:

http://www.fujaba.de
REFERENCES


Interactive Graph Clustering based on User Hints

Hugo A. D. do Nascimento\textsuperscript{1}  
Depart. of Computer Science and S.E.  
University of Newcastle - Australia  
hadn@cs.newcastle.edu.au

Peter Eades\textsuperscript{2}  
Depart. of Computer Science.  
University of Sydney - Australia  
peter@cs.usyd.edu.au

Abstract

Graph clustering is a difficult optimization problem that arises in Software Engineering. This paper presents a framework for graph clustering where users play a strong role. In the framework, a soft computing method produces a clustering of the graph and a visualization of it is provided using some graph drawing techniques. Through the visualization the user can then analyze the clustering and give “hints” that help the soft computing method to find better solutions. Hints include a variety of constraints for solutions, as well as direct manipulation of the previously computed clustering. The framework is flexible: it can accommodate several kinds of hints, clustering algorithms, and visualization techniques.

Keywords: clustering, user interaction, graph drawing, hints.

1. Introduction

The role played by users in computationally intensive processes can be far more significant than as a mere observer. Users always need to provide a representation of a problem to be solved (the input), and solution mechanisms (programs); in this paper we discuss ways in which the user can evaluate and control the results produced (the outputs). Such interaction between users and computers can be intensive or sporadic, through a graphical user interface, or hard-coded in command lines; but it undoubtedly represents an important aspect of modern software, and it is especially important for problems that require intensive computation. In the present paper, we introduce a framework for user interaction with methods for solving difficult problems. In our framework, users play a strong role. We have implemented our framework as a system for graph clustering using soft-computing methods, some graph drawing methods, and a graphical interface for interaction. Basically, the user drives the clustering process by defining new pieces of information (what we call hints) to be considered in the model, thus helping the soft computing method to find good solutions.

2. Graph clustering

Graph clustering, also known as graph partitioning, aims to divide the set of vertices of a graph\textsuperscript{3} into disjoint subsets (clusters or partitions) while keeping minimal the amount of edges linking vertices of distinct sets, as well as satisfying some constraints.

2.1 Graph clustering and Software Engineering

In Software Engineering, clustering is used to divide huge programs into blocks with high “cohesion” so that the “coupling” between blocks is minimal. Here the graph may be a data flow graph, a call graph, a control flow graph, or one of the many graphs used in object-oriented design. Many clustering methods have been developed for this purpose; see, for example, [RRHK00, KE00, TH00]. Graphs arising from legacy code are clustered for two main reasons:

- Program comprehension. Human understanding of legacy software is a problem that has become critical in recent years. A legacy system often consists of thousands of interdependent functions, and the human must understand these dependency relations. Clustering reduces the amount of information to be understood, and allows the human to think in terms of higher-level architectural dependencies rather than at the function level.
- Re-modularization. Code that has been maintained, updated, adjusted and ported over a period of several years has a tendency to lose the elegant structure that it once had. Clustering the code units (terms, functions, or files) can suggest new structures for the code.

\textsuperscript{1} Lecturer of the Instituto de Informática, UFG – Brazil. Supported by CAPES-Brazil and UFG.
\textsuperscript{2} Supported by the Australian Research Council.
\textsuperscript{3} A graph $G$ is consists of a set $N$ of vertices (or nodes) and a set $E$ of pairs of vertices, called edges. For the terminology of Graph Theory, see [BM76].
Clustering is also used in other areas of Software Engineering. For example, in distributed and parallel processing [KGGK94], clustering is used to split a large set of tasks into subsets such that each subset can be allocated to a different processor (here tasks are seen as nodes of a graph and the dependencies between tasks are modeled by edges; the aim is to assign tasks to each processor to achieve a balanced load and to minimize the communication between processors).

Moreover, clustering is an element of VLSI design [AK95], where a VLSI circuit needs to be divided into simpler circuits, reducing the complexity of dealing with its whole structure.

In general, clustering is important for all kinds of problems where either the size of the graph is big and has to be reduced, or some inherent structure of the graph can be discovered by looking for coherent parts.

2.2 Clustering quality and constraints

Formally, a clustering of a graph \( G = (N,E) \) (where \( N \) is a set of weighted nodes and \( E \) is a set of weighted edges) is a partition of \( N \) into disjoint subsets (called clusters) \( N_1, N_2, \ldots, N_k \). The quality of the clustering is measured by a function \( Q(N_1, N_2, \ldots, N_k) \) which assigns a positive real number to the clustering.

Two common constraints in clustering problems are to produce a partition with exactly \( k \) subsets, and impose a “balance” on the size of the subsets used. This balance is defined as a limit on the variation between the sums of node weights in each cluster. In this case, the problem is called \( k \)-way partitioning [Jfa98]. When \( k=2 \), we have a special case called bisection. Also, several other kinds of constraints can be specified, e.g. to restrict the total weight of each cluster to a particular range.

2.3 Graph clustering methods

Every imaginable variation of the graph clustering problem is well known to be NP-Hard (even the bisection case [GJS76]) and, therefore, heuristics have to be used for providing fast solutions. These include:

- **Flow methods** can be used to identify small “cuts” in the graph, on which a clustering can be based.
- **Integer linear programming methods**. It is fairly straightforward to encode the quality requirements and constraints of a clustering problem in a linear fashion, and then use methods from integer linear programming [FSS87,GW90].

However, this rich literature on heuristics for graph clustering lacks a general approach that includes the users as an essential element. The object of this paper is to discuss the role that a user can play in such heuristics. A general framework for user interaction with optimization methods is introduced in the next section.

For the moment, we note that the majority of clustering heuristics have an iterative improvement structure, as in Figure 1. This means that at any stage of the method, a feasible solution is available. (Note that in some cases, the solution is “feasible” in a fairly loose sense – for example, it may not satisfy every constraint.)

![Figure 1](image_url)

In this paper we use soft computing methods. This is because the flexibility of soft computing allows easy integration with user hints.

3. A framework based on hints

We aim for a flexible interactive framework where users interact with an optimization algorithm in order to refine the most recent solution obtained. At runtime, users observe a visualization of the current (or, more precisely, recent) feasible solution, and provide feedback to the algorithm. Users play two logical roles:

1. **Insert domain knowledge**: The visualization of the current feasible solution can prompt the user to insert domain knowledge that is not known to the hard-coded algorithm.
2. **Guide the search**: The optimization algorithm may attempt to explore areas that, from the visualization, the user can see are not promising. In this case, the user can guide the algorithm towards more promising areas.

Each of these roles is implemented by user adjustment of either the constraints, the objective function, or by directly manipulating the feasible solution. More specifically, the loop of Figure 1 is augmented as in Figure 2.

In the next subsections we discuss how this approach applies to graph clustering.

### 3.1 Hints for graph clustering

In graph clustering, the logical roles of the user are refined as follows:

1. **Insert domain knowledge**: a visualization of the current clustering may lead the user to believe that, although it satisfies the current constraints and has a good value for the objective function, it is not “right” in the domain context. The user must give the algorithm a hint to move toward a solution that is correct in the domain context. As an example, suppose that the nodes represent modules of a software system, and the edges represent data-sharing relationships between the modules. The user may know that two specific modules share data in a way that is not captured by the formal graph model extracted from the source code, and may use this knowledge by forcing these two specific nodes into the same module. As another example, the user may see that the constraints on cluster size are too tight to result in a clustering with good cohesion; in this case the user can relax the constraint.

2. **Guide the search**: the visualization may show that the algorithm is stuck in a local minimum, or is far from an acceptable value for the objective function. The user can give hints to guide the algorithm toward a better solution. For example, the user may see the algorithm is spending considerable time making small adjustments to a specific cluster with very poor cohesion. The user can destroy the bad cluster, forcing the algorithm to re-assign the nodes of the bad cluster to other clusters.

The user drives these hints in three main ways:

(a) **Constraints** – constraints are an essential part of clustering, as we saw in the definition of the \(k\)-way partitioning problem in Section 1. However, for strong user involvement we need to allow not only the definition of constraints prior the clustering process, but also allow dynamic adjustment of constraints at run-time (adding new constraints, removing old constraints, or changing the importance of a constraint). By changing the constraints at run-time, the user can guide the algorithm to converge to a different solution. Simple examples of global constraints that can be specified are:

- minimum and maximum numbers of clusters,
- bounds on the number of nodes in each cluster,
- limits on the variation in cluster size.

There can also be local constraints. For example, for two nodes \(a\) and \(b\), we can require:

- that \(a\) and \(b\) share a cluster,
- that \(a\) and \(b\) be in different clusters.

(b) **Objective function** – Although “good cohesion within a cluster” and “loose coupling between clusters” are undoubtedly desirable for a good quality clustering, there are many formal interpretations of these intuitive ideas. If the user sees that the current objective function is not leading to a clustering which has good quality in the domain, then the function can be changed at run-time.

(c) **Direct manipulation** – The user can directly operate on the clustering by:

- moving a node from one cluster to another,
• destroying a cluster,
• merging two clusters.

Another kind of hint, that is not so intuitive as the previous ones, but may also be considered is the choice of the clustering method. The user selects a more appropriate clustering algorithm from a library whenever the current method cannot improve the quality of the actual solution. In this case, the user can even decide to run the algorithm on the whole clustering or just on part of it that shows poor quality.

We assume that the user supplies hints through a graphical interface. The interface has to implement a way of easily inputting them as well as to provide some visualization that assures feedback to the user.

3.2 How the approach works

Six items take part in the computation: a graph $G$ to be clustered, a set $O$ of objectives and a set $R$ of constraints, a clustering $C$ of $G$, a vector $Q$ that measures the quality of $C$ according to $O$ and $R$, and a picture $P$ of $C$ ($P$ also displays some quality aspects from $Q$). See Figure 3 for a dependency graph of these items and the order in which they are set.

![Dependency Diagram](image)

The user can change $G$, $O$, $R$ and $C$ through the interface. As he or she does, the other items are recomputed to reflect the change.

Even though the three steps are shown sequentially, many tasks can be executed in parallel. For instance, the user can add new constraints while the clustering algorithm is working, to instantaneously modify the clustering computation.

Several kinds of clustering algorithms may be used in the clustering module. Some of them are mentioned in Section 2; another is presented in the next section. For the purposes of applying hints, however, algorithms that can work with a wide range of constraints and have a structure as in the loop of Figure 1 are most suitable. Soft computing methods, such as simulated annealing, are suitable for the job.

4. Experiments

In order to test our framework we developed a system called HINTS that implements some of the features described in the last section. The main resources of our system are:

• An interface for setting some global constraints (e.g. limits on clusters’ size and on the numbers of clusters, and maximum balance on clusters’ size). The user can activate or deactivate these constraints as well as set their relative importance.
• Simulated annealing and hill-climbing clustering algorithms, with flexible constraints.
• A visualization module that shows drawings and various measures of the current clustering.

Figure 4 shows a snapshot of the system. The way it works is described below.

Initial clustering. The user can select a graph through the File menu. After this, the system calls a function that computes an initial clustering of it (by a heuristic method).

Executing a clustering algorithm. A clustering method (at the moment, simulated annealing and hill-climbing are available) can be selected through the Algorithm menu. Buttons on the bottom left corner of the interface allow the user to start, pause, resume, or terminate the clustering process. A progress bar indicates how much of the processing remains based on fixed number of iterations. The clustering algorithms attempt to minimize...
a weighted function involving the constraints and the objective function:

\[ E = \text{Min}(O + R), \]
\[ O = w \* L \]
\[ R = \sum_{i=1}^{5} \lambda_i * f_i \]

where \( w \) is a constant, \( L \) is the number of edges between vertices in distinct clusters, \( f_1, f_2,..,f_5 \) are functions representing how distant the actual clustering is from the optimal solution for each constraint, and \( \lambda_1, \lambda_2,.., \lambda_5 \) represent the relative importance of the constraints. In general, \( w \) is set with a value small enough for making the constraints more important than the objective function. The system keeps track of the best solution, which is drawn on the screen.

Moreover, the clustering algorithms iterate only a fixed number of steps so they do not take much time for producing a reasonable quality solution. If the user is not happy with this solution, then it is possible to “re-cook” it for a longer period by pressing the button Start on the bottom left panel. In this case, the system uses the current computed solution as a seed for the reprocessing.

Another way of helping the system is to execute different clustering algorithms. The hill-climbing approach produces solutions quickly, however, it easily gets stuck in a local minimum. On the other hand, simulated annealing takes much longer for producing a solution, but in general achieves better results than the simple hill-climbing method. We can alternate executions of these two algorithms whenever the system is in a local minimum, in order to continue the improvement of the clustering.

**Visualization.** The interface offers feedback to the user through three main views: a histogram shows how many clusters there are by size, a scatter-plot style graphic shows the size of each cluster, and a graph drawing shows the current feasible clustering. See samples of these visualizations in Figure 5. Constraints are indicated in the views by drawing extra lines or by using different colors. For example, the dotted lines in Figures 5a and 5b represent the limits on clusters’ size. In Figure 5c, clusters whose sizes are out of the desired limits are drawn in red. Using these kinds of features, it is possible to visualize problems with the current solution.

The general evolution of the search algorithm is shown by continuously updating the visualization. Note that if the screen is not updated for a long time, this means that the system may have fallen into a local minimum and needs the user’s intervention.

**Remarks.** The current version of our system does not include more complex constraint-based hints, such as grouping vertices. However, the currently available facilities show a very promising path for considering the user as a strong part of the optimization process.

The system can be used for several interesting clustering problems that arise in many areas, as described in section 1. Particularly in Software Engineering, it can be used for identifying efficient ways of organizing modules of a program into libraries that present maximum cohesion and minimum coupling. It is also suitable for re-engineering legacy applications by automatically recovering their general structure and functionality.
Further, the general framework presented here aims to pave the way for the development of interactive systems for other kinds of complex problems that are NP-hard, multi-objective, multi-constraint and dynamic.

Figure 5: visualization of a clustering: a histogram graphic (a), a scatter-plot graphic (b), and a clustering drawing (c).

5. Related work

Anderson et al. [AAL+00] introduced the concept of human guided simple search and applied it to scheduling problems, most notably to a vehicle routing problem. Our concept of hint is closely related to the notion of human guidance in [AAL+00].

Following [AAL+00], an interactive system for clustering by Lesh, Marks and Patrignani [LMP00] shows an approach similar to the one presented here. Lesh et al. developed an interesting visualization technique by which users can analyze the structure of a clustering and identify changes that may reduce the cut size, leading to a better clustering. Their system implements part of the diagram shown in Figure 2 and works as follows: firstly, a clustering method is used to divide the graph into clusters; then drawings of the clustering are generated, through which the user can visualize nodes that are not strongly attached to their current cluster. Those nodes can then be moved manually to a more promising cluster, or the two clusters (the actual and the promising ones) can be sent to an algorithm for automatic improvement. When the clustering is updated, a new drawing is produced and the processing repeats.

Drawings in this system are generated using a force-directed method where nodes are represented by masses and a spring links a pair of vertices whenever there is an edge incident to these vertices in the graph. Moreover, additional springs connect each node to a hub representing its cluster. The hubs are fixed and positioned uniformly around a circle. By using this technique, a node that has many edges linking it to nodes in different clusters appears far from its hub, showing the weak attachment.

In our framework, it is possible to move vertices manually between clusters as in [LMP00], but it is also possible to define constraints that force some vertices to be put together into the same cluster, or even separately in distinct partitions. The constraints are included in the optimization process and change the way that it looks for quality solutions.

6. Conclusion

This paper is one of several current attempts to test the hypothesis that a human provided with a good interface can assist an optimization method. Our experience with HINTS, together with [AAL+00] and [LMP00], adds weight to the hypothesis.

We plan to extend our investigation by considering more kinds of hints and testing the system with users in real applications.
References


Requirements and Design
Design and Experimentation of a Fuzzy Logic Controller for Evaluating Domain Knowledge

Bedir Tekinerdoğan
TRESE Software Engineering group,
Dept. of Computer Science, University of Twente,
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{bedir@cs.utwente.nl}

Abstract
Software engineering can be seen as a problem-solving process in which a software solution is developed for a given problem. Thereby, one needs to capture the corresponding knowledge sources from which the fundamental concepts for the software solution can be extracted. The quality of the adopted knowledge sources intrinsically defines the quality of the software solution. This quality of knowledge sources is, on the one hand, determined by its objectivity value, and on the other hand, by its relevance value for the given problem. The relevance and objectivity values may change due to newly generated knowledge or evolving requirements. Crisp decisions that either result in accepting or rejecting the knowledge source may easily result in information loss. We propose to apply fuzzy logic techniques in which knowledge sources are assigned fuzzy linguistic quality values to express the quality degrees and as such to cope with the evolution of the knowledge sources, and the problems. We validate our proposal with an experimental case study on the evaluation of domain knowledge for the design of transaction systems.

keywords: fuzzy control, software engineering, domain analysis, synthesis-based design, case study

1. Introduction
Software engineering can be essentially seen as a problem solving process that aims to find software solutions for a given problem. The problem is typically initiated by the client's requirement specification and the solution is defined as a software program. Providing a solution for a given problem is not trivial and involves the accumulation and utilization of a huge amount of knowledge. This process of identifying the solution domain knowledge and extracting this knowledge to produce solutions is defined as solution domain analysis.

One of the core activities of the solution domain analysis process is the identification of the knowledge sources from which the necessary solution domain concepts will be extracted. To provide quality software it is necessary to elicit the important knowledge sources for a given problem, so that suitable solution abstractions can be identified. The corresponding domain knowledge space may be very large and evaluating the knowledge sources may as such be complicated. As a matter of fact, the quality of the adopted knowledge sources intrinsically defines the quality of the software solution.

This quality of knowledge sources is, on the one hand, determined by its objectivity value, and on the other hand, by its relevance value for the given problem. In practice, evaluation of knowledge domains is uncertain,
vague and as such based on the subjective interpretation of the domain engineer. Moreover, the relevance and
objectivity values may change due to newly generated knowledge or evolving requirements. Two-valued logic
based decisions that either result in accepting or rejecting the knowledge source do not conform with the
conceptual evaluation of the human engineer and may easily result in information loss.

We propose to apply fuzzy logic techniques in which knowledge sources are assigned fuzzy linguistic quality
values to express the quality degrees and as such to cope with the evolution of the knowledge sources, and the
problems. For this purpose we have developed a fuzzy controller to evaluate the relevance and objectivity
quality factors of the knowledge sources and determine the abstraction qualities. The fuzzy controller is
validated with an experimental case study in which a set of knowledge sources need to be evaluated by a
transaction domain expert and a group of novice transaction designers, for two distinct problems.

The remainder of the paper is organized as follows: In section 2, we define the background knowledge on
solution domain analysis. Section 3 defines the problem statement in evaluating knowledge domains. Section 4
describes the fuzzy control approach that we have adopted to evaluate knowledge domains. Section 5 provides a
case study for the design of atomic transaction systems. Finally, section 6 provides the related work and finally
section 7 that provides the conclusions.

2. Solution Domain Analysis

Solution domain analysis aims to identify the right solution domains for the given problems and extract the
relevant knowledge from these domains to come up with a feasible solution. Figure 1 represents a conceptual
model for illustrating the solution domain analysis process that we apply.

![Figure 1. The basic concepts in solution domain analysis](image)

Hereby, the rounded rectangles represent the concepts and the directed arrows represent the associations
between these concepts. The figure typically illustrates the relations between the given problem, the solution
domain analysis process and the extracted solution domain concepts that forms the output of the solution
domain analysis process. The concept **Technical Problem** represents the problem that needs to be solved and
likewise forms an input to the solution domain analysis process. For every **Technical Problem** a solution is
provided by one or more **Solution Domains**. The concept **Technical Problem** includes zero or more **Sub-Problems**.
The concept **Solution Domain** represents the set of **Knowledge Sources** that provide the concepts for solving the
problem. From every **Knowledge Source** one or more **Solution Domain Concepts** can be derived.

For the overall problem and each sub-problem we search for the solution domains that provide the solution
abstractions to solve the technical problem. The solution domains for the overall problem are more general than
the solution domains for the sub-problems. In addition, each sub-problem may be recursively structured into
sub-problems requiring more concrete solution domains on their turn.

Each identified solution domain may cover a wide range of solution domain knowledge sources. These
knowledge sources may not all be suitable and vary in quality. For distinguishing and validating the solution
domain knowledge sources we basically consider the quality factors of **objectivity** and **relevance**. The objectivity
quality factor refers to the solution domain knowledge sources itself, and defines the general acceptance of the
knowledge source. Solution domain knowledge that is based on a consensus on a community of experts has a higher objectivity degree than solution domain knowledge that is just under development. The relevance factor refers to the relevance of the solution domain knowledge for solving the identified technical problem.

The relevance of the solution domain knowledge is different from the objectivity quality. A solution domain knowledge entity may have a high degree of objective quality because it is very precisely defined and supported by a community of experts, though, it may not be relevant for solving the identified problem because it addresses different concerns. To be suitable for solving a problem it is required that the solution domain knowledge is both objective and relevant.

The evaluation of a knowledge source based on the quality factors of relevance and objectivity will result in the quality factor that we term abstraction quality [Aksit 00]. The abstraction quality defines the importance of the corresponding knowledge source for extracting solution concepts. The highest abstraction quality is achieved when both the relevance and the objectivity of the knowledge source are high. A high abstraction quality of the knowledge source means that it provides the fundamental concepts for producing a solution with high quality, that is, a solution that fully meets the requirements and which is stable.

The relation between the three quality factors may be given in the following empirical formula [Aksit 00]:

\[
\text{Abstraction Quality} (ks) = \text{Objectivity}(ks), \text{Relevance}(ks)
\]

Hereby \text{Abstraction Quality}(), \text{Objectivity}() and \text{Relevance}() represent functions that define the corresponding quality factors of the argument \(ks\), that stands for solution domain knowledge source. For solving the problem, first the solution domain knowledge with the higher abstraction qualities is utilized. The measure of the objectivity degree can be determined from general knowledge and experiences. The measure for the relevance factor can be determined by considering whether the identified solution domain source matches the goal of the problem. Note, however, that this formula should not be interpreted too strictly and rather be considered as an intuitive and practical aid for prioritizing the identified solution domain knowledge sources.

### 3. Problem Statement

A simple approach to evaluate the available knowledge sources for a given problem is to provide Boolean variables \textit{objective, relevant, and abstraction quality} for each knowledge source and as such assign either the values true or false to it. Typically we could express the corresponding heuristic rule as follows:

\[
\text{IF knowledge source is \textit{RELEVANT} for the problem AND \textit{OBJECTIVE}} \\
\text{THEN knowledge source has \textit{ABSTRACTION QUALITY}}
\]

If we apply traditional two-valued logic for this rule, then a knowledge source either completely possesses the qualities of relevancy, objectivity and abstraction quality, or it does not. This implies that a knowledge source possesses the abstraction quality only in case it is both considered relevant and objective.

In practice, the process of domain analysis, though, is complex and often related to subjective evaluations, vagueness and uncertainty. Therefore, for a more practical and precise evaluation of the knowledge sources we state that the following three requirements are necessary:

1. **Expressing the degree of quality**

The evaluation of the objectivity and the relevance value of knowledge domains are basically dependent on the background and expertise of the domain engineer. For a given knowledge source it may be hard to decide whether it completely possesses the quality factors or not. Rather, the domain engineer may decide that it partially possesses the objectivity, relevance and the abstraction quality. Formally, this means that a knowledge source \(ks\) of a solution domain \(SD\) is mapped to a number in \([0,1]\). This holds for all the three quality factors:
Relevance(ks) : SD $\rightarrow [0, 1]$
Objectivity(ks) : SD $\rightarrow [0, 1]$
Abstraction Quality(ks) : (Objectivity(ks), (Relevance(ks))) $\rightarrow [0, 1]$

2. **Need for linguistic evaluation of knowledge sources**

Knowledge sources may be evaluated by assigning numbers to their corresponding quality factors. In practice, however, this is counter to the intuition of the domain engineer, which is rather based on linguistic evaluations, such as fairly, substantially, possibly etc. To cope with this, the domain knowledge evaluation approach must therefore provide means to express quality factors using natural linguistic terms to facilitate the communication about their decisions.

3. **Providing means to cope with evolution of knowledge and problems**

As a matter of fact, knowledge domains are not static but evolve over time. On the one hand, knowledge domains may become obsolete and less useful for solving a problem. On the other hand, they may become of more value, for example, after one has better understood the problem. In addition to the evolution of knowledge, the requirements may evolve as well, and as such the corresponding problem that needs to be solved may change in parallel. Both cases, that is, evolution of knowledge and evolution of problems, may impact the value of the quality factors of the knowledge sources. This implies that the evaluation of the quality of knowledge sources should not be absolute but adaptable to the changing context. Adopting two-valued logic inherently leads to the absolute elimination or acceptance of the knowledge sources and as such fails to cope with this evolution of knowledge and problems appropriately. To address this evolution properly, it is required that knowledge sources are preserved and their quality is adapted if the context changes.

4. **Fuzzy Knowledge Source Evaluator**

We believe that the evaluation of knowledge domains may be more effectively supported by the use of fuzzy logic and in particular fuzzy control techniques. For this purpose we have designed a fuzzy control system for evaluating domain knowledge, which is illustrated in Figure 2.

![Fuzzy Controller for defining Abstraction Quality of Knowledge Sources](image)

**Figure 2.** Fuzzy Controller for defining Abstraction Quality of Knowledge Sources

Note that Figure 2 is an elaboration on the model of Figure 1 in that it provides a fuzzy controller, called Fuzzy Knowledge Source Evaluator (FKSE). The FKSE follows the general structure of fuzzy controllers [Klir & Yuan 95] and consists of the four modules Fuzzifier, Fuzzy Inference Engine, Fuzzy Rule Base, and Defuzzifier. The basic inputs for FKSE are the values for the relevance and the objectivity quality factors of the knowledge source, which are used to compute the value for abstraction quality.

The membership functions for the input linguistic variables, relevance and objectivity, as well as the output linguistic variable abstraction quality are given in Figure 3. From an experimental perspective we have applied the triangular membership functions, though, other membership functions may be adopted as well. The optimal membership functions may be determined, for example, after a set of comparative experiments. We will not
elaborate on this topic in this paper. For the triangular membership functions of the quality factors we have adopted five linguistic values: weakly, slightly, fairly, substantially, and strongly.

Figure 3. Shape of membership functions of the linguistic input variables Relevance, Objectivity and the linguistic output variable Abstraction Quality

The evaluation of the relevance and objectivity quality factors are provided to the module Fuzzifier. The evaluation may be both expressed numerically, for example as a crisp value in [0..1] or linguistically using one of the five linguistic values. In the first case, the module Fuzzifier takes these crisp values as input and maps these into their membership functions and truth-values. The resulted fuzzy set is then provided to the module Fuzzy Inference Engine. In the latter case, the Fuzzifier provides the fuzzy set directly to the module Fuzzy Inference Engine.

The module Fuzzy Inference Engine uses the fuzzified values to evaluate the control rules that are stored in the Fuzzy Rule Base. The adopted meta-rule in this fuzzy rule base is as follows:

\[
\text{IF knowledge source is } \text{<relevance value> RELEVANT and <objectivity value> OBJECTIVE}
\]

\[
\text{THEN knowledge source has <abstraction quality value> ABSTRACTION QUALITY.}
\]

This meta-rule can be parameterized with one of the five linguistic values for the quality factors, to define sub-rules. Since, both the relevance and objectivity quality factors have five linguistic values, we can derive 25 rules from this meta-rule. These rules are represented in Table 1.

<table>
<thead>
<tr>
<th>ABSTRACTION QUALITY</th>
<th>weakly</th>
<th>slightly</th>
<th>fairly</th>
<th>Substantially</th>
<th>strongly</th>
</tr>
</thead>
<tbody>
<tr>
<td>weakly</td>
<td>weakly</td>
<td>weakly</td>
<td>slightly</td>
<td>slightly</td>
<td>fairly</td>
</tr>
<tr>
<td>slightly</td>
<td>slightly</td>
<td>slightly</td>
<td>fairly</td>
<td>fairly</td>
<td>fairly</td>
</tr>
<tr>
<td>fairly</td>
<td>slightly</td>
<td>slightly</td>
<td>fairly</td>
<td>fairly</td>
<td>substantially</td>
</tr>
<tr>
<td>substantially</td>
<td>slightly</td>
<td>fairly</td>
<td>fairly</td>
<td>substantially</td>
<td>substantially</td>
</tr>
<tr>
<td>strongly</td>
<td>fairly</td>
<td>fairly</td>
<td>substantially</td>
<td>substantially</td>
<td>strongly</td>
</tr>
</tbody>
</table>

Table 1. Rules for inferring abstraction quality of knowledge sources with equal weighting factors for the input variables relevance and objectivity

The first cell of the table, for example, represents the following fuzzy sub-rule:

\[
\text{IF knowledge source is } \text{<weakly> RELEVANT and <weakly> OBJECTIVE}
\]

\[
\text{THEN knowledge source has <abstraction quality value> ABSTRACTION QUALITY.}
\]

In Table 1, the relevance quality and objectivity quality have equal weight and as such the table is symmetric along the upper left to right bottom diagonal axis. Alternatively, one may consider the relevance quality factor more important than the objectivity quality factor. In that case, the relevance quality factor must have a larger weight than the objectivity quality factor. Based on this assumption we developed the fuzzy rules as given in Table 2. Note that this table is not symmetric anymore.

The module Fuzzy Inference Engine executes the fuzzy rules given the input fuzzy set. In our FKSE we have defined the max-min inferencing method as described in [Broek 99] to derive the results of a rule, although, other inferencing methods may be adopted equally. We do not further discuss this with respect to the scope of the paper.
After the module Fuzzy Inference has executed the fuzzy rules it generates a fuzzy output set, which is then provided to the module Defuzzifier. The module Defuzzier converts the provided fuzzy set into a crisp value for the abstraction quality. For the defuzzification the centroid method is applied.

Once the knowledge sources have been assigned values for abstraction quality, the domain engineer can do an explicit trade-off analysis to decide whether the corresponding knowledge source fulfills the required quality to extract the solution domain concepts. An appropriate evaluation will as such improve the quality of the solution abstractions that are derived from selected knowledge sources.

### 5. Case Study: Evaluating Transaction Domain Knowledge

In this section, we will illustrate the application of the fuzzy controller using an experimental case study on the evaluation of the knowledge domains for the design of atomic transaction systems. The goal of the case study is to validate the applicability of our knowledge evaluation approach to real design cases, both for domain experts, and designers who are inexperienced in the corresponding domain.

5.1 presents the set of selected knowledge sources that need to be evaluated against two distinct transaction design problems. Section 5.2 illustrates the application of the fuzzy controller by adopting the evaluation of the given knowledge sources by a transaction domain expert. Section 5.3 adopts and discusses the evaluation of the knowledge sources by novice transaction system designers. Finally, in section 5.4 we provide the conclusions of this experimental case study with respect to the earlier defined requirements in section 3.

#### 5.1 Selection of Problem and Knowledge Sources

We can derive a large number of publications on the theory of transaction systems. Informally atomic transactions are characterized by two properties: serializability and recoverability [Bernstein 87]. Serializability means that the concurrent execution of a group of transactions is equivalent to some serial execution of the same set of transactions. Recoverability means that each execution appears to be all or nothing; either it executes successfully to completion or it has no effect on data shared with other transactions.

Many different transaction systems can be designed by extracting various concepts from the transaction literature. Although a common architecture for transaction systems can be derived, transaction systems may nevertheless differ in the selected transaction protocols, such as transaction management, concurrency control protocols, recovery protocols, and data management techniques. Obviously, every transaction system design may need its own dedicated knowledge. The kind of transaction system is basically defined by the corresponding problems. In our experimental case study we adopted the following two distinct design problems.

**Example Problem 1:**

Design a transaction system for a distributed system which is based on a two-phase locking concurrency control scheme and a recovery scheme that handles transaction failures based on image logging. The transaction management needs to consider only flat transactions.
Example Problem 2:
Design an advanced transaction system with adaptable transaction properties. In addition to flat transactions, the system must be able to compose nested transactions as well. The choice of the concurrency control and the recovery protocols must be made adaptable according to the performance characteristics of the system.

Note that the first design problem requires the design of a transaction system with fixed properties while the second design problem requires adaptable transaction protocols. To provide a solution for the first problem requires knowledge on the specific protocols, whereas the second problem requires knowledge on a wide range of transaction protocols and additionally it requires knowledge on performance of transaction protocols, and adaptation protocols to switch between the various protocols.

The knowledge sources that needed to be evaluated are presented in Table 3. We have deliberately selected knowledge sources that differ from each other to capture the impact of the fuzzy reasoning process. Differences can be observed with respect to the form of the knowledge source, the date of publication, the location of the publication, the generality of the publication etc.

<table>
<thead>
<tr>
<th>KS</th>
<th>KNOWLEDGE SOURCE</th>
<th>FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>KS1</td>
<td>Concurrency Control &amp; Recovery in Database Systems [Bernstein et al. 87]</td>
<td>Textbook</td>
</tr>
<tr>
<td>KS2</td>
<td>Atomic Transactions [Lynch et al. 94]</td>
<td>Textbook</td>
</tr>
<tr>
<td>KS3</td>
<td>An Introduction to Database Systems [Date 90]</td>
<td>Textbook</td>
</tr>
<tr>
<td>KS4</td>
<td>Database Transaction Models for Advanced Applications [Elmagarmid 92]</td>
<td>Textbook</td>
</tr>
<tr>
<td>KS5</td>
<td>The design and implementation of a distributed transaction system based on atomic data types [Wu et al. 95]</td>
<td>Journal paper</td>
</tr>
<tr>
<td>KS8</td>
<td>Course Notes of Transaction Design</td>
<td>Course Notes</td>
</tr>
<tr>
<td>KS10</td>
<td>Conference Proceedings on Advanced Transaction Systems and Applications</td>
<td>Proceedings</td>
</tr>
<tr>
<td>KS12</td>
<td>Transaction Domain Expert with 15 years of experience</td>
<td>Person</td>
</tr>
<tr>
<td>KS14</td>
<td>Nested Transactions [Moss 85]</td>
<td>PhD. thesis</td>
</tr>
</tbody>
</table>

Table 3. A selected set of knowledge sources for the overall solution domain

5.2 Evaluation by a Domain Expert

Table 4 represents the evaluation of relevance and the objectivity of the knowledge sources by a transaction domain expert who has experience in the theory, design and implementation of a wide range of transaction systems.

<table>
<thead>
<tr>
<th>KS</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevance P1</td>
<td>ST</td>
<td>FA</td>
<td>SL</td>
<td>WE</td>
<td>SU</td>
<td>WE</td>
<td>SL</td>
<td>SL</td>
<td>FA</td>
<td>WE</td>
<td>WE</td>
<td>ST</td>
<td>FA</td>
<td>FA</td>
<td>WE</td>
<td>SU</td>
</tr>
<tr>
<td>Relevance P2</td>
<td>SU</td>
<td>FA</td>
<td>WE</td>
<td>ST</td>
<td>SL</td>
<td>ST</td>
<td>WE</td>
<td>WE</td>
<td>ST</td>
<td>FA</td>
<td>WE</td>
<td>ST</td>
<td>SU</td>
<td>SU</td>
<td>ST</td>
<td>WE</td>
</tr>
<tr>
<td>Objectivity</td>
<td>SU</td>
<td>ST</td>
<td>FA</td>
<td>SU</td>
<td>ST</td>
<td>ST</td>
<td>FA</td>
<td>SL</td>
<td>SU</td>
<td>SU</td>
<td>FA</td>
<td>ST</td>
<td>FA</td>
<td>ST</td>
<td>SU</td>
<td>WE</td>
</tr>
</tbody>
</table>

Table 4. The evaluation of the relevance and objectivity of knowledge sources by a transaction domain expert

During the evaluation all the knowledge sources were actually made available and could be analyzed. The knowledge sources have been evaluated for both problem 1 and problem 2. Since relevance is dependent on the problem, the table provides one row for the relevance of each of the two problems. In contrast, objectivity is problem independent and as such includes only one row.
Using the input values for the relevance and objectivity quality factors, the fuzzy knowledge evaluator can infer 
the abstraction quality for each knowledge source. To illustrate the inference mechanism we consider, for 
example, the inference of the abstraction quality for knowledge source KS4 for problem 1. For this knowledge 
source the fuzzy relevance value is \textit{weakly}, for problem 1, and its objectivity value is \textit{substantially}. Considering the 
membership functions as illustrated in Figure 3, we can derive that for the linguistic variable relevance, the fuzzy 
value \textit{weakly} overlaps with fuzzy value \textit{slightly}. For the linguistic variable objectivity, the input value \textit{substantially} 
overlaps with the values \textit{fairly} and \textit{strongly}. The inference engine will fire all rules but in the end only those rules 
for which relevance is \textit{weakly} or \textit{slightly}, and objectivity is \textit{fairly}, \textit{substantially} or \textit{strongly}, will have an impact on the 
final result. For the experimental case study we adopted the rules as defined in Table 1, that is, \textit{relevance} and 
\textit{objectivity} of knowledge sources have equal weight. As such, for determining the fuzzy set of the abstraction 
quality of KS4, the following six rules will have a impact:

1. \textbf{IF} knowledge source is \textit{weakly} RELEVANT and \textit{fairly} OBJECTIVE \textbf{THEN} knowledge source has \textit{slightly} ABSTRACTION QUALITY.
2. \textbf{IF} knowledge source is \textit{weakly} RELEVANT and \textit{substantially} OBJECTIVE \textbf{THEN} knowledge source has \textit{slightly} ABSTRACTION QUALITY.
3. \textbf{IF} knowledge source is \textit{weakly} RELEVANT and \textit{strongly} OBJECTIVE \textbf{THEN} knowledge source has \textit{strongly} ABSTRACTION QUALITY.
4. \textbf{IF} knowledge source is \textit{slightly} RELEVANT and \textit{fairly} OBJECTIVE \textbf{THEN} knowledge source has \textit{slightly} ABSTRACTION QUALITY.
5. \textbf{IF} knowledge source is \textit{slightly} RELEVANT and \textit{substantially} OBJECTIVE \textbf{THEN} knowledge source has \textit{fairly} ABSTRACTION QUALITY.
6. \textbf{IF} knowledge source is \textit{slightly} RELEVANT and \textit{strongly} OBJECTIVE \textbf{THEN} knowledge source has \textit{fairly} ABSTRACTION QUALITY.

The execution of these rules results in the fuzzy set that is illustrated in Figure 4a. The execution of the fuzzy 
rules for problem 2 will yield the fuzzy set as given in Figure 4b. Note that for problem 2 the set of rules that 
provide impact on the final result will be different than for problem 1 because of the different value for relevance. 
Using the centroid method the module Defuzzifier computes a crisp value of the fuzzy sets, which are 0.35 and 
0.67 for problem 1 and problem 2, respectively. These numbers give the software engineer a practical indication 
of the quality of the knowledge source. In this case, KS4 has a clearly higher abstraction quality for problem 1 
than for problem 2.

\textbf{Figure 4.} Fuzzy set of the inferred abstraction qualities for knowledge source 4 for a) problem 1 and b) problem 2

Similar to KS4, the fuzzy knowledge evaluator yields the fuzzy sets for the other knowledge sources. Figure 5 
shows the defuzzified values of the abstraction qualities of the sixteen knowledge sources, both for problem 1 
and problem 2. This figure gives already an hint of the quality of the various knowledge sources and this 
information is valuable for the software engineer who needs to extract the abstractions from the knowledge 
sources to develop the solution. Let us now take a closer look at Figure 5 and interpret the resulted abstraction 
qualities.

A global look at the figure shows, that knowledge sources that only deal with advanced transactions (4, 9, 10), 
performance modeling (6) and dynamic adaptation (13,15) have got a low abstraction quality for problem 1, that 
is, the design of a flat transaction system with fixed properties.
Knowledge source 3 has got a low abstraction quality for both problems. This may be because it is not directly or explicitly related to the subject of transaction systems.

Knowledge source 5 has the highest abstraction quality for problem 1 but a lower value for problem 2. This may be attributed to the fact that it is both a journal paper, leading to a strong objectivity value, and because it is directly related to the design of flat transaction systems.

Knowledge source 6, the journal paper on performance modeling, has the highest abstraction quality for problem 2 but a rather low abstraction quality for problem 1. This may be due to its irrelevance to problem 1. It is not the lowest abstraction quality for problem 1 because it has been evaluated as strongly objective, which indirectly increases the abstraction quality.

Knowledge source 12, the transaction domain expert, has the highest abstraction quality for both problems. This may be explained from the fact that the expert knows both problems well and possesses recent and strongly objective knowledge.

Knowledge source 13, a MSc thesis on the design of adaptable transaction systems, although strongly relevant for problem 2, has not got the highest abstraction value. This may be because it is not recent and the fact that it is a MSc thesis.

The above observations show that the evaluation of the knowledge sources using fuzzy linguistic rules is not ad hoc and reasonably match the intuition. In addition, we can observe that the abstraction qualities have indeed been evaluated differently for both problems, and this difference also provides a sound conceptual interpretation. The knowledge sources have now been more precisely evaluated than in case of a two-valued logic evaluation approach, because every evaluation closely matches the intuition and as such no information is lost.

5.3 Evaluation by Novice Transaction System Designers

To determine the validity and the applicability of the heuristic fuzzy rules for inexperienced domain engineers, we have presented the 16 knowledge sources in Table 3, also to novice transaction system designers. This group consisted of 25 fourth year students of Computer Science at the University of Twente, in the lecture on object-oriented software analysis and design. The students got a week before a one and half-hour lecture on solution domain analysis and evaluation of knowledge sources based on the relevance and objectivity quality factors. Similar to the evaluation by the transaction domain expert, the students got the actual knowledge sources during the experimental case study, and evaluated these one by one, separately and independently. The students did not know in advance, though, that they were involved in an experimental case study, but considered the evaluation process more as a practical assignment. The evaluation of the knowledge source took about two hours and consisted again of assigning the five fuzzy values of weakly, slightly, fairly, substantially, and strongly to the
corresponding knowledge sources. We have collected and processed all the results of all students and did not eliminate any.

For every knowledge source we have counted the frequency of the five linguistic fuzzy values and represented these in histograms as illustrated in the Appendix. For example, for the relevance of knowledge source 1 for problem 1, none student assigned the values *weakly* and *slightly*, 2 students assigned the value *fairly*, 12 students thought that it was *substantially*, and 11 students assigned the value *strongly*.

We have implemented an algorithm for deriving the fuzzy sets of a given histogram. Hereby, the average of the 25 fuzzy values in each histogram is taken to yield a new fuzzy set. Consider for example the computation of the average fuzzy set for the relevance of knowledge source 1 (KS1) for problem 1. The histogram of KS1 can be fuzzified by taking the average of 2 *fairly*, 12 *substantially*, and 11 *strongly* fuzzy values. The resulted fuzzy set is shown in Figure 6.

![Figure 6](image.png)

*Figure 6. Fuzzy set for the relevance of knowledge source 1 for problem 1, derived from the histogram of the evaluation of novice transaction system designers*

We have done this for each histogram and derived the fuzzy sets of the relevance and objectivity quality factors of each knowledge source. The fuzzy sets have then been provided as an input to the fuzzy knowledge evaluator to provide the abstraction qualities. The results of the defuzzified sets of the abstraction qualities are shown in Table 5.

<table>
<thead>
<tr>
<th>KS</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defuzzified Abstraction Quality P1</td>
<td>0.67</td>
<td>0.65</td>
<td>0.33</td>
<td>0.35</td>
<td>0.67</td>
<td>0.4</td>
<td>0.33</td>
<td>0.33</td>
<td>0.5</td>
<td>0.35</td>
<td>0.33</td>
<td>0.8</td>
<td>0.5</td>
<td>0.65</td>
<td>0.35</td>
<td>0.35</td>
</tr>
<tr>
<td>Defuzzified Abstraction Quality P2</td>
<td>0.52</td>
<td>0.62</td>
<td>0.42</td>
<td>0.50</td>
<td>0.50</td>
<td>0.53</td>
<td>0.58</td>
<td>0.44</td>
<td>0.51</td>
<td>0.62</td>
<td>0.49</td>
<td>0.58</td>
<td>0.55</td>
<td>0.53</td>
<td>0.95</td>
<td>0.43</td>
</tr>
</tbody>
</table>

*Table 5. Inferred (Defuzzified) Abstraction Quality of a group of novice transaction designers*

To interpret and validate these results we will provide a comparison with the evaluation of the transaction domain expert. Figure 7 shows a comparison between the evaluation of the inferred evaluation values for the domain expert and the novice group.

![Figure 7](image.png)

*Figure 7. Comparison of defuzzified abstraction qualities of domain expert and novice transaction designers for (a) problem 1 and (b) problem 2*

A first glimpse at Figure 7 makes clear that the novice group is more careful in their evaluation than the transaction domain expert, because for both problems the difference among the abstraction qualities of the various knowledge sources is less than in the evaluation by the domain expert. This may be attributed to the self-confidence of the domain expert who has a complete and objective overview of the domain and as such can make...
more sharp decisions. The students on the other hand may be more careful if they are not sure about their decisions.

If we take a look at the histograms we can observe that there is a reasonable consensus among the 25 students for the evaluation of the knowledge sources. For some knowledge sources (such as 1, 6, 12, 14 and 15) this is more obvious, than others (such as 2, 5, 11, 16). The lack of consensus may show the difficulty of the evaluation, which is influenced by the background and experience of the individual students.

The experience factor of the novice group is obvious for the evaluation of knowledge source 2, a textbook including many formal algorithms and proofs on basically nested transactions. For the relevance quality for problem 1 we can observe that the students do not form a common opinion. This may be attributed due to their lack of experience in interpreting and applying formal algorithms. The values for the objectivity, however, are commonly assigned high values. The reason for this may be that although they cannot precisely evaluate the relevance they have confidence in its scientific objectivity because of the many formulas.

The students were able to distinguish the relevancy of the knowledge sources for both problems. If we compare the defuzzified abstraction qualities with that of the abstraction qualities inferred from the domain expert's evaluation we can observe that the difference in the quality values are reasonably similar. For example, for KS6 the evaluation of the domain expert resulted in the values of 0.4 and 0.8 (Figure 5), for problem 1 and problem 2, respectively. The evaluation by the novice group resulted in the same increasing order for the abstraction quality of 0.4 and 0.53 (Table 5), for problem 1 and problem 2, respectively. Although, the domain expert's evaluations are more sharp, the application of the fuzzy heuristic rules resulted generally in the same ordering of the abstraction qualities of the knowledge sources. As a matter of fact, this ordering of the knowledge sources is of higher importance than the assigned values.

We may infer many more conclusions from this experimental case study, though, due to space limitations we will not elaborate on it.

5.4 Conclusions of the Overall Case Study

The goal of the case study was to validate the applicability of our knowledge evaluation approach with respect to the identified requirements. In the previous two sections we have respectively discussed the evaluations of the domain expert and the novice group individually. For both cases we have shown that the fuzzy knowledge evaluator is useful and applicable.

First of all, the use of linguistic values for the assessment of the knowledge sources was applied quite straightforward; the evaluation of knowledge sources was done in a rather short period (about 2 hours) and every student could evaluate the knowledge sources independently.

Instead of a classification of accepted and rejected knowledge source as it would result from using two-valued logic, there is a clear graded ordering of the knowledge sources, which likewise provide a more precise evaluation. This is valuable for selecting the knowledge sources for extracting the solution abstractions.

The evaluation of both the domain expert and the novice group illustrates the shift in the values for the abstraction qualities for problem 1 and problem 2. Our approach does not need to eliminate knowledge sources and as such the abstraction qualities may be changed according to the evolution of the context, that is, problems and knowledge.

The reasonable some ordering of the inferred abstraction qualities from the domain expert and the novice group show that the fuzzy rules may be successfully applied by The novice group was more careful and had more problems in the evaluation, possibly due to the lack of general scientific experience.
6. Related Work

The solution domain analysis process basically forms the core of our earlier work on the synthesis-based design process [Tekinerdogan 00]. Synthesis is a problem solving approach that is applied in mature engineering disciplines such as electrical engineering, mechanical engineering and chemical engineering. The synthesis-based design process consists basically of the sub-processes of technical problem analysis, solution domain analysis and alternative design space analysis. The technical problem analysis phase aims to define the technical problems that have been initiated by the requirement specification. The solution domain analysis aims to find the corresponding solution domains for the given problem and extracts solution domain concepts from these domains. The alternative design space analysis aims to define the space of the alternatives for the given problem and evaluates these against quality criteria. This paper focuses on the solution domain analysis process of the synthesis-based design process. We aim to formalize the whole synthesis process and apply fuzzy control were necessary. The next step to integrating fuzzy control in the synthesis-based design process will be the evaluation of the extracted solution abstractions.

Fuzzy control has been applied in many different fields but very few of them in the area of software engineering. In [Aksit & Marcelloni 00] fuzzy control is adopted to enhance object-oriented methods by modeling and controlling the design alternatives. The authors maintain that design alternatives during the software development process needs to be preserved to allow further refinements. Two-valued logic is not able to meet this requirement and eliminates alternatives too early.

Several domain analysis processes have been published, e.g. [Kang et al. 90], [Prieto-Diaz & Arrango 91], [Simos et al. 96] and [Czarnecki & Eisenecker 00]. Two surveys of various domain analysis can be found in [Arrango 94] and [Wartik & Prieto-Diaz 92]. In [Czarnecki & Eisenecker 00], a more recent and extensive up-to-date overview of domain engineering methods is provided. The fuzzy evaluation approach in this paper can be applied to all of these domain analysis methods.

Solving a problem requires first identifying the corresponding solution domains. This may a difficult task if the domain knowledge space is very large. To support the search for the right solution domains we may categorize the domain knowledge [Glass & Vessey 95]. Software engineering applies knowledge of a wide range of application domains, one of them especially is the Computer Science domain, such as programming languages, operating systems, analysis and design methods etc. This type of knowledge has been recently compiled in the so-called Software Engineering Body of Knowledge [Bourque et al. 99].

7. Conclusion

Software engineering is a problem solving process in which domain knowledge needs to be applied to provide software solutions. The quality of the produced software solution is intrinsically related to the selected knowledge sources. It is therefore necessary to elicit the important knowledge sources for a given problem, so that suitable solution abstractions can be identified. To be useful for solving a problem, the knowledge source must be relevant to the problem and have objective quality, which together define the so-called abstraction quality. A simplistic approach to evaluate domain knowledge is by using two-valued logic, whereby a knowledge source inherently either has the abstraction quality or not. In practice, however, the software engineer may decide that it partially possesses the abstraction quality and need to express these in natural language. In addition, the evaluation of the knowledge sources cannot be absolute but need to change along with the evolution of the available knowledge and the given technical problems. Adopting two-valued logic leads either to the absolute elimination or preservation of the identified knowledge sources.

To cope with these requirements, we have provided a model and an approach for evaluating domain knowledge using fuzzy logic techniques, the so-called fuzzy knowledge source evaluator (FKSE). The FKSE takes as input the relevance and the objectivity (fuzzy) values of a knowledge source and computes the abstraction quality. The inference engine adopts 25 fuzzy heuristic rules.
We have illustrated the application of FKSE in an experimental case study on evaluating domain knowledge for the design of atomic transactions. Thereby, we have provided two distinct problems to a domain expert and a group of novice transaction system designers. From the case study we concluded that applying fuzzy logic techniques in evaluating domain knowledge is of practical use and can support the solution domain analysis process.

Acknowledgements

I would like to thank Mehmet Aksit for his discussion on the subject of this paper, Pim van den Broek for providing me his Java implementation of the fuzzy algorithms, and Lodewijk Bergmans for helping me to extend the Java implementations for deriving fuzzy sets from frequency histograms. Further, I thank the students who have cooperated in the experimental case study on evaluating knowledge sources for the transaction domain.

This research is supported by the Dutch Scientific Organization (NWO).

References


References of the knowledge sources of the case study


### Appendix - Evaluation of Knowledge Sources by Novice Group

<table>
<thead>
<tr>
<th>KS</th>
<th>Relevance DP1</th>
<th>Relevance DP2</th>
<th>Objectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>KS1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KS16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Security Concepts for Agents-Based Systems

Georg Lehrenfeld*, Wolfgang Mueller**, Rainer Tellmann*
*Paderborn University
**C-LAB
Fuerstenallee 11
33102 Paderborn, Germany

Abstract

In the context of the growing number network-based solutions secure concepts in mobile agents communication and migration are of emerging interest. This article introduces extensions to FIPA 98 security management concepts. After a short introduction to security and FIPA this article presents additional FIPA security management extensions and their application by the example of document management system.

1 Introduction

With the wide acceptance of network-based solutions, secure exchange of data and components is of large interest in various commercial applications. In the context of network-based solutions agent-based approaches are of emerging interest. Today, FIPA 97 and FIPA 98 has been largely accepted as a standard for the implementation and management of agent-based systems. Unfortunately, the secure transmission of data and mobile agents has not been elaborated very well so far.

In this article, we present a FIPA-compliant runtime architecture for the secure communication and component exchange (i.e., secure mobile agents) in agent-based systems. It covers all attacking scenarios for mobile agents:

(i) agent attacking the communication between other agents,
(ii) platform attacking an agent and
(iii) agent attacking a platform.

While the first is solved by existing encryption techniques and the latter is more or less solved by sandbox techniques, the second problem is often ignored. In this article, we propose an extension of existing and some new FIPA-compliant ontologies and well as an extension of the FIPA agent management platform. The agent management platform is extended by a so-called Domain Manager (DM), Certification Manager (CM), and a Non-Repudiation Manager (NRM). Additional, extensions are required for additional ontologies as well as for parameters of existing ontologies. Based on this extended platform we discuss potential applications by the example of a documents management system.

The remainder of this article is structured as follows. The next section gives a basic introduction to important security concepts and algorithms. Thereafter, we introduce the basic concepts of FIPA focusing on the security issues in FIPA 98. Section 5 presents our security extensions for FIPA 98 and discusses an application in the field of document management. After a very short outline of the current status of our implementation this article closes with a conclusion.

2 Security

Security services can be roughly divided into

- data encryption
- authentication
- certification

For data encryption, there exists quite a couple of symmetric and asymmetric encryption algorithms which are in use in various applications like smart cards, SIM cards for mobile phones, SSL transmission in web browsers etc. Symmetric encryption is based on the computation of complex permutations and substitutions with one private key. Important algorithms are DES (Data Encryption Standard), Triple-DES (2/3-DES), IDEA (International Data Encryption Algorithm), RC2, RC4, and RC5. Symetric encryption is
popular for devices with limited resources due to the less complex decryption algorithm. Asymmetric techniques encrypt with a public key and decrypt with a public and private key pair. Important algorithms are RSA (Rivest/Shamir/Adleman), EC (elliptic curves), and DSA (Digital Signature Algorithm), where the latter is mainly used for the encryption of signatures.

In implementation of both principles, keys have to be encrypted when being stored in a digest. Encoding is done by so-called hashing algorithms or functions. Important algorithms are defined by ISO/IEC 10118-2, MD5 (message digest 5), and SHA-1. The latter is a NIST standard developed in the context of DSA.

Secure communication requires to verify the identity of a trusted communication partner, i.e., authentication. We can distinguish uni- and bi-directional authentication. In the first case only one partner is identified. In the latter case both partners are verified. One popular approach for uni-directional authentication is the challenge-response protocol. One partner sends a challenge. From the response of the partner one can see if it can be trusted or not. The challenge-response approach is mostly based on symmetric/asymmetric encryption. In the context of smart cards and asymmetric encryption, for instance, the terminal can send the card a random number (challenge), the card encrypts that number with a private key and send it back (response). The terminal can finally verify the authenticity by the use of a public key. Authentication can also be implemented by the use of digital signatures which are attached to a message. A popular example is the MAC (Message Authentication Code).

When retrieving public keys one cannot be sure that the public key belongs to the wanted communication partner. This is the task of a so-called certification authority or trust center which binds a key to an entity or agent. The widely used standard is given by X.509 which standardises a protocol and a signature with version, serial number, signature algorithm identifier, issuer name, validity period, subject name, and public key.

In literature and academia there are a couple of other additional approach under development like mobile cryptheographic, clueless agents, cryptographic tracing, and prove carrying code. We do not discuss them in this article since they are currently of minor practical importance.

3 FIPA

FIPA (Foundations of Physical Agents) is an international association for the promotion of agent-based application, services, and equipment. Meanwhile, FIPA has released a couple of standards known as FIPA97, FIPA98, and FIPA99. Standards cover the areas Agent management, communication, software integration, and a couple of reference applications.

FIPA97 defines the basic architecture of agent management, the ACL communication language, and some applications. Figure 1 gives an overview of the basic agent platform with the agent management system (AMS), directory facilitator (DF), agent communication channel (ACC), and the internal platform message transport.

![Figure 1: FIPA Agent Platform](image)

The AMS basically manages the creation, deletion, suspension, resumption, and migration of agents. Agents may register their services at the DF so that other agents can ask the DF for agents with specific services. The ACC provides a communication channel to other platforms.

The ACL is an ASCII-based inter-agent communication language composed of embedded object where objects are enclosed in braces and denoted by so-called parameters each of which is marked with a preceding colon, e.g., (message ... ).

For secure platforms the AMS may manage certification of agents based on public key pairs. For certification repository and encryption methods, parameters for DF registry entries are extended by agent-certificate, owner-certificate, and security-encapsulation-method.

![Figure 2: Secure FIPA Agent Platform](image)

It is recommended by FIPA to introduce an ad-
ditionally Agent Platform Security Manager (APSM) (cf. Figure 2) through which all incoming and outgoing intra-/inter-domain communication requests are managed. The APSM is in charge of all run-time security activities, like communication, transport-level security, and creation of audit trails. Audit trails are records of events of a particular agent.

When communicating, agents can send their message enclosed in a letter which has an envelope and a message part. The envelope part defines security properties for encrypted messages. These properties are given by keywords: confidentiality, integrity, authentication, and non-repudiation. Each of them can be either low, medium, or high. Alternatively, a specific security mechanism can be assign like DES-40, RC2, MAC, MD5, RSA, Kerberos.

Though, FIPA 95 part 10 introduces security issues it only gives rough guidelines and leaves important security details open or postpones them to future FIPA versions, respectively. Moreover, the mobility aspect is not covered at all by the present standard definitions [3, 4] though security should be of high importance for those applications.

Our approach which is presented in the next section extends the existing definition in order to complete it for secure agents and platforms also covering mobile agents.

4 Secure Mobile Agents

For mobile agents we basically have to distinguish three different attacking scenarios:

1. attacking the communication
2. agents attacking the platform
3. platform attacking the agent

For the first one we can distinguish a passive attack which just monitors a data stream and an active attack which manipulates the data. Data encryption basically prevents from passive attacks. Authentication, certification, and non-repudiation mechanisms are for active attacks.

Once running on a platform, agents may attack it in various ways. This heavily depends on the platform, if and how it provides a migration control and an access control to its resources. The most secure way is to run the agent in a sandbox and give it no access to any resource like it is usually done with Java applets, for instance.

Mobile agents can be attacked from the platform they are running on. The platform can (mainly actively) attack code, state, and data. The first attacks are for implanting a virus and for unwanted copying of programme code (like byte code) where the latter are for accessing collected data, for instance.

The next section introduces some extension to the existing FIPA standard. Thereafter, we introduce an FIPA application for document management in the context of security.

4.1 FIPA Extensions

We first introduce a flexible extension of the basic agent platform architecture. Thereafter, we present extensions to existing ontologies w.r.t. enhanced security.

Firstly, we introduce additional management objects which are in charge of managing subtask. This is in contrast to the existing FIPA strategy which tries to extend the scope of existing services like AMS, DF, and APSM instead. Nevertheless, we think that security issues should be separately managed since (i) security management largely varies for each platform and (ii) separated concepts easier allows to separate the implementation of security relevant management (in particular in combination with mobility) from non-security relevant agent management. The latter minimizes the possibility to break the system when entering through non-secure services.

Figure 3 gives our extended version of the agent management platform. We additionally introduce a domain manager (DM), a certification manager (CM), and a non-repudiation manager (NRM). This is only a conceptual separation into three agents. An implementation may combine these three to one agent, e.g., a security manager.

![Extended Secure FIPA Agent Platform](image)

Figure 3: Extended Secure FIPA Agent Platform

The DM is checking an agent before it wants to enter the platform. That means that it checks the agent’s policies for resource access, delegation, non-repudiation, and authorisation against the policies of the platform. In the case those are not compliant the agent is rejected to enter the platform. If accepted, the platform policies are communicated to the agent. In turn, the agent can now decide if the platform’s poli-
cies are acceptable. After registration, the monitoring of those policies is performed by the APSM. Since the APSM is also in charge of recording of audit trails it has to efficiently implement a sort of a intrusion detection system, e.g., by the use of state appraisal techniques.

The CM manages certificates and authorisation for the platform. The CM should have ability to manage certificates locally giving additional access to trusted external certificate authorities. Authorisation management has to be individually adopted can unfortunately not be generalized. It has to cover basic authorisation methods and algorithms, like the challenge-response protocol and the DSS.

The NRM basically provides non-repudiation proofs by generating tokens. In the case a sender S would like to get a proof from a receiver R that it has received a message m, S first sends m to the NRM to generate a token t₁ with a proof for m. Thereafter, S sends m to R. R requests another token t₂ for m from the NRM and sends it to S. Comparing t₁ and t₂ S can be sure that R has received the correct message. The NRM has to support a couple of proof algorithms and protocols for different types of proofs, e.g., data sent, data received. Again, the individual aggregation of the NRM heavily depends on the individual platform and application.

The above services require a couple of extensions w.r.t. to the existing FIPA ACL ontologies. For domain management two object definitions for agent-DM communication are needed: granted-rights-description and a required-rights-description, where parameters object-description, rights-family, rights, and interface-description have to be supported. For access management actions like grant-rights, revoke-rights, replace-rights, get-rights, set-required-rights, and get-required-rights need to be captured. Other policies provided by the DM require similar parameters and actions.

Due to our investigation it also makes sense to define an additional route ontology for agents. The route ontology covers the migration path of a particular agent over the past and future platforms as well as the security migration properties like signatures and public keys.

As introduced in the previous section, the APSM already covers a basic ontology for communicating integrity, confidentiality, authentication, and non-repudiation. However, the properties for audit managemt has to be worked out like event types and event (thresh-hold) values for audit trails.

The platform profile which is managed by the AMS has to be extended to additionally cover the description of possible (tamper-resistant) hardware restrictions, description of transport layer security policies like SSL, and a description of the execution environment.

On the other hand the agent’s profile (fipa-mobile-platform-profile) has to be extended to describe hardware requirements, the interfaces to required (external) agents, as well as its state space description. The latter defines the state space the agent is operating in. Leaving that state space indicates abnormal and unwanted behavior of that agent.

Finally, the DF needs little modification. The additional parameters non-repudiation-authority and :certification-authority describes for each agent which authorities are acceptable for that agent.

### 4.2 Secure Applications

Based on the previously presented security platform we now introduce an example of a secure document management system, i.e., an agent-based system where users exchange documents which migrate as agents through multiple agent platforms before it arrives at the destination platform.

In this system we decide to basically implement three sorts of agents:\footnote{The separation into 3 agent has been mainly chosen for demonstration purpose in order to clearly distinguish concepts. One can combine SA and and PSA, for instance, for individual implementation. This could be necessary for an efficient implementation on some platforms.}

- personal security agent (PSA)
- security agent (SA)
- document agent (DA)

![Secure FIPA Document Management System](image)

Figure 4: Secure FIPA Document Management System

Thereafter, the PSA creates an SA. The SA is configured for the individual needs of the PSA. This requires a specific level of data encryption, integrity, authentication, and non-repudiation. The PSA may identify its need by parameters low, medium, or high. The SA then keeps track for checking for appropriate services and algorithms by contacting the corresponding managers on
that platform. The SA basically acts as a proxy for the PSA w.r.t. security issues. It manages the different policies with the domain manager, identifies the communication partner by authentication, generates signatures, and manages the PSA’s personal certificate with the certificate manager CM.

After instantiation of the SA, the PSA generates a document agent DA. The DA has an initially empty document content component, an interface component, an access right component, and an encryption component. The document agent with given access rights now tries to retrieve content from two databases DB1 and DB2. The databases can only be accessed through so-called wrappers. Wrappers when being accessed for database requests check the access rights with the domain manager before retrieving the query. It is a design decision that the access to the database is additionally controlled by a wrapper and not only by the APSM since the DA only needs very limited read access to the database. The definition of this limited access through the APSM would have meant an overspecification in our example plus the implementation of unnecessary protocols between the SA, APSM, and DM.

After having retrieved the content from the databases the DA can migrate to another platform by contacting the DM from the platform via the ACC. After been accepted by the domain manager, the DA creates an SA, and registers. After that, DA is able to access resources on that platform and to migrate to a third one and/or to communicate with another PSA.

5 Implementation

We are currently investigating an integration of the presented concepts into the FIPA97-compliant MECCA framework [1]. However, since MECCA does not support security concepts so far, we have to explore the implementation of FIPA 98 part 10 security in a first step. It recently has turned out that this is by no means a straightforward extension in the implementation since the standard document [3] does not give very precise outlines how to implement an APSM and how to extend the AMS and DF. At present, it seems that the APSM should be better grouped as an additional (vertical) manager rather than to implement it as an additional transport layer.

6 Conclusion

We have presented extensions to FIPA 98 security management concepts with modifications to the agent platform architecture and the associated ontologies. The example of a document management system outlines an application which is based on these extensions.

In order to verify the presented concepts we have started a partial implementation of them. Due to the generic principles of FIPA and our concepts we are optimistic that the developments provide a sound basis for more or less complex distributed systems. However, covering only the most important security concepts and algorithms already ends up with a considerably complex implementation. The next step needs clearly a deep investigation of the associated protocols and some more applications.

References


Delayer Inconsistency Resolution Using Fuzzy Logic

Francesco Marcelloni
Dipartimento di Ingegneria della Informazione
University of Pisa
Via Diotisalvi, 2 –56126 Pisa, ITALY
E-mail: f.marcelloni@iet.unipi.it

Abstract

While developing complex systems, software engineers generally have to deal with various kinds of inconsistencies. Certain kinds of inconsistencies are inevitable, for instance, in case of multiple persons working independently of each other within the same project. Some inconsistencies are desirable when, for instance, alternative solutions exist for the same problem, and these solutions have to be preserved to allow further refinements along the development process. Current software development methods do not provide adequate means to model the desired inconsistencies and, therefore, aim to resolve the inconsistencies whenever they are detected. Although early resolution of inconsistencies reduces complexity of design by eliminating possible alternatives, it results in loss of information and excessive restriction of the design space. This paper aims to enhance the current methods by modelling and controlling the desired inconsistencies through the application of fuzzy logic.

Keywords: Inconsistency resolution, software development methods, fuzzy logic, adaptable design models, software artifacts.

1. Introduction

While developing complex software applications, software engineers have generally to deal with various kinds of inconsistencies that may originate from requirement specifications, involvement of multiple persons in the same project, errors in the software development process, alternative solutions, etc. [1]. Certain kinds of inconsistencies are inevitable, for instance, in case of descriptions from different persons involved in the requirement engineering process or in case of multiple software engineers working independently of each other within the same project. Some inconsistencies are desirable when, for instance, alternative solutions exist for the same problem. These solutions should be preserved to allow a software engineer to refine them along the development process and to take a decision when all relevant information is collected. Delaying inconsistency resolution, therefore, helps to retain the most appropriate solution to the specific application domain. Unfortunately, current methods do not introduce means to model desired inconsistencies and, therefore, tend to resolve them whenever they are detected.

Methods aim to create software products (or artifacts) through the application of a number of rules [2][3]. For example, the OMT method [3] introduces rules for identifying and discarding object-oriented artifacts such as classes, associations, and part-of and inheritance relations. Rules are derived from intuition of experienced software engineers. In object-oriented methods, for instance, a candidate class is identified by rule: If an entity in a requirement specification is relevant and can exist autonomously in the application domain then select it as a candidate class. While applying the object-oriented intuition of what a class should be, this rule enforces the consistency constraint “an entity is either a candidate class or not a candidate class, but not partially both”. Although a software engineer can perceive that the entity partially fulfils the relevance criterion and conclude that the entity is, for instance, substantially relevant, the consistency constraint forces her/him to take an abrupt decision, such as accepting or rejecting the entity as a candidate class. This results in loss of information because the information about the partial relevance of the entity is not modelled and therefore cannot be considered explicitly in the subsequent phases. Further, if the entity is not recognized as a candidate class, methodological rules that apply to candidate classes will not be fired. This implies that the path relative to candidate class will not be explored and possible re-evaluation of the entity as a class will not be possible, unless an explicit conversion will change an entity from a non-class to a class.

We believe that inconsistency resolution imposed by methodological rules is not necessary especially in the first phases of the development process. Enforcing consistency constraints is basically due to a lack of expressive ability of the methods themselves rather than to an intrinsic impossibility to cope with inconsistency. While identifying candidate classes, for instance, software engineers can perceive different grades of relevance, but heuristics of current methods do not allow them to express this gra-
In this paper, we propose fuzzy logic as a reasoning tool able to cope with desired inconsistencies. Fuzzy logic provides a sound framework to define a language, to associate a meaning with each expression of the language and to provide a means to compute these expressions. Software engineers can, therefore, describe their perception in their natural language and this perception can be maintained along the overall development process. Thus, an entity can be considered, for instance, as a weak candidate class and a substantial candidate attribute. Although class and attribute are two conflicting design alternatives for an entity, fuzzy logic allows managing this inconsistency. The linguistic expressions used to qualify the object-oriented concepts (weak and substantial in the previous example) can be considered as measures of each alternative. These measures prove to be particularly useful in selecting the best alternative in a set of possible conflicting design alternatives. It follows that fuzzy logic provides a tool not only to cope with inconsistencies, but also to resolve inconsistencies whenever it is required. Preserving inconsistencies as long as possible fits the principle of least commitment [4], according to which, when dealing with vague, uncertain or noisy domains, soft decision should be taken during all phases of the development process, because it may be difficult (or even impossible) to recover from a wrong crisp decision.

2. Enforcing consistency

2.1 Current software methods

Software methods are described in terms of artifacts, rules for identifying, defining and transforming artifacts, and a notation to represent artifacts. To identify or eliminate an artifact, and relate an artifact to other artifacts, methods provide rules. In most methods, rules are defined informally using a natural language (see, for instance, [2][3]). Each rule codifies the intuition with which an experienced software engineer identifies an artifact. For instance, in OMT [3] candidate classes are identified by the following rule Candidate Class Identification:

\[
\text{IF AN ENTITY IN A REQUIREMENT SPECIFICATION IS RELEVANT AND CAN EXIST AUTONOMOUSLY IN THE APPLICATION DOMAIN THEN SELECT IT AS A CANDIDATE CLASS.}
\]

Here, an entity in a requirement specification and a candidate class are the two object-oriented artifact types to be reasoned. If the antecedent of the rule is true, then the result of this rule is the classification of an entity in a requirement specification as a candidate class. The rule enforces the consistency constraint imposed by the artifact type Entity: “an entity is either a candidate class or not”.

Once an artifact has been classified, for instance into the rejected set of a rule, it is not considered anymore by the rules that apply to the accepted set of that rule. Of course, a rejected entity can be considered by another rule, which applies to the entities in a requirement specification. Consider, for example, the rule Candidate Attribute Identification:

\[
\text{IF AN ENTITY IN A REQUIREMENT SPECIFICATION IS RELEVANT AND CANNOT EXIST AUTONOMOUSLY IN THE APPLICATION DOMAIN, THEN IDENTIFY IT AS A CANDIDATE ATTRIBUTE.}
\]

This rule can be applied to the entities in a requirement specification, which are rejected by the rule Candidate Class Identification. Here, the rule follows the consistency constraint imposed by the artifact type Entity: “an entity is either an attribute or not”. Further, it is obvious that if an entity is a class, it cannot be an attribute or vice versa. If all the rules, which are applicable to an entity in a requirement specification, reject that entity, then the entity is practically discarded.

After identifying candidate classes, redundant candidate classes can be eliminated for instance by using rule Candidate Class Elimination:

\[
\text{IF TWO CANDIDATE CLASSES EXPRESS THE SAME INFORMATION THEN DISCARD THE LEAST DESCRIPTIVE ONE.}
\]

An artifact can be converted from an artifact type to another artifact type. For instance, a candidate class can be converted to a candidate attribute applying the rule Candidate Class to Candidate Attribute Conversion:

\[
\text{IF A CANDIDATE CLASS QUALIFIES ANOTHER CLASS THEN IDENTIFY IT AS A CANDIDATE ATTRIBUTE OF THAT CLASS.}
\]

In case candidate classes and attributes are not eliminated or converted, they are selected as classes and attributes, respectively.

2.2 Early resolution of inconsistency

Methodological rules of current methods enforce consistency at each stage of the development process. This reduces the complexity of the overall process, but may dramatically affect the quality of final product. Although software development is an uncertain domain, and soft decisions should be taken during all phases of the development process, each methodological rule forces the software engineer to perform abrupt classifications. In general, consistency constraints implicitly imposed by the application of a rule quantize a set of object-oriented artifacts into two subsets: accepted or rejected. If on the one side this two-level quantization process reduces the complexity of the design process, on the other side it generates the so-called quantization error.

Quantization error has been extensively studied in the field of digital signal processing. Here, quantization proc-
ess consists of assigning the amplitudes of a sampled analog signal to a prescribed number of discrete quantization levels. Quantization error is defined as the difference between an analog and the corresponding quantized signal sample. If the amplitude distribution of the signal is known, the value of quantization error can be computed [5]. In general, the less the number of quantization levels, the higher the quantization error.

In current methods, consistency constraints imposed in methodological rules generate a quantization process with only two quantization levels. For example, the rule Candidate Class Identification requires from the software engineer to decide whether an entity in a requirement specification is relevant or not. The software engineer may, however, perceive that an entity partially fulfils the relevance criterion, and may conclude that the entity is substantially relevant. Here, the quantization error is the difference between the perception of the software engineer and the “quantization levels” imposed by the methodological rules. If the distribution of the input values of rules is known, formulas adopted in digital signal processing can be used to determine quantization error, which affects software development process [6] [7].

It can be proved that the result produced by the execution of a rule with multiple conditions in the antecedent part is affected by a quantization error higher than the one computed for each single input [6]. Considering that methodological rules are chained, that is, the result of a rule is the input to another rule, the quantization error can assume intolerable values especially in the last stages of the development process. We point out that the value of quantization error does not indicate that the resulting object-oriented model will have the same percentage of error. The measurement of error in the resulting object model requires detailed semantic analysis of the requirement specification and the object model. It is however expected that high loss of information will be likely to cause errors in the resulting object-oriented model.

One of the dramatic effects of the quantization error on the development process is early elimination of artifacts. Each decision taken by a rule is based on the available information up to that phase. For the early phases, there may not be sufficient amount of information available to take abrupt decisions like discarding an entity. Such an abrupt decision must be taken only if there is sufficient evidence that the entity is indeed irrelevant. In most object-oriented methods, however, each identification process is followed by an elimination process. For example, the OMT method proposes a process that includes class identification and elimination, association identification and elimination, and so on. Now, assume that a software engineer discards an entity because it is considered non-relevant. The discarded entity, however, could have been included as a candidate class, if the software engineer had gathered more information about its structure and operations.

The application of subsequent rules and the consequent acquisition of new property values could revalue the entity as class. In current methods, this revaluation process is not possible because a non-relevant entity is discarded and therefore is not considered anymore in the development process. Here, property Relevance, which is only a partial view of artifact types as Class and Attribute, determines the membership of the entity to these artifact types and consequently the possible elimination of the entity. In current methods, therefore, the application of a methodological rule can cause the elimination of an artifact (see, for instance, rule Candidate Class Elimination), or the conversion of an artifact from an artifact type to another. For example, a class may be converted into an attribute by rule Class to Attribute Conversion:

IF A CLASS IS NOT RESPONSIBLE FOR THE REALIZATION OF ANY FUNCTION THEN CONVERT THE CLASS TO AN ATTRIBUTE.

If, at the end of the development process, the software engineer realizes that the resulting object model is not satisfactory, there are two possible options: improving the model by applying subsequent rules and/or by iterating the process. The application of subsequent rules may not adequately improve the model because of the loss of information due to quantization errors. The iteration of the process still suffers from the quantization error problem. Moreover, managing iteration remains as a difficult task.

3. Improving current methods

To improve current methods, desired inconsistencies should be preserved and resolved whenever it is actually necessary. Such an inconsistency resolution, for instance, may be requested by the language compiler. A demand for resolving an inconsistency therefore may be context or language dependent. For example, the C++ language allows multiple inheritance specification whereas the Smalltalk language forbids it.

Delaying inconsistency resolution, however, does not guarantee to improve current methods unless a measure is associated with each conflicting alternative and this measure is updated whenever new relevant information is collected. Consider for example that an entity may be classified as an attribute and a class at the same time. To be able to reason about the alternatives, there is a need to give a measure for each alternative. The software engineer, for instance, may classify an entity more like a class than an attribute and give a higher measure to it.

Current methods do not provide a means to deal with multiple design alternatives and to measure the quality of each alternative during the development process. For conflicting artifact types, whenever a property value is collected, current methods propose rules to convert artifacts from the one to the other of the conflicting artifact types. Rule Class to Attribute Conversion is an example of this
conversion.

Finally, deferring consistency enforcement decreases the loss of information but increases the complexity of design. There is a need for introducing appropriate techniques to manage this increased complexity without necessarily giving up the design flexibility. In particular, the trade-off between flexibility and complexity should be controlled by the software engineer.

4. Using fuzzy logic in modelling inconsistencies

To cope with desired inconsistencies along the overall development process, a new expressive form rather than two-valued logic used in the current methodological rules has to be investigated. Such a form has to be able to capture as much as possible the method developers’ intuition and software engineer’s perception of an artifact type. Further, it has to be able to manage the increased complexity derived from the concurrent exploration of conflicting alternatives. To this aim, two features are considered basic: i) similarity to the natural language typically used by the software engineer and ii) capability to reason on the linguistic expressions to deduce conclusions and conduct the development process. Fuzzy logic looks to be the ideal solution.

As L. Zadeh claims in [8], one of the main contributions of fuzzy logic is computing with words. Fuzzy logic provides a sound framework to define a language, to associate a meaning with each expression of the language and to compute these expressions. Basic in fuzzy logic is the concept of linguistic variable: A linguistic variable is a variable whose values, called linguistic values, have the form of phrases or sentences in a natural language [9]. Each linguistic value is associated with a fuzzy set that represents its meaning. A fuzzy set $S$ of a universe of discourse $U$ is characterized by a membership function $\mu_S : U \rightarrow [0,1]$ which associates with each element $y$ of $U$ a number $\mu_S(y)$ in the interval $[0,1]$ which represents the grade of membership of $y$ in $S$. Relations between linguistic variables are defined by means of fuzzy rules, which are typically expressed as: IF $X$ IS $A$ THEN $Y$ IS $B$, where $X$ and $Y$ are linguistic variables and $A$ and $B$ are linguistic values.

In contrast with two-valued logic, the propositions $X$ IS $A$ and $Y$ IS $B$ may return values between 0 and 1 rather than Boolean values. Given a fact and a rule, one of the most known fuzzy inference tools, the generalized modus ponens, allows deducing a fuzzy conclusion (see [10] for more details). Notice that the generalized modus ponens allows to infer conclusions also if the facts, in this case $X$ IS $A'$, match only approximately the antecedent part of the rule.

A conclusion is expressed as a linguistic value. If we are interested in a crisp value, we must defuzzify the conclusion by a defuzzification strategy [9]. A defuzzification strategy is aimed at producing the crisp value which best represents the linguistic value. At present, the commonly used strategies may be described as the mean of maxima and the center of area. The crisp value produced by the mean of maxima strategy represents the mean value of the elements, which belong to the fuzzy set characterizing the conclusion with maximum grade. The center of area strategy produces the center of gravity of the fuzzy set characterizing the conclusion.

4.1 Modeling artifacts

Denote each artifact type as $[A, (P_1, D_1), (P_2, D_2), \ldots, (P_n, D_n)]$ where $A$ is the artifact type, $P_i$ is a property of $A$ and $D_i$ is the definition domain of $P_i$. An example of an artifact type is $[Entity, (Relevance, (Relevant, Non-Relevant)), (Autonomy, (Autonomous, Non-Autonomous))]$. Here, Relevant and Non-Relevant, and Autonomous and Non-Autonomous are respectively the pairs of values that Relevance and Autonomy can assume in current methodological rules. A software artifact is an instantiation of an artifact type and can be expressed as $[A, id, (P_1: V_1), (P_2: V_2), \ldots, (P_n: V_n)]$, where $A$ is the artifact type, $id$ is the unique identifier of the artifact, and $V_i$ is a value defined in domain $D_i$ of property $P_i$. Artifacts can be also named.

In the following example, Name is the name of the artifact:

Name-- [Entity, id, (Relevance: Relevant), (Autonomy: Autonomous)]

As current methods strengthen consistency at each stage of the development process, the definition domain of properties is suited to meet consistency constraints. For instance, the possible values for property Relevance are only Relevant and Non-relevant. Anyway, when method developers define methodological rules, they intuit that entities can be partially relevant and a partially relevant entity should be selected as a partial member of artifact type Candidate Class. Nevertheless, consistency constraints force method engineers to quantify their intuition of partial relevance in such a way as to create a sharp boundary between instances and non-instances of an artifact type. Then, enforcing consistency at each stage of the development process generates a semantic gap between method developers’ intuition of an artifact type and actual representation of this intuition by means of methodological rules. Also, definition domains established by method developers do not allow software engineers to express completely their perception of an artifact. Although software engineers can perceive, for instance, different grades of relevance of an entity, they are required to quantize their perception in order to match input values imposed by rules. There exists a semantic gap between the software engineers’ perception and the input required by the rule.
We considered each property as a linguistic variable and investigated which values and meanings can be significant to a software engineer. For instance, we verified that the property \textit{Relevance} of the artifact \textit{Entity} can be expressed as \textit{weakly}, \textit{slightly}, \textit{fairly}, \textit{substantially} and \textit{strongly} relevant and the property \textit{Autonomy} as dependent, partially dependent and autonomous.

The meaning of the linguistic values of \textit{Relevance} and \textit{Autonomy} are shown in Figures 1 and 2. Here, the X and Y axes indicate the universe of discourse and the membership values, respectively. The universes are supposed to vary from 0 to 1. Consequently, the artifact type \textit{Entity} can be expressed as:

\begin{equation}
\text{[Entity, (Relevance, \{Weakly, Slightly, Fairly, Substantially, Strongly\}), (Autonomy, \{Dependent, Partially Dependent, Autonomous\})]}
\end{equation}

Fitting the software engineer’s perception increases the number of quantization levels and therefore decreases the quantization error [11]. Methodological rules are also expressed using fuzzy logic. Consider, for example, the modified rule \textit{Candidate Class Identification}:

\begin{center}
\textbf{IF AN ENTITY IN A REQUIREMENT SPECIFICATION IS RELEVANCE VALUE RELEVANT AND CAN EXIST AUTONOMY VALUE AUTONOMOUS IN THE APPLICATION DOMAIN, THEN SELECT IT AS A RELEVANCE VALUE RELEVANT CANDIDATE CLASS.}
\end{center}

Here, an entity and a candidate class are the concepts to be reasoned, \textit{Relevance} and \textit{Autonomy} are the properties, and relevance value and autonomy value indicate the domains of these properties.

The rule \textit{Candidate Class Identification} can be represented in the following way:

\begin{equation}
P \leftarrow \text{[Entity, id, (Relevance: } V_1 \in \text{\{Weakly, Slightly, Fairly, Substantially, Strongly\}}, (Autonomy: } V_2 \in \text{\{Dependent, Partially Dependent, Autonomous\})]} \Rightarrow \text{P} \leftarrow \text{[CandidateClass, id, (Relevance: } V_3 \in \text{\{Weakly, Slightly, Fairly, Substantially, Strongly\})]}
\end{equation}

Here, \(P\) and symbol \(\Rightarrow\) indicate a generic artifact name and the fuzzy implication operator, respectively. Each combination of relevance and autonomy values of an entity has to be mapped into one of the five candidate class relevance values. The resulting 15 sub-rules are shown in Table 1. Each element of the table, shown in italics, represents the output value of a sub-rule, which is the relevance value of the candidate class being considered. For example, if the relevance and autonomy values are respectively \textit{Strongly} and \textit{Autonomous}, then the candidate class relevance value is \textit{Strongly}. We selected these output values based on our intuition and knowledge on object-oriented methods [12] [13] [14].

If fuzzy logic-based methodological rules are applied, no artifacts are theoretically eliminated: each artifact belongs to an artifact type with different membership degrees. Alternative design solutions can therefore be investigated concurrently. Software engineers are not forced to take abrupt decisions, but are encouraged to express their perception of artifacts in their natural language. This reduces the loss of information and increases the quality of the whole development process. Further, perceptions expressed as linguistic expressions can be used as measures of alternatives and exploited to resolve inconsistencies whenever it is necessary. The complexity of this concurrent analysis of multiple alternative solutions requires an appropriately designed CASE environment.

**Table 1. Sub-rules of the rule Candidate Class Identification.**

<table>
<thead>
<tr>
<th>P ← CandidateClass, Relevance:</th>
<th>Dependent</th>
<th>Partially Dependent</th>
<th>Autonomous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weakly</td>
<td>Weakly</td>
<td>Weakly</td>
<td>Weakly</td>
</tr>
<tr>
<td>Slightly</td>
<td>Weakly</td>
<td>Slightly</td>
<td>Slightly</td>
</tr>
<tr>
<td>Fairly</td>
<td>Weakly</td>
<td>Slightly</td>
<td>Fairly</td>
</tr>
<tr>
<td>Substantially</td>
<td>Weakly</td>
<td>Fairly</td>
<td>Substantially</td>
</tr>
<tr>
<td>Strongly</td>
<td>Slightly</td>
<td>Fairly</td>
<td>Strongly</td>
</tr>
</tbody>
</table>

\begin{center}
\text{P} \leftarrow \text{Entity, Autonomy:}
\end{center}

\begin{center}
\text{P} \leftarrow \text{Entity, Relevance:}
\end{center}
To this aim, we extended the Rational Rose™ environment [15]. We built tools to support method engineers in defining fuzzy rules. Fuzzy reasoning was implemented as an object-oriented framework [12]. Thus, different implementations of fuzzy inference can be experimented and relatively compared. The development process is guided by the fuzzy methodological rules. When an input value is requested, the tool shows the definition of the linguistic values so that the software engineer can relatively compare the possible input values.

4.2 Inconsistency Resolution

In our case tool, each artifact type defines when it is inconsistent with other artifact types and how this inconsistency can be resolved. For instance, artifact type Class knows that it can be inconsistent with artifact type Attribute: this occurs when an entity is classified both as a class and as an attribute. When a class, say A, has to be committed, artifact type Class checks if an instance of artifact type Attribute is derived from the same entity from which class A is derived. If it is the case, an inconsistency is detected and an inconsistency resolution policy is activated. Policies depend on the inconsistency type and may be affected by contextual factors such as application type, sensibility and experience of the software engineer and desired level of quality.

An inconsistency resolution policy is typically implemented as comparing defuzzified linguistic values and selecting the artifact corresponding to the highest value. Optionally, before the selection is made the software engineer may be consulted. In our CASE tool, this consulting is activated if the compared defuzzified values are close to each other. Let us suppose that the fuzzy logic-based method has selected both inheritance and aggregation relations between two classes with similar relevance degree. If conceptual modeling is considered important, then the software engineer may decide to select the aggregation relation. If reusability is the main concern, then the inheritance relation can be the choice. In this case, the goal of the application and the experience of the software engineer affect the resolution policy.

Resolution of inconsistencies may generate new possible inconsistencies. Let us suppose that an entity A was considered as a fairly relevant class and a substantially relevant attribute. As a fairly relevant class, possible inheritance and association relations with other classes were analyzed. Let us assume that the adopted resolution policy is to select the artifact corresponding to the highest value. Entity A is, therefore, selected as an attribute. This implies that all the inheritance and association relations including A are not consistent anymore. The resolution of the class-attribute inconsistency can therefore generate other inconsistencies, which require being resolved. In general, an inconsistency resolution may activate a chain of inconsistency resolutions. At the present, we are investigating the possible effects raised by these chains.

4.3 Advantages of adopting fuzzy logic-based methodological rules

Using fuzzy logic to implement methodological rules allows increasing the number of quantization levels. It follows that quantization error may be considerably reduced [6]. Further, in the fuzzy logic-based method, the accumulation of the quantization error during the software development process is much less than the accumulation of error in the current methods. The improvement is achieved by capturing as much as possible the software engineer’s perception.

Current methods force inconsistency resolution during the application of each rule. It follows that the resulting object model is less adaptable to changes and will not adapt itself to the new information available during the software development process. Fuzzy logic-based method allows living with inconsistencies and, therefore, in principle, eliminates none of the artifacts. The fuzzy logic-based method can be considered as a learning process: a new aspect of the problem being considered is learned after the application of each rule. Obviously, a new aspect can modify the previously gathered property values. Software development through learning creates very adaptable and reusable design models.

Further, the increased expressive ability provided by fuzzy logic allows method developers to highlight peculiarities, which cannot be represented by two-valued logic used in current methods. Let us consider, for instance, rule Class to Attribute Conversion as defined in Section 2. Here, only the intuition about lack of responsibility is captured. In the reality, method developers also reason on the presence of responsibility. It is evident that the more a class is responsible for at least a function, the more this class is an instance of artifact type Class. This intuition can easily be modeled in fuzzy logic by fuzzy version of rule Class to Attribute Conversion.

\[
\text{IF } P \text{ IS A RELEVANCE VALUE RELEVANT CLASS AND } P \text{ IS RESPONSIBILITY VALUE RESPONSIBLE FOR THE REALIZATION OF FUNCTIONS THEN } P \text{ IS A RELEVANCE VALUE RELEVANT CLASS AND } P \text{ IS A RELEVANCE VALUE RELEVANT ATTRIBUTE.}
\]

Table II shows the definition of the rule. Each element of the table represents the output value of Relevance for artifact types Class and Attribute.

Though on the one side the increasing of the number of quantization levels improves the quality of the software development process, on the other side this enlarges considerably the design space. To maintain the development process manageable, our CASE tool allows a software engineer to establish a threshold for each property of an artifact type. When the relevance value of an artifact is below the threshold, the artifact is discarded and not considered anymore in the development process.
Further, various automatic control mechanisms can be established to manage the complexity. For example, in case of inconsistent alternatives, the software engineer may be guided to work on the alternative with the highest measure. If this measure decreases in the subsequent phases, other alternatives can be automatically brought to the attention of the software engineer. Our experience with the experimental CASE environment shows that being able to process rules and inconsistencies automatically hides the internal complexity.

5. Related Work

Some methods have been proposed in last years to defer elimination of alternatives. The Demeter system, for instance, defers certain design decisions so that software may be adapted to the changing context [16]. For example, functionality of software can be developed independently of the class structure. Only when the executable software has to be generated, the operations are allocated with the appropriate classes. In [17], inconsistent data are automatically marked by means of pollution markers. A pollution marker makes the inconsistent data known to procedures or human agents, which are responsible for solving the inconsistency. Further, it protects the inconsistent data from the action of other procedures sensitive to the inconsistency.

In [1], inconsistency handling in multi-perspective specifications is studied by using the ViewPoint framework [18]. In this framework, each developer specifies the system by using a representation language and a development process according to his/her own viewpoint. The consistency rules are expressed in terms of classical logic and represent some of the implicit assumptions and integrity constraints used in controlling and coordinating a set of viewpoints. A meta-language based on linear-time temporal logic is used to specify the actions necessary to cope with inconsistency.

In [19] [20], the approach of inconsistency handling shown in [1] is further developed by introducing quasi-classical logic. On the contrary of classical logic, quasi-classical logic allows the derivation of non-trivial inferences from inconsistent information. In the presence of inconsistencies, this allows limited reasoning and consequently the possibility of analyzing such inconsistencies. The analysis may identify the sources of inconsistency and may also qualify the inconsistent information.

In [21], inconsistencies, which occur between definition and actual instance of a development process, have been studied in human-centered systems. It is argued that processes defining the interaction between humans and computerized tools have to tolerate, control and support inconsistencies and deviations of real-world behaviors with respect to the process model. This is necessary to maintain an effective flexibility and adaptability to the evolving needs and preferences of the humans. They propose a framework for formally defining the concepts of inconsistency and deviation between a human-centered system and its process support system. Deviations are tolerated as long as they do not affect the correctness of the system. Then, a reconciling sequence of feasible events starting from an inconsistent state and terminating in a consistent state has to be executed.

Our work is similar to the related work presented in this section in that tolerating and coping with inconsistency are considered important in creating flexible software systems. This paper analyses how inconsistencies are eliminated in current software development methods. Further, inconsistencies are defined as artifact properties within our CASE environment. Finally, to model and

---

**TABLE II**

**SUB-RULES OF RULE CLASS TO ATTRIBUTE CONVERSION.**

<table>
<thead>
<tr>
<th>Class, Attribute: Membership</th>
<th>Weakly</th>
<th>Slightly</th>
<th>Fairly</th>
<th>Substantially</th>
<th>Strongly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weakly</td>
<td>Weakly</td>
<td>Slightly</td>
<td>Weakly</td>
<td>Substantially</td>
<td>Strongly</td>
</tr>
<tr>
<td>Slightly</td>
<td>Weakly</td>
<td>Slightly</td>
<td>Fairly</td>
<td>Slightly</td>
<td>Substantially</td>
</tr>
<tr>
<td>Fairly</td>
<td>Slightly</td>
<td>Slightly</td>
<td>Fairly</td>
<td>Slightly</td>
<td>Fairly</td>
</tr>
<tr>
<td>Substantially</td>
<td>Fairly</td>
<td>Slightly</td>
<td>Substantially</td>
<td>Substantially</td>
<td>Substantially</td>
</tr>
<tr>
<td>Strongly</td>
<td>Substantially</td>
<td>Weakly</td>
<td>Strongly</td>
<td>Strongly</td>
<td>Strongly</td>
</tr>
</tbody>
</table>
manage inconsistencies, fuzzy logic-based software development rules are proposed.

6. Conclusions

Most of the current software development methods consider inconsistencies undesirable and tend to resolve them whenever they are detected. Some types of inconsistencies, however, should be maintained as long as possible. For example, when multiple solutions exist for the same problem and these solutions should be examined concurrently to determine which solution is actually the most effective.

In this paper, we have proposed fuzzy logic to cope with desirable inconsistencies during the software development process.

We have shown that fuzzy logic can model inconsistencies effectively without altering the intuitive expressiveness of the current methods. Further, fuzzy logic allows capturing software engineer’s perception in a natural way exploiting linguistic expressions such as an entity is substantially relevant. Finally, the meaning associated with the linguistic expressions represents a measure of each alternative. For instance, a software engineer can identify an entity as a substantially relevant class and a slightly relevant attribute. The meaning of substantially and slightly is a measure associated with the two conflicting alternatives class and attribute, respectively. Such measures are useful in providing a software engineer with a means to control the complexity of the development process, and in defining policies for resolving inconsistencies whenever it is requested.

A small fuzzy logic-based method has been implemented using our experimental CASE environment and tested on an example problem. We have observed that the resulting object model is more adaptable than the one developed by using standard methods. We are currently developing a fuzzy logic based-method derived from the heuristics of the most popular object-oriented methods such as OMT.

Acknowledgements

The author thanks Prof. M. Aksit for valuable discussion on the approach proposed in this paper.

References