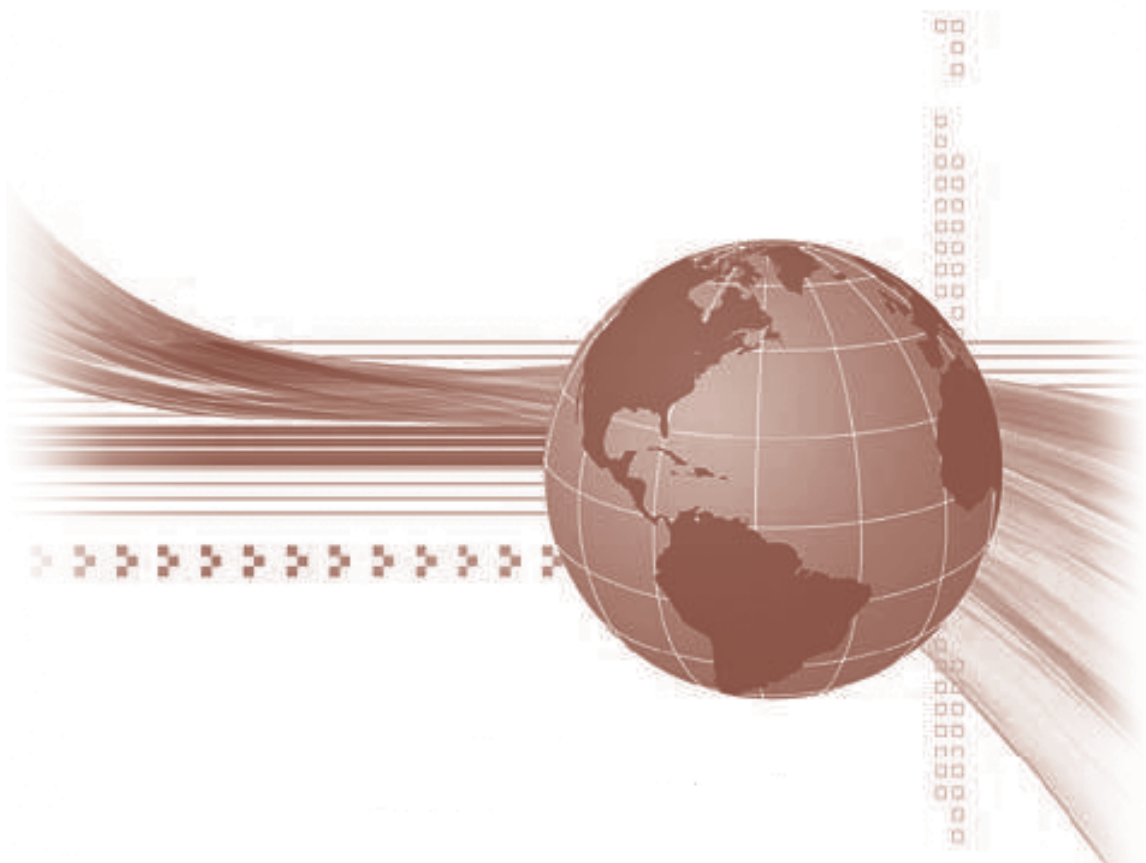
STUDIA UNIVERSITATIS
BABEŞ-BOLYAI



INFORMATICA

2/2012

YEAR **Volume 57 (LVII) 2012**
MONTH **JUNE**
ISSUE **2**

S T U D I A
UNIVERSITATIS BABEŞ-BOLYAI
INFORMATICA

2

EDITORIAL OFFICE: M. Kogălniceanu 1 • 400084 Cluj-Napoca • Tel: 0264.405300

SUMAR – CONTENTS – SOMMAIRE

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

A FUZZY-MEMBRANE-IMMUNE ALGORITHM FOR BREAST CANCER DIAGNOSIS

EMAD NABIL [1], AMR BADR [2], AND IBRAHIM FARAG [2]

Abstract. The automatic diagnosis of breast cancer is an important medical problem. This paper hybridizes metaphors from cells membranes and intercommunication between compartments with clonal selection principle together with fuzzy logic to produce a fuzzy rule system in order to be used in diagnosis. The fuzzy-membrane-immune algorithm suggested were implemented and tested on the Wisconsin breast cancer diagnosis (**WBCD**) problem. The developed solution scheme is compared with five previous works based on neural networks and genetic algorithms. The algorithm surpasses all of them. There are two motivations for using fuzzy rules with the membrane-immune algorithm in the underline problem. The first is attaining high classification performance. The second is the possibility of attributing a confidence measure (degree of benignity or malignancy) to the output diagnosis, beside the simplicity of the diagnosis system, which means that the system is human interpretable.

1. Introduction

P Systems are used to search large, and often complex or exponential search spaces. They have proven worthwhile on numerous diverse problems, able to find optimal solutions, of course by creating exponential search space, in polynomial or even linear time. Fuzzy modeling can be considered as an optimization process where part or all of the parameters of a fuzzy system constitute the search space [6]. Works investigating the application of approximate techniques in the domain of fuzzy modeling had first appeared about a decade ago [18]. These focused mainly on the tuning of fuzzy inference

Received by the editors: October 12, 2011.

2010 *Mathematics Subject Classification.* 03E72, 68T20, 92D25, 92F05.

1998 *CR Categories and Descriptors.* I.2.8 [**Computing Methodologies**]: artificial intelligence – *Problem Solving, Control Methods, and Search*; I.1.2 [**Computing Methodologies**]: Symbolic and algebraic manipulation – *Algorithms*; I.5.1 [**Pattern Recognition**]:Models – *Fuzzy set.*

Key words and phrases. membrane computing, P systems, artificial immune system, clonal selection algorithm, breast cancer diagnosis, fuzzy logic.

systems involved in control tasks (e.g. cart-pole balancing, liquid level system, and spacecraft rendezvous operation)[2]. Approximate fuzzy modeling has been applied to number of domains, as chemistry, medicine, telecommunications, biology, and geophysics.

This paper uses the power of P systems with the clonal selection principle [14] to simulate solving the breast cancer diagnosis problem. The solution scheme depends on generating a fuzzy rule system for diagnosis. The fuzzy rule system itself could be found in a large or an exponential search space. We import from membrane computing The Intercellular communication between cells as depicted in figure 1. The proposed algorithm imports the positive/negative selection and proliferation from clonal selection principle.

This paper is organized as follows: section 2 gives an entry to P systems, section 3 gives an introduction to the problem to be solved, namely the Wisconsin breast cancer diagnosis problem; section 4 presents the used solution scheme. The section presents the fuzzy-membrane-immune algorithm and how to evolve a fuzzy system for the WBCD problem, after that the fuzzy-membrane-immune algorithm setup, in section 5 experimental results and testing are explained. Finally conclusions and future work are discussed in section 6.

2. An Entry to P systems

Membrane Computing is a branch of natural computing; which was introduced by Paun in [5] under the assumption that the processes taking place in the compartmental structure of a living cell can be interpreted as computations. The devices of this model are called P systems. A P system consists of a membrane structure, in the compartments of which one places multi-sets of objects which evolve according to given rules in a synchronous, non-deterministic maximally parallel manner.

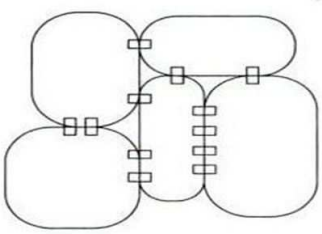


Figure 1. Intercellular communication

An important part of the cell activity is related to the passage of substances through membranes, and one of the most interesting ways to handle this trans-membrane communication is by coupling molecules. The process by which two molecules pass together across a membrane (through a specific protein channel) is called symport, see figure 2(a). When two molecules pass simultaneously through a protein channel in opposite directions is called antiport, see figure 2(b).

The symport and antiport operations could be formalized in an obvious way: $(ab,\ in)$ or $(ab,\ out)$, in symport rules, stating that a and b pass together through a membrane, entering in the former case and exiting in the latter case; similarly, $(a,\ out;\ b,\ in)$ is an antiport rule, stating that a exits and, at the same time, b enters the membrane. Separately, neither a nor b can cross a membrane unless we have a rule of the form $(a,\ in)$ or $(a,\ out)$, called, for uniformity, the uniport rule. The used communication method in the fuzzy-membrane-immune algorithm is antiport.

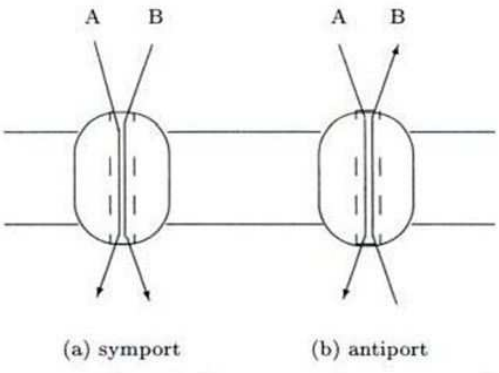


FIGURE 2. symport and antiport cellular communication

Beside the symport/antiport intercellular communication imported from membranes, the dividing and dissolution of membranes is be controlled using another nature inspired computation model, namely, the clonal selection principle, which is part of the immune system.

3. The Wisconsin breast cancer diagnosis problem

The Wisconsin breast cancer diagnosis problem [1, 2] is the test case of our proposed algorithm. Breast cancer is the most common cancer among women,

TABLE 1. The attributes in the Wisconsin data base

Clump thickness	V_1
Uniformity of cell size	V_2
Uniformity of cell shape	V_3
Marginal adhesion	V_4
Single epithelial cell size	V_5
Bare nuclei	V_6
Bland chromatin	V_7
Normal nucleoli	V_8
Mitosis	V_9

TABLE 2. The WCBD data representation

case	V_1	V_2	...	V_9	Diagnosis
1	1	2	...	8	Benign
2	2	4	...	3	Benign
...	...	...	...	...	...
683	4	8	...	1	Malignant

excluding skin cancer. The presence of a breast mass is an alert sign of a cancer, but it does not always indicate a malignant one. Fine Needle Aspiration (**FNA**) is an outpatient procedure that involves using a small-gauge needle to extract fluid directly from a breast mass. **FNA** procedure over breast masses is a cost-effective, non-traumatic, and mostly non-invasive diagnostic test that obtains information needed to evaluate malignancy. The Wisconsin Breast Cancer Diagnosis (**WBCD**) database [3] is the result of the efforts made at the university of Wisconsin hospital for accurately diagnosing breast masses based solely on a **FNA** test. Nine visually assessed characteristics of an **FNA** sample considered relevant for diagnosis were identified, and assigned an integer value between 1 and 10. The measured variables are described in table1.

The database itself consists of 683 cases. The general form of the database is described in Table 2. There exist some previous systems that achieved high classification ration, but these systems look like black boxes and with no explanation or interpretation about how the decision was taken. Further, the degree of benignity or malignancy is not provided. These two points are covered in this study besides a high classification ratio. In the next section, an entry about fuzzy modeling is given, as an entry to our problem solution scheme.

4. The solution scheme

The solution scheme we propose for the **WBCD** problem is depicted in Figure 3; Note that the fuzzy subsystem displayed to the left of figure 3 is the fuzzy inference system of Figure 4. Figure 3 consists of a fuzzy system and a threshold unit. The fuzzy system computes a malignancy value of the malignancy of a case, based on the input values, the threshold unit then outputs a benign or malignant diagnostic according to the fuzzy system's output. If the malignancy value is less than or equals 3, it is considered a benign case. Other than that, it is diagnosed as a malignant one. Look at figure 5.

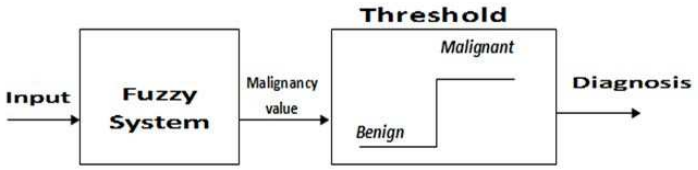


Figure 3. The proposed diagnosis system; the fuzzy subsystem displayed to the left is the fuzzy inference system in Figure 4

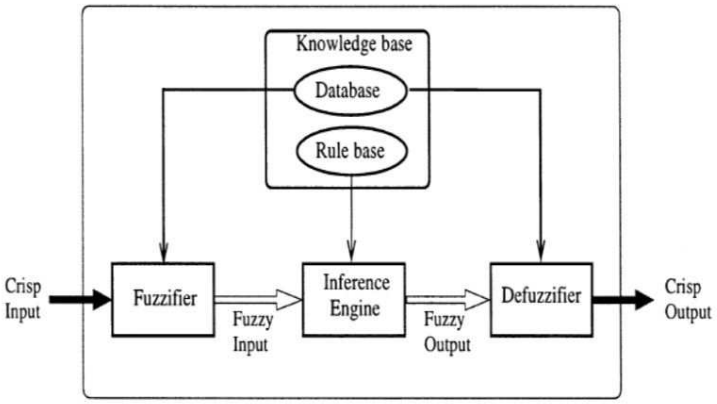


Figure 4. Basic structure of a fuzzy inference system

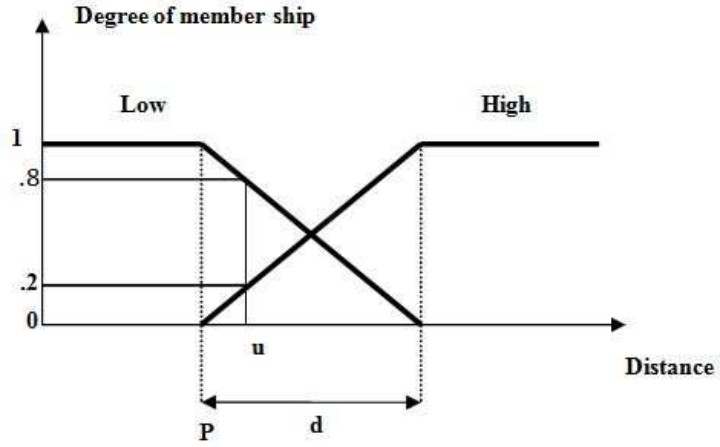


FIGURE 5. Example of a fuzzy variable length which has two possible fuzzy values, labeled low and high, and orthogonal membership functions, plotted above as degree of membership versus input values. P and d define the start point and the length of membership function respectively. The orthogonality condition means that the sum of all membership functions at any point is one. In the figure, an example: value u is assigned the membership values $\mu_{low}(u) = 0.8$ and $\mu_{high}(u) = 0.2$ (as it can be seen $\mu_{low}(u) + \mu_{high}(u) = 1$).

According to information obtained from previous work [1, 2], we have deduced the following knowledge: Systems with no more than four rules have been shown to obtain high performance. Rules with no more than four antecedents have proven adequate. Higher-valued variables are associated with malignancy [16, 17]. Some fuzzy models forgo interpretability in the interest of improved performance. Where medical diagnosis is concerned, interpretability, also called linguistic integrity, is the major advantage of fuzzy systems. This motivated us to take into account the following semantic criteria, defining constraints on the fuzzy parameters [8]:

- *Distinguishability*: To what extent the system is understood and has interpretability
- *Justifiable number of elements*: The number of membership functions of a variable. This number should not exceed the limit of 7 ± 2 distinct terms. The same criterion is applied to the number of variables in the rule antecedent; this is to be familiar for humans.

- *Orthogonality.* For each element of the universe of discourse, the sum of all its membership values should be equal to one.

4.1. **The fuzzy system setting.** In this subsection an explanation of the fuzzy system setting is shown.

Logical parameters

- *Reasoning mechanism*: singleton-type fuzzy system, i.e. Output membership functions are real values, rather than fuzzy ones.
- *Fuzzy operators*: min.
- *Input membership function type*: orthogonal, trapezoidal.
- *Defuzzification method*: weighted average.

Structural parameters

- *Relevant variables*: there is insufficient a priori knowledge to define them; therefore, this will be one of the algorithm's objectives.
- *Number of input membership functions*: two membership functions denoted Low and High.
- *Number of output membership functions*: two singletons are used, corresponding to the benign and malignant diagnostics.
- *Number of rules*: in our approach, this is a user-configurable parameter. Will there be only one rule? The rule itself is to be found by the fuzzy-membrane-immune algorithm.

Connective parameters

- *Antecedents of rules*: to be found by the algorithm.
- *Consequent of rules*: the algorithm finds rules for the benign diagnostic; the malignant diagnostic is an else condition.
- *Rule weights*: active rules have a weight of value 1 and the else condition has a weight of 0.25.

Operational parameters

- *Input membership function values*: to be found by the evolutionary algorithm.
- *Output membership function values*: following the **WBCD** database, we used a value of 2 for benign and 4 for malignant.

4.2. **The fuzzy-membrane-immune algorithm description.** The clonal selection principle, or theory, is the algorithm used by the immune system to describe the basic features of an immune response to an antigenic stimulus [4, 10]. Clonal selection establishes the idea that only cells that recognize the antigens will proliferate where the rest will not, as depicted in figure 6 [11]. The most triggered cells selected as memory cells for future pathogens attacks

and the rest mature into antibody secreting cells called plasma cells [9, 11, 12, 15].

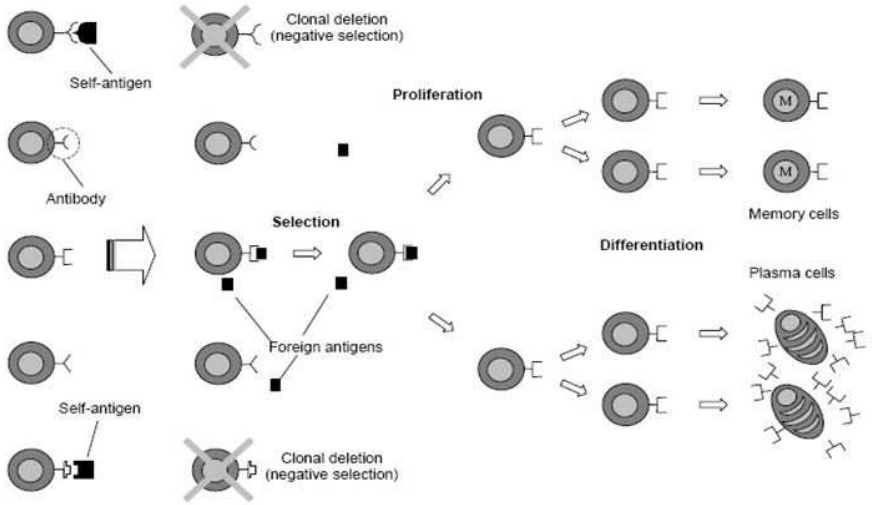


FIGURE 6. The clonal selection principle

Repertoire diversity is maintained by mutation. Mutation is random changes, these changes are introduced into the variable region genes and occasionally one such change will lead to an increase in the affinity of the antibody [20]. These higher-affinity variants are selected to enter the pool of memory cells [19]. Due to the random nature of the somatic mutation process, large proportions of mutating genes become non-functional or develop harmful antiself specificities. Those cells with low affinity receptors, or the self reactive cells, must be eliminated so that they do not significantly contribute to the pool of memory cells. The killing process here maintained by the selection algorithms. For this algorithm to work the receptor population or repertoire has to be diverse enough to recognize any foreign shape, we maintain the diversity by metadynamics. Metadynamics is adding randomly generated Abs into repertoire. A mammalian immune system contains a heterogeneous repertoire of approximately 10^{12} lymphocytes in human [9, 10, 13].

In the fuzzy-membrane-immune algorithm, the repertoire is a set of P systems, look at figure 7, each one consists of set of nested membranes, and each region has a set of solutions and a simple mutation algorithm that mutates all solutions in this region. These solutions are initialized randomly.

In every region there are a few solutions of the optimization problem to be solved and a mutation algorithm works on them. After the initial settings

all solutions are updated by the mutation algorithm placed in regions. Simultaneously, in every region, the best and worst solutions, with respect to the optimization criterion, are sent to the adjacent inner and outer regions, respectively. The best solution exists in the innermost region of a P system.

The process of updating and transporting solutions is repeated until a termination condition is satisfied. In our implementation a fixed number of iterations are used as a termination condition. The ratio of valid classification of cases is used as affinity measure of any solution. Each p system in the repertoire is supposed to be enhanced continuously by time because of maturation performed by mutation. The higher affinity solution will be found in the inner most region and cloning will performed proportional to this solution's affinity, i.e. a high membrane inner most solution affinity means a high cloning rate; a low membrane inner most solution affinity means a low cloning rate or a negative selection (clonal deletion). The concept of mutation is essential to fuzzy-membrane-immune algorithm, as it is the only way for repertoire diversification and maturation. Without sufficient mutation the population will tend to converge towards a few good solutions, possibly representing local optima. On the other hand, with too much mutation the search will have problems of focusing on potentially good solutions; this is the tradeoff between exploration and exploitation. According to our experiments we found that 0.1 mutation rate is suitable. To keep the repertoire diversity, a number of randomly generated P systems are added to the repertoire, our experiments showed that there is no need to perform this step, but in other problems it could be useful.

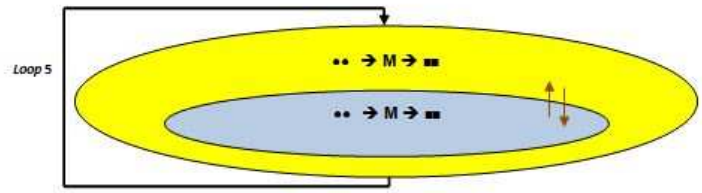


FIGURE 7. a single P system used in the fuzzy-membrane-immune algorithm

The proposed algorithm can be summarized as follows.

4.3. **The fuzzy-membrane-immune algorithm setup.** The membrane-immune algorithm used a repertoire size = 15 P systems, the algorithm maximum iterations = 30 times of +ve/-ve selection, single P systems perform 5

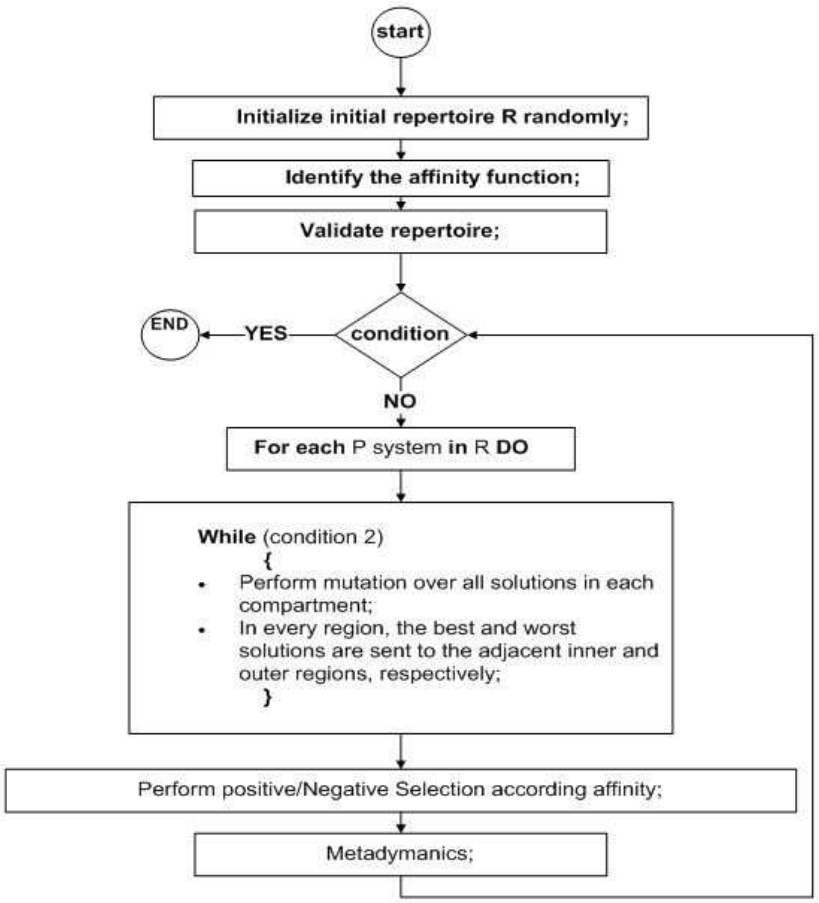


FIGURE 8. the fuzzy-membrane-immune algorithm

iterations, and every single P system has 2 regions (compartments) as depicted in figure 7. Mutation rate = 0.1. Selection and Cloning is proportional to the inner most genome's affinity. The algorithm terminates when the maximum number of iterations is reached. A flowchart of the used algorithm is depicted in figure 8. The used parameters are chosen heuristically and using try and error technique.

The algorithm applies Pittsburgh-style structure learning, the algorithm searches for three parameters, the relevant variables, the input membership function values and the antecedents of rules. They are constructed as follows: There are nine variables ($V_1 - V_9$), each variable has two parameters P and

d, defining the start point and the length of the membership function edges, respectively. The i^{th} rule has the form: $if(V_1 is A_i^1) and ... and (V_9 is A_i^9)$ *then (output is benign)* where A_i^j represents the membership function applicable to variable V_j. A_i^j can take the values: 1 (Low), 2 (High), or 0 or 3 (Other). Relevant variables are searched for implicitly by letting the algorithm choose non-existent membership functions as valid antecedents; in such a case, the respective variable is considered irrelevant.

TABLE 3. Parameters encoding of a genome, total genome length is 54+18= 72

Parameter	Values	Bits	Quantity	Total bits
P	1-8	3	97	27
d	1-8	37	9	27
A	0-3	2	9	18

TABLE 4. Database

p_1	d_1	p_2	d_2	p_3	d_3	p_4	d_4	p_5	d_5	p_6	d_6	p_7	d_7
3	5	4	1	2	8	5	1	7	7	2	5	5	5
p_8	d_8	p_9	d_9	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	
7	2	4	7	1	1	3	3	3	1	3	1	1	

TABLE 5. Rule Base

Rule	If ((v_1 is low) and (v_2 is low) and (v_6 is low) and (v_8 is low) and (v_9 is low)) then(output is benign)
Default	Else(output is malign)

The parameters encoding are described in Table 3, which form a single individual's genome. Table 4 shows a sample genome and table 5 shows an interpretation of that genome. a population size of 200 individuals showed to be sufficient for generating the fuzzy inference system; no significant enhancements appear by increasing the population above 200 individuals , and fitness-proportionate selection. The algorithm terminates when the maximum number of generations is reached.

5. Experimental Results and testing

To gain an intuitive understanding of how a classification value is computed, let us sketch a simple example. Referring to the system produced by the fuzzy-membrane-immune algorithm table 5, assume that the following values in table 6 (these represent case number 1 of the WBCD database), are presented as inputs, then the membership value of each variable is then computed in accordance with the (evolved) fuzzy rule. The result of membership values are represented in table 7. This completes the fuzzification phase.

Having computed these membership values, the inference engine can now go on to compute the so-called truth value of each rule. This truth value is computed by applying the fuzzy *'and'* operator to combine the antecedent clauses (the membership values) in a fuzzy manner; this results in the output truth value, namely, a continuous value which represents the rule's degree of activation. Thus, a rule is not merely either activated or not, but in fact is activated to a certain degree represented by a value between 0 and 1. In our example, the activation value for a rule is 0.6 and for the default case is 0.25.

Table 6. case number 1 of the WBCD database

attribute	V1	V2	V3	V4	V5	V6	V7	V8	V9
value	5	1	1	1	2	1	3	1	1

Table 7. The membership value of each variable in table 6 using the generated database of table 5.

attribute	V1	V2	V3	V4	V5	V6	V7	V8	V9
m_Low	3/5	1	-	-	-	1	1	1	-
m_High	2/5	0	-	-	-	0	0	0	-

The inference engine now goes on to apply the aggregation operator (Min), combining the continuous rule activation values to produce a fuzzy output with a certain truth value (the point marked 'fuzzy output' in Figure 4) as follows:

Degree of activation(rule 1) = $\mu_{low}(V_1)$ and $\mu_{low}(V_2)$ and $\mu_{low}(V_6)$ and $\mu_{low}(V_7)$ and $\mu_{low}(V_8)$= min{ $\mu low(V_1)$, $\mu low(V_2)$, $\mu_{low}(V_6)$, $\mu_{low}(V_7)$, $\mu_{low}(V_8)$ }= min{0.6, 1, 1, 1, 1}= 0.6.

After that the defuzzifier then kicks in (Figure 4), producing the final continuous value of the fuzzy inference system; this latter value is the appraisal value that is passed on to the threshold unit (Figure 3). In our example the appraisal value is 2.44 (i.e. benign), which is true classification according to the WCBD data base, and computed as explained below.

$$appraisal = \frac{0.6*2+0.25*4}{0.6+0.25} = 2.44$$

The membrane-immune algorithms reached a valid classification ratio equal to 97.36% using only one fuzzy rule, i.e. 665 valid diagnosis cases from 683 cases.

TABLE 8. Comparison of the best systems evolved by our approach with four other rule-based diagnostic approaches

rules number	Setiono	Setiono&Liu	Taha&Ghosh	Pen&Sipper[1]	Pena&Sipper[2]	Our study
1	95.42(2)	-	-	96. 35(3)	97.07(4)	97.36(4)
2	-	-	-	96.65(7)	-	-
3	97.14(4)	97.21(4)	-	-	-	-
4	-	-	-	-	-	-
5	-	-	96.19(1.8)	-	-	-

Table 8 presents a five related works that solved the same problem. Setiono [16], Setiono&Liu [17], and Taha&Ghosh [7] are based on boolean rule bases extracted from trained neural networks. Pen&Sipper[1], Pen&Sipper[2] are based on genetic algorithms.

Column one in table 8 represents the number of rules included in every generated fuzzy rule system. Sentio in [16] reached a valid classification ration = 95.42% with one rule and two variables in this rule. The same work reached 97.14% valid classification ratio using three rules and the average number of variables in each rules = four. Sentio and Liu [17] reached 97.21% valid classification ration with three fuzzy rules and four variables per each in average. Taha and Ghosh [7] reached 96.19% valid classification ration with five rules and average number of variables =1.8. Pen and Sipper in [1] reached 96.35% using one rule and three variables per each rule in average, 96.65% using two rules and seven variables per rule in average. In [2] the last authors reached 97.07% with one rule using four variables. Our work is explained in the last column; we reached the highest valid classification ration 97.36% using only one rule and only four variables in this rule.

The fitness measure of the fuzzy-membrane-immune algorithm can take into consideration with the number of rules to be developed in the fuzzy system, the number of variables included in each rule to improve the system's interpretability. This means smaller number of variables in rules implies higher interpretability. Table 8 shows that the fuzzy-membrane-immune algorithm produced the fuzzy system with the highest valid classification ration in comparison with other techniques in table 8. We run the membrane-immune algorithm 90 times. In every run the algorithm finds a solution of a classification ration = 97.36% as depicted in figure 9, in the other hand the best solutions of genetic algorithm have classification ratio between 94.5% to 97.07%, as mentioned in [1]. As depicted in figure 10 about half of the solutions have

classification performance less than 96.5. Thus the proposed algorithm surpasses genetic algorithm in terms of the best solution and also in the average classification ration of produced systems. From these observations we can claim that the membrane-immune is a promising algorithm in solving similar optimization problems.

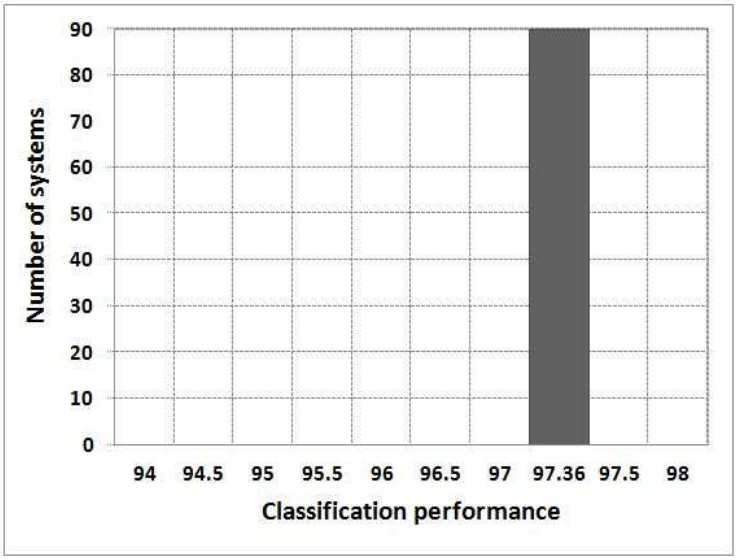


FIGURE 9. Summary of results of 90 runs. The histogram depicts the number of systems Exhibiting a given performance level.

6. Conclusions and Remarks

The proposed algorithm can be hybridized by many approaches other than artificial immune system, e.g. genetic algorithms, swarm intelligence, rough sets, etc. There are many possibilities for improving the proposed algorithm, for example different termination conditions could be used, i.e. one can terminate execution if the good solution is not changed during a predetermined number of steps, considering metadynamics in the earlier execution in the algorithm could enhance performance, metadynamics may be useful in some applications and in other not, Metadynamics needs a deeper view, and it could be implemented by applying very high mutation rates to a number of selected members besides adding some new randomly generated ones. Instead of performing cloning according the affinity of the inner most solution, it could be

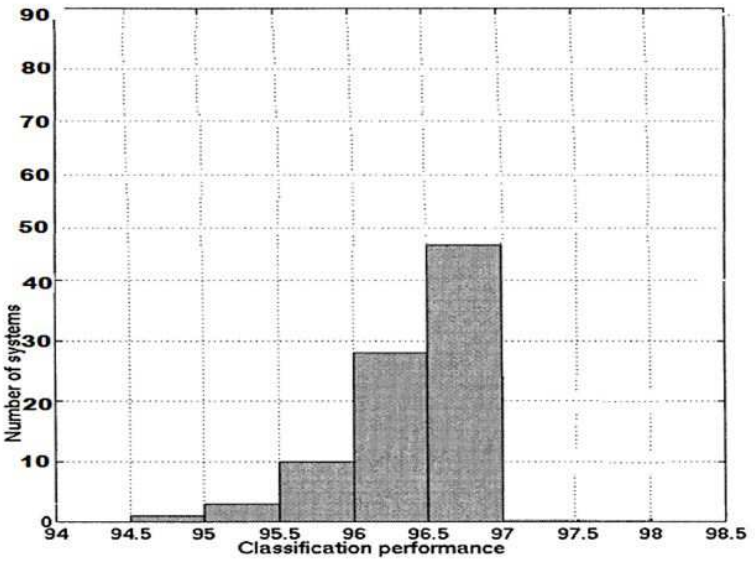


FIGURE 10. Summary of results of 90 evolutionary runs using GA. The histogram depicts the number of systems Exhibiting a given performance level at the end of the evolutionary run.

performed according to the average value of all solutions for a P system; the first mechanism takes into consideration the affinity of one solution to perform cloning but the latter counts for all solutions' affinity in a p system.

The repertoire has a number of independent P systems so; the algorithm will be easily implemented in parallel, distributed, or grid computing systems which indicates a better performance. Each p system itself could be implemented in a parallel system as it contains a number of independent regions all of them has its solutions and maturation/mutation technique. Different mutation techniques may be more suitable for different optimization problems like mutation per solution or Swap Mutation. The best mutation rate for repertoire is an open point; mutation may be high in early iteration and get smaller by time. The proposed structure could be involved in more complex structures as well. Finally, our positive results support the idea that our algorithm is a suitable approach for tackling highly constrained optimization problems.

REFERENCES

[1] C. A. Pena-Reyes, M. sipper, A fuzzy-genetic approach to breast cancer diagnosis. Artificial Intelligence in Medicine, 17(2): 131-155, October 1999.

[2] C. A. Pena-Reyes, M. Sipper, Evolving fuzzy rules for breast cancer diagnosis, Proceedings of 1998 International Symposium on Nonlinear Theory and Applications (NOLTA'98), Vol. 2. Lausanne: Presses Polytechniques ET Universitaires Romandes, 369-372, 1998.
[3] C. J. Merz, P.M. Murphy, UCI repository of machine learning databases. http://www.ics.uci.edu/m̃learn/MLRepository.html, 1996.
[4] D. Dasgupta, N. Attoh-Okine, Immunity-Based Systems:A Survey, In the proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, Orlando,1997.
[5] G. Paun, Computing with membranes ,TUCS Report 208, Turku Center for Computer Science, 1998.
[6] H. Zhang, D. Liu, Fuzzy Modeling and Fuzzy Control, Birkhauser, 2006.
[7] Taha, J. Ghosh, Evaluation and ordering of rules extracted from feed forward networks, Proceedings of the IEEE International Conference on Neural Networks, pp. 221-226, 1997.
[8] J.J. Espinosa, J. Vandewalle, Constructing fuzzy models with linguistic integrity, IEEE Transactions on Fuzzy Systems, 1999.
[9] L. N. De Castro, F. J. Zuben, Artificial Immune Systems: Part I - Basic Theory and Applications, Technical Report - RT DCA, pp. 89, 1999.
[10] L. N. de Castro,J. Timmis, H. Knidel, F. Von Zuben, Artificial Immune Systems: structure, function, diversity and an application to biclustering, Natural Computing, vol. 9, no. 3, 2010.
[11] L. N. De Castro, F. J. Zuben, Learning and optimization using the clonal selection principle, IEEE transactions on evolutionary computation, 2002.
[12] L. N. De Castro, F. J. Zuben, The Clonal Selection Algorithm with Engineering Applications, proceedings of the genetic and evolutionary computation conference, workshop on artificial immune systems and their applications; pp. 36-37, 2000.
[13] L. N. De Castro, Fundamentals of natural computing: basic concepts, algorithms, and applications. CRC Press LLC; 2007.
[14] L. N. De Castro, J. Timmis, Artificial Immune Systems A new computational approach, Springer, 2002.
[15] L. N. De Castro, Natural computing, Encyclopedia of information science and technology, vol. IV. Idea Group Inc, 2005.
[16] R. Setiono, Extracting rules from pruned neural networks for breast cancer diagnosis, Artificial Intelligence in Medicine, vol. 8, no. 1,pp. 37-51, 1996.
[17] R. Setiono, H. Liu, Symbolic representation of neural networks. IEEE Computer, vol. 29, no.3, pp.71-77, 1996.
[18] R.R. Yager, L.A. Zadeh, Fuzzy Sets, Neural Networks, and Soft Computing, New York, Van Nos-trand Reinhold, 1994.
[19] S. Forrest, S. A. Hofmeyrt, A. Somayajit, Computer Immunology, University of New Mexico Albuquerque, NM 87131-1386, March 21, 1996.
[20] T. Back, D. Fogel and Z. Mechalewicz, Evolutionary computation, basic algorithms and operators, institute of physics publishing, 2000.

[1]Department of Computer Science, Faculty of Information Technology Misr University for Science and Technology, Al-Motamayez District, Postal code: 15525, 6th of October City, Egypt

E-mail address: emadnabilcs@gmail.com

[2] Department of Computer Science, Faculty of Computers and Information Cairo University, 5 Dr. Ahmed Zewail Street, Postal Code: 12613,Orman, Giza, Egypt.

E-mail address: a.badr.fci@gmail.com, i.farag@fci-cu.edu.eg

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

A STUDY ON HIERARCHICAL CLUSTERING BASED SOFTWARE RESTRUCTURING

ZSUZSANNA MARIAN

Abstract. Finding refactorings that can automatically improve the internal structure of a software system is continuously researched in the search based software engineering literature. In this paper we will investigate through a case study, whether the use of unsupervised learning methods (hierarchical clustering) can be beneficial in the process of automatic refactoring identification. We will compare the results of two algorithms (one that uses hierarchical clustering and one that does not) for a case study, and show that the algorithm that uses hierarchical clustering is capable of finding refactorings which are not found by the other algorithm.

1. Introduction

It is well-known that software system restructuring (also called refactoring) is an important part both for the development and the maintenance of software systems. During development, with the change of requirements or with the addition of new features, refactoring might become necessary. Also, refactoring is one of the steps in the Test-Driven Development cycle. During the maintenance phase of the software life-cycle, refactoring can be used to restructure the system in order to facilitate other maintenance activities, such as improving performance or implementing new features.

Since nowadays most software systems are complex, containing many classes with complicated relations between them, the field of automatic refactorings identification gained importance. The development of new algorithms and tools that are capable of finding a good restructuring of a software system is an important research domain.

Received by the editors: April 10, 2012.

2010 *Mathematics Subject Classification.* 68N99, 62H30.

1998 *CR Categories and Descriptors.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement - Restructuring, reverse engineering, and reengineering; D.2.8 [**Software Engineering**]: Metrics - Product metrics; I.5.3 [**Computing Methodologies**]: Pattern Recognition - Clustering.

Key words and phrases. software refactoring, hierarchical clustering, software metrics.

In this paper we aim at investigating the usefulness of using unsupervised learning for software restructuring. The case study we considered for evaluation highlights the effectiveness of unsupervised learning models to uncover hidden patterns in data.

The structure of this paper is the following: Section 2 presents a short review of the existing literature. Section 3 presents the existing theoretical background. Section 4 describes the HAC algorithm, a method that uses unsupervised learning for software restructuring. Section 5 presents a case study we used, and the results obtained for an existing algorithm (Subsection 5.1) and our unsupervised learning based algorithm (Subsection 5.2), together with a comparative analysis of the results (Subsection 5.3). Section 5 ends with the comparison of our algorithm to other existing algorithms from the literature that use unsupervised learning (Subsection 5.4). Finally, Section 6 presents our conclusions and further research directions.

2. Literature review

In this section we aim at presenting other clustering based software restructuring methods that can be found in the literature.

One of the early works that examines the idea of using clustering for reverse engineering is [1]. It describes a series of experiments on how clustering could be used for software remodularization by defining how an entity can be represented, how to decide when two entities should belong to the same clusters and how to apply a clustering algorithm to the entities. It is an important study, many other papers that use clustering reference it, for example when deciding what kind of linkage method to use.

A recent paper, [11], first identifies classes that have a low cohesion and then finds *Extract Class* refactorings to divide the class into more cohesive parts. In the first step, three software metrics are used (LCOM2, TCC and DOCMA(AR)) to find classes with low cohesion, also called God classes. In the second step, agglomerative clustering is used for the found classes, using as a distance metric the Jaccard distance on property sets defined for attributes and methods. The property set for an attribute is the attribute and the methods that use it, while for a method it contains the method itself and the methods and fields used by it.

Another paper that finds *Extract Class* refactorings is [5]. They use hierarchical clustering for attributes and methods inside a class, using as distance the Jaccard distance on entity sets. The entity set of an attribute contains the attribute and the methods that access it, while the entity set of a method contains the method itself, the methods invoked by it and the attributes used by it.

Unlike the above presented two methods, the HARS algorithm presented in [4], is capable of finding many types of refactorings (*Move Method*, *Move Attribute*, *Inline Class* and *Extract Class*). It considers as entity a method, a class or an attribute, and defined a set of relevant properties, and a distance metric between these sets. This method is similar to the one we will present below.

3. Background

We have previously introduced in [10] an approach that uses software metrics for identifying an improved structure of a software system, i.e. a structure that is likely to correspond to an improved design of it. In this section we will give a brief overview of the vector space model (Subsection 3.1) and the ARI algorithm (Subsection 3.2) introduced in [10].

3.1. **The Vector Space Model.** The main idea of this approach is to represent the components of a software system as a multidimensional vector, whose elements are the values of different software metrics applied to the given component. In order to give a formal definition, we considered that a software system S is a set of components $S = \{s_1, s_2, ..., s_n\}$, where s_i, $1 \leq i \leq n$, can be either an application class, or a method from an application class, and will be called *entity* in the following.

Each entity from a software system is characterised in [10] by a list of relevant features, which are the values for the following software metrics:

(1) Relevant Properties (RP) [3]
(2) Depth in Inheritance Tree (DIT) [2]
(3) Number of Children (NOC) [2]
(4) Fan-In (FI) [8] and [9]
(5) Fan-Out (FO) [8] and [9]

Using the above presented metrics, each entity s_i, $1 \leq i \leq n$, from the software system S can be represented as a 5-dimensional vector, having as components the values of the different metrics, scaled to [0,1]: $s = (rp(s_i), dit(s_i), noc(s_i), fi(s_i), fo(s_i))$ or, more formally, $s = (s_{i1}, s_{i2}, s_{i3}, s_{i4}, s_{i5})$. Each element s_{ik}, $1 \leq k \leq 5$ is the value of the corresponding software metric, and the number of metrics is denoted by m (to have a more general definition).

Using this vector space model representation of the entities, a distance metric, $d(s_i, s_j)$, can be defined as an adaptation of the *Euclidian distance*. This metric, computed using Formula 1, is defined in such a way to measure the dissimilarity between two entities. This means that it will assign small values to pairs of entities that are cohesive and have to belong together, in the same application class, and high values (possibly ∞) to pairs that are not cohesive.

(1)
$$d(s_i, s_j) = \begin{cases} 0 & if \quad i = j \\ \sqrt{\frac{1}{m} \cdot \left(1 - \frac{|s_{i1} \cap s_{j1}|}{|s_{i1} \cup s_{j1}|} + \sum_{k=2}^{m} (s_{ik} - s_{jk})^2\right)} & if \quad s_{i1} \cap s_{j1} \neq \emptyset \\ \infty & otherwise \end{cases},$$

It is easy to prove that Formula 1 is a semi-metric function, and that the distance between two entities that have common relevant properties is less than 1.

3.2. **The Automatic Refactorings Identification - ARI algorithm.** Considering the vector space model presented in 3.1, an algorithm capable of finding a good restructuring of a software system was introduced in [10]. The input of the algorithm is the vector space model representation for each entity s from the software system S, while the the output is a grouping of these entities into different groups - *clusters*, - according to their similarity. Each such cluster corresponds to an application class from the system. The set of all clusters is called a partition and it represents a possible structure of the software system S. The algorithm ARI tries to find a partition that corresponds to a good internal structure of S. Then, comparing this partition to the original structure of the software system, a list of refactorings can be identified.

The ARI algorithm initially places each entity that is an application class into a separate cluster. Then, the main idea is to place each method from the software system in the cluster (i.e. application class), to which it is closest, considering the distance metric d, or to place it in a new cluster if its distance to the already existing application classes is greater than 1. Finally, if there are application classes whose distance is less than 1, they will be merged.

Comparing the result of the ARI algorithm to the original partition of the software system, three types of refactorings can be identified: *Move Method*, *Extract Class*, *Inline Class*. For a detailed description of the ARI algorithm and the identified refactorings, one can consult [10].

4. Unsupervised learning based restructuring

In this section we introduce an extension of the algorithm proposed in [10], presented in Subsection 3.2, named HAC (Hierarchical Agglomerative Clustering). It uses the same vector space model and distance semi-metric as the ARI algorithm and hierarchical clustering, an unsupervised learning method, to build a good restructuring of a software system.

The HAC algorithm is based on *hierarchical agglomerative clustering* and uses a heuristic function for merging two clusters. This heuristic function expresses that two clusters are merged only if the distance between them is less than 1. This value for threshold was chosen, because distances higher than 1 are obtained only for unrelated entities. The distance between two clusters was computed using *complete link* linkage method, because this method gave the best results.

The main steps the HAC algorithm performs for identifying the partition $\mathcal{K}$ that is likely to correspond to an improved structure of the software system S are the following:

(1) Initialise $\mathcal{K}$ as an empty partition.
(2) For each entity s_i from S create a new cluster that contains only s_i, and add it to $\mathcal{K}$. Now $\mathcal{K} = \{K_1, K_2, ..., K_n\}$ is the initial partition, containing as many clusters as the number of entities in S.
(3) As long as changes can be done, the following steps are repeated:
    i. Compute the distance $d_{i,j}$ as $d(K_i, K_j)$, where $1 \leq i \leq n$ and $i \leq j \leq n$.
    ii. Select the minimum distance d_{min} from the distances computed at the previous step. Let i^* and j^* be the indexes of the clusters whose distance is d_{min}.
    iii. If $d_{min} \leq 1$ then we will create a new cluster, $K_{new} = K_{i^*} \cup K_{j^*}$, and modify $\mathcal{K}$ in the following way: $\mathcal{K} = \mathcal{K} \setminus \{K_{i^*}, K_{j^*}\} \cup K_{new}$.

Like the ARI algorithm, this algorithm will also return a partition $\mathcal{K}$ of the software system, where each cluster corresponds to an application class. If this partition is equal to the original partition of S, $\mathcal{K}'$, then we consider that S has a good internal structure and no changes need to be done. If there are differences between $\mathcal{K}$ and $\mathcal{K}'$, S needs to be restructured. Different types of differences correspond to different refactorings. The refactorings identified by the HAC algorithm are the following:

- **Move Method refactoring** [6] - It means moving method m from class c to class c'. Such a refactoring is justified when a method uses or is used more by a different class than its own class. This type of refactoring is identified, when a method m will be placed by HAC into a cluster with a different application class, than its original application class.
- **Extract Class refactoring** [6] - Extract class refactoring means creating a new application class c and moving some methods into it. This refactoring is justified in case of a large class, with many functions that should be split into at least two classes. This type of refactoring appears, when at the end of HAC there are clusters, that contain only

methods, and no application classes. For these clusters a new application class will be created that will contain the methods from the corresponding cluster. Also, in such cases, the number of clusters in $\mathcal{K}$ increases compared to the number of clusters in $\mathcal{K}'$.

- **Inline Class refactoring** [6] - It means moving one class (together with methods and attributes), inside of another class, and it usually happens for small classes that does not do many things. This type of refactoring is identified, when $\mathcal{K}$ contains clusters which have two application classes inside. In this case, one of the classes will become an inline class for the other one, which means that it will be moved - together with methods and attributes - inside the other class. The presence of such refactoring is shown also by the fact that $\mathcal{K}$ contains fewer clusters than $\mathcal{K}'$.

5. Computational experiments

In this section we aim at experimentally evaluating ARI and HAC algorithms, providing a comparative analysis of the obtained results. Two case studies are considered for evaluation, for which we applied both algorithms. The first case study was a small, artificial example (taken from [12]) with two classes, one of them containing a method that should belong to the other class. For this simple example both algorithms identified the correct restructuring.

For our second case study we used JHotDraw (version 5.1), an open source software [7]. It is a Java GUI framework for technical and structured graphics, developed by Erich Gamma and Thomas Eggenschwiler. We chose JHotDraw because it is a well-known example for the use of design patterns and for good design, so we expected our algorithms to find only a few possible refactorings. Another reason for choosing JHotDraw was the fact that, unlike our first case study, this is a complex project, consisting of 173 classes, 1375 methods and 475 attributes.

5.1. **ARI results.** Applying the ARI algorithm to the JHotDraw framework, 20 *Move Method* refactorings were identified, which are presented in Table 1. The first column shows the name of the method to be moved, while the second contains the name of the class where the method should be moved to. The last column shows whether we considered the given refactoring justifiable or not. The justifications for the values in the last column (both Justified and Misplaced) are given below. When deciding on the justifications, we considered three criteria. The first criterion was, whether the refactoring is justified conceptually. The second criterion was, how complicated would it be to perform the actual refactoring. In case of the Move Method refactorings, we considered how many attributes does the method use and where is it called inside

the class. The third criterion, used mainly for the Extract Class refactoring identified by the HAC algorithm, was whether the methods to be moved to a new class could be replaced by a single method, sufficiently general that it would implement the same responsibilities as the methods replaced with it. For this criterion the justification was the"Rule of Three" presented in Martin Fowler's book on refactoring: when you write a similar piece of code the third time, it is time to refactor [6].

	Method	Target class	Remark
1.	**DrawApplet.createAttributeChoices**	**CommandChoice**	Justifiable
2.	**DrawApplet.createFontChoice**	**CommandChoice**	Justifiable
3.	**DrawApplication.saveAsStorableOutput**	**StorableOutput**	Justifiable
4.	**DrawApplication.paletteUserOver**	**ToolButton**	Justifiable
5.	**DrawApplet.paletteUserSelected**	**ToolButton**	Justifiable
6.	**DrawApplet.paletteUserOver**	**ToolButton**	Justifiable
7.	**DrawApplet.toolDone**	**ToolButton**	Misplaced
8.	**DrawApplication.createColorMenu**	**ColorMap**	Misplaced
9.	**DrawApplet.setupAttributes**	**ColorMap**	Misplaced
10.	**DrawApplet.createColorChoice**	**ColorMap**	Misplaced
11.	**DrawApplication.createEditMenu**	**CommandMenu**	Misplaced
12.	**DrawApplication.createAlignmentMenu**	**CommandMenu**	Misplaced
13.	**DrawApplication.createArrowMenu**	**CommandMenu**	Justifiable
14.	**DrawApplication.createFontMenu**	**CommandMenu**	Justifiable
15.	**DrawApplication.createFontSizeMenu**	**CommandMenu**	Justifiable
16.	**DrawApplication.createFontStyleMenu**	**CommandMenu**	Justifiable
17.	**DrawApplication.selectionChanged**	**CommandMenu**	Misplaced
18.	**DrawApplet.readFromStorableInput**	**StorableInput**	Justifiable
19.	**PertFigure.asInt**	**NumberTextFigure**	Misplaced
20.	**PertFigure.setInt**	**NumberTextFigure**	Misplaced

TABLE 1. *Move Method* refactorings suggested by the ARI algorithm

We give below the justification for the extracted refactorings.

DrawApplet.createFontChoice, *DrawApplet.createAttributeChoices* and *DrawApplet.createColorChoice*. These are the only three methods from the DrawApplet class that have a name of the form *"createSomethingChoice"* and all three of them were identified as possible *Move Method* refactorings, but *createColorChoice* was suggested to be moved to the *ColorMap* class (this is why it is marked "Misplaced"). All three of them use the method *addItem* from class *CommandChoice* many times, which can be a sign of a need for refactoring.

DrawApplication.saveAsStorableOutput. This method creates an object of

type *StorableOutput* and calls its *writeStorable* method, passing to it as parameter an object of type *Drawing* (which, in turn, extends interface *Storable*). Analysing the method *writeStorable*, we can see that the most important part is calling the *write* method on the *Storable* object. Since the main part of writing is already done in class *StorableOutput*, and *saveAsStorableOutput* does nothing strictly related to the class *DrawApplication*, it would make sense moving the whole method to the *StorableOutput* class.

DrawApplication.paletteUserOver, *DrawApplet.paletteUserSelected*, *DrawApplect.paletteUserOver*. Methods *paletteUserOver* and *paletteUserSelected* are defined in interface *PaletteListener*. Since both *DrawApplication* and *DrawApplet* implement this interface, the methods are implemented in both classes (so there is a *paletteUserSelected* method in class *DrawApplication*, too). The implementation of the methods is the same in both classes, which is an argument for moving them to a single class. It would be possible to move the implementation of the *PaletteListener* interface, together with the methods, to the *ToolButton* class as suggested by our algorithm.

DrawApplet.toolDone. This method comes from the *DrawingEditor* interface which is implemented by the *DrawApplet* class. Since this interface contains six methods, moving only one of them to a separate class is impossible.

DrawApplication.createColorMenu, *DrawApplet.setupAttributes*, *DrawApplet.createColorChoice*. All three methods use many functions from the *ColorMap* class, but they cannot be moved there. As mentioned above *createColorChoice* could be moved, but not to the *ColorMap* class. The same is true for the *createColorMenu* method, and it will be detailed in the next part. *SetupAttributes* uses five attributes of the *DrawApplet* class, so it cannot be moved either.

DrawApplication.createColorMenu, *DrawApplication.createEditMenu*, *DrawApplication.createAllignmentMenu*, *DrawApplication.createArrowMenu*, *DrawApplication.createFontMenu*, *DrawApplication.createFontSizeMenu*, *DrawApplication.createFontStyleMenu*. First of all, *createColorMenu* was marked "Misplaced", because it was suggested to be moved to class *ColorMap* instead of *CommandMenu* like the rest of the methods. The rest of the methods are quite independent from class *DrawApplication*, they use only one of its attributes, which could easily be passed as a parameter if the methods were moved. The reason why methods *createEditMenu* and *createAllignmentMenu* were marked as "Misplaced" is that they are called in a method *createMenus*, together with three other methods that create menus and it would be strange to move two

methods to a different class, but leave three in *DrawApplication*.

DrawApplet.readFromStorableInput. This case is very similar to the one with method *DrawApplication.saveAsStorableOutput*, the main part is done in the *read* method from the *Storable* interface and the *readStorable* method from class *StorableInput*, so the whole method could be moved to the *StorableInput* class.

PertFigure.asInt, *PertFigure.setInt*. The source code of the JHotDraw framework contains a package *samples* with four examples of how to use the framework. These two methods belong to the example *pert*, so they cannot be moved into a class from the framework.

5.2. **HAC results.** Applying the HAC algorithm to the same JHotDraw framework, we got slightly different results: out of the 158 clusters identified by the algorithm only 3 did not correspond to the existing structure of the framework. These three clusters, represent possible *Extract Class* refactorings, meaning that new classes should be created, and the selected methods moved to the classes. The methods belonging to these three clusters are presented in Table 2. Just like in Table 1, we marked each method either "Justifiable" or "Misplaced". The reasons for assigning the labels are presented below.

	Method	Remark
1	**ChangeConnectionHandle.findConnectableFigure**	Justifiable
2	**ConnectionHandle.findConnectableFigure**	Justifiable
1	**GroupFigure.handles**	Justifiable
2	**TextFigure.handles**	Justifiable
3	**StandardDrawing.handles**	Justifiable
4	**NodeFigure.handles**	Misplaced
5	**PertFigure.handles**	Misplaced
1	**GroupCommand.execute**	Misplaced
2	**UngroupCommand.execute**	Misplaced
3	**AlignCommand.execute**	Misplaced
4	**SendToBackCommand.execute**	Justified
5	**BringToFrontCommand.execute**	Justified

TABLE 2. *Extract Class* refactorings suggested by the HAC algorithm

We give below the justifications for the found refactorings.

ChangeConnectionHandle.findConnectableFigure, *ConnectionHandle.findConnectableFigure*. The reason why these two methods belong to the same cluster

is that they look exactly the same. They both are private methods which take as parameter two integers and a *Drawing*, and they are both called from the *invokeStep* method of their class, so they could be moved together in the same class.

GroupFigure.handles, *TextFigure.handles*, *StandardDrawing.handles*, *NodeFigure.handles*, *PertFigure.handles*. All these methods are similar, but the last two of them are marked "Misplaced", because they belong to the examples provided in the JHotDraw framework. All of them build a vector containing four objects from classes that implement the *Handle* interface. They also use the *RelativeLocator* class to define the position of these handles. This class contains nine different "positions": center, southWest, southEast, south, northWest, northEast, west, north, east, but in the above methods only northWest, northEast, southWest and southEast is used.

GroupCommand.execute, *UngroupCommand.execute*, *AlignCommand.execute*, *SendToBackCommand.execute*, *BringToFrontCommand.execute*. Even if all the methods use only one or two attributes from their class, only the methods *SendToBackCommand* and *BringToFrontCommand* can be moved together, because they are very similar, with the difference that one calls the *sendToBack* method and the other the *bringToFront* method on a *Drawing* object. But the rest of the three methods are so different from these two and from each other, that they cannot be moved.

5.3. **Comparative Analysis.** Considering the results obtained after applying ARI and HAC algorithms and illustrated in Tables 1 and 2 we can conclude the following:

- Both algorithms identify most of the existing classes in the JHotDraw framework as correct, internally well-structured classes, which was expected, because JHotDraw is considered an example of good design. Still, both algorithms find possible refactorings, some of them justifiable, some of them not. The HAC algorithm finds only 5 misplaced methods, while ARI finds 9 misplaced methods. For both algorithms, two of the misplaced methods come from the package which contains sample applications, so they do not really count as errors.
- The HAC algorithm correctly identifies 3 possible *Extract Class* refactorings, which are not identified by the ARI algorithm.

The obtained results show that applying an unsupervised learning model (like hierarchical clustering in this paper) would be beneficial, as machine learning models are able to capture hidden patterns in data.

5.4. **Comparison to Related Work.** In this section we aim to compare our method to other methods from literature, presented in Section 2, that use hierarchical clustering to find possible refactorings.

All the methods presented in Section 2, compute the distance between elements using some kind of sets, defined for attributes, methods and, in case of [4], classes. These sets are similar to the relevant properties metric in our approach, but we use four other metrics for the distance, making it more complex.

Compared to [11] and [5], the HAC (and also the ARI) algorithm is capable of identifying three types of refactorings (not just the *Extract Class*). Direct comparison of the results is impossible, because they use different software projects as case study, which might not be available (in [11], a banking application is used, developed by some students), and they do not report the exact results (refactorings), only percentages of correctly identified refactorings.

The closest to our method is the HARS algorithm, in [4]. The difference is, that it uses only the set of relevant properties for computing the distance between two entities, and considered attributes as entities, too. [4] uses JHotDraw as a case study too, and it finds only one out of the three *Extract Class* refactorings that HAC finds (but it finds also two *Move Attribute* refactorings, which neither ARI nor HAC can find).

6. Conclusions and further work

In this paper we presented a new algorithm, HAC, that uses unsupervised learning to find a good restructuring of a software system. Using the JHotDraw framework as a case study, we compared the new algorithm, to an existing one, called ARI, which identifies refactorings, but without using unsupervised learning.

The results of the case study, presented on Tables 1 and 2 show that the HAC algorithm identifies *Extract Class* refactorings, that are not identified by the ARI algorithm. This demonstrates that using unsupervised learning methods (hierarchical clustering in our case) for automatic refactorings identification is beneficial, because it can uncover hidden patterns in data.

Although the introduced algorithm is capable of finding three types of refactorings, in the future we want to increase this number. This can be done by considering the attributes as entities too, thus adding the *Move Attribute* refactoring. Further investigations will be made to extend the vector space model characterising the entities from the software system by identifying other software metrics that are relevant for software restructuring.

References

[1] Nicolas Anquetil and Timothy Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of 6th Working Conference on Reverse Engineering*, pages 235–255, Atlanta, USA, October 1999.

[2] Shyam Chidamber and Chris Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[3] Istvan Gergely Czibula and Gabriela Serban. Improving systems design using a clustering approach. *International Journal of Computer Science and Network Security*, 6(12):40–49, 2006.

[4] Istvan Gergely Czibula and Gabriela Serban. Hierarchical clustering for software systems restructuring. *INFOCOMP Journal of Computer Science*, 6(4):43–51, 2007.

[5] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiu, and Jorg Sander. Decomposing object-orietend class modules using an agglomerative clustering technique. In *Proceedings of International Conference on Software Maintenance*, pages 93–101, Edmonton, Canada, 2009.

[6] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] Erich Gamma. JHotDraw Project. http://sourceforge.net/projects/jhotdraw.

[8] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

[9] Sayyed Garba Maisikeli. *Aspect Mining Using Self-Organizing Maps With Method Level Dynamic Software Metrics as Input Vectors*. PhD thesis, Graduate School of Computer and Information Sciences Nova Southeastern University, 2009.

[10] Zsuzsanna Marian, Gabriela Czibula, and Istvan Gergely Czibula. Using software metrics for automatic software design improvement. *Studies in Informatics and Control*, 2012. Submitted for review.

[11] Akepogu Ananda Rao and Kalam Narendar Reddy. Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique. *International Journal of Computer Science Issues*, 8(2):185–194, 2011.

[12] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics based refactoring. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 30–38, Washington, DC, USA, 2001. IEEE Computer Society.

Department of Computer Science,, Faculty of Mathematics and Computer Science,, Babeş-Bolyai University, Kogălniceanu 1, Cluj-Napoca, 400084, Romania.

E-mail address: marianzsu@cs.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

A STUDY ON USING ASSOCIATION RULES FOR PREDICTING PROMOTER SEQUENCES

MARIA IULIANA BOCICOR

Abstract. The problem of promoter identification in DNA sequences is of major importance within bioinformatics. As the conditions for a region of DNA to function as a promoter are not known, machine learning based classification models are still developed to approach this problem. Relational association rules are a particular type of association rules and describe numerical orderings between attributes that commonly occur over a data set. This paper aims to investigate a classification model based on relational association rules mining for the problem of promoter sequences prediction. Some further extensions to this model are introduced and we provide a comparison of the original algorithm with its newly introduced versions.

1. Introduction

Association rule mining means searching attribute-value conditions that occur frequently together in a data set [4, 9]. Ordinal association rules [5] are a particular type of association rules, which specify ordinal relationships between record attributes that hold for a certain percentage of the records in a data set. However, in real world data sets, attributes with different domains and relationships between them, other than ordinal, exist and therefore ordinal association rules are not powerful enough to describe data regularities. Consequently, Serban et al. introduced in [8] *relational association rules* in order to be able to capture various kinds of relationships between record attributes.

The problem of predicting whether a DNA sequence contains or not a promoter region is an important problem within bioinformatics, mainly because determining the promoter region in the DNA is a significant step in the process

Received by the editors: May 5, 2012.

2010 *Mathematics Subject Classification.* 68P15, 68T05.

1998 *CR Categories and Descriptors.* I.2.6[**Computing Methodologies**]: Artificial Intelligence – *Learning*; H.2.8[**Information systems**]: Database Applications – *Data Mining*.

Key words and phrases. Bioinformatics, Promoter Sequences Prediction, Machine Learning, Association Rule Mining.

of detecting genes. This classification problem was already approached both in the biological and computer science literature and several machine learning methods have proven to be very suitable and efficient.

In this paper we aim to investigate some extensions to a relational association rules based classification model for the problem of promoter sequences prediction, model that we previously introduced in [2]. The modified model will be evaluated on the data set that was used in [2] and the obtained results will be analysed and interpreted.

The rest of the paper is organized as follows. The problem of promoter sequences classification, as well as an existing classification model based on relational association rules are introduced in Section 2. Section 3 presents two extensions to this model. Experimental evaluations, analysis and comparisons of the three algorithms are given Section 4. Conclusions and further work are outlined in Section 5.

2. Background

In this section we will briefly present the problem of promoter sequences prediction, then review some fundamental aspects related to the relational association rules based classifier that we previously introduced for solving this problem [2].

2.1. **Promoter Sequences Prediction.** There are two processes that are involved in the synthesis of proteins from the DNA molecules. During the first process, called *transcription*, a single stranded RNA molecule, called messenger RNA is synthesized from one of the strands of DNA corresponding to a gene (a gene is a segment of the DNA that codes for a type of protein). This process begins with the binding of an enzyme called RNA polymerase to a certain location, that determines which of the two strands of DNA will be transcript and in which direction. This exact site is recognized by the RNA polymerase due to the existence of certain regions of DNA placed near the beginning of a gene, regions called *promoters*. The *promoter sequences prediction problem* refers to determining if a given DNA sequence contains or not a promoter region.

Because determining the promoter regions in the DNA is an important step in the process of detecting genes, the problem of promoter identification is of major importance within bioinformatics. As the conditions for a DNA sequence to function as a promoter are not known, machine learning methods are suitable to approach this problem because they can learn useful descriptions of concepts when given only instances - DNA sequences that are assumed to contain underlying but unknown patterns of base pairs [10].

2.2. **Promoter sequences Classifier using Relational Association Rules - *PCRAR*.** *Relational association rules* were introduced in [8] as an extension to association rules, in order to be able to discover various kinds of relations or correlations that exist between data in large data sets. Classical association rules discard any quantitative information that may exist between record attributes in data sets, but many times this type of information can give valuable insights into the problem at hand. Therefore, the extension of classical association rules towards ordinal and more general, relational association rules allows the uncovering of much stronger rules that consequently achieve superior data mining, or classification.

In [2], we introduced *PCRAR* - a supervised learning technique for the prediction of promoter sequences, based on relational association rules mining. We have started from the intuition that in the problem of deciding if a DNA sequence contains or not promoter regions, relationships between the nucleotides that form the DNA sequence [7] may be relevant.

The main idea of this classifier is the following. In a supervised learning scenario for predicting promoter sequences, two sets containing positive and negative instances are given. These sets will be used for training the classifier, which actually refers to discovering binary relational association rules between nucleotides in the given DNA sequences. We detect in the training data sets all the interesting binary relational association rules (rules between two attributes), with respect to the user-provided support and confidence thresholds. After the training is completed, when a new instance (DNA sequence) has to be classified (positive - if it contains a promoter region, or negative, otherwise), we reason as follows. Considering the binary rules discovered during training using the sets of positive and negative instances, the probability to assign the new instance to the positive class will be computed. If this probability is greater than or equal to 0.5, then the query instance will be classified as a positive instance, otherwise it will be classified as a negative instance. For more details about the relations that were used, how the data was preprocessed and the way in which the probabilities of assigning a new sequence to the positive or the negative class were computed, we refer the reader to [2].

3. Extensions of the *PCRAR*

This section aims to present two extensions we propose for the relational association rules based classifier introduced in [2] and used for promoter sequences prediction. The first method was developed in order to investigate how the confidence of the relational association rules discovered in the training data influences the accuracy of the classification task. The second refers to the length of the generated rules. As the classifier mentioned in Subsection

2.2 generated and used only binary rules (rules of length 2), we now aim to investigate the use of rules of any length.

3.1. **Confidence Based Probability Computation.** After the training phase of the classifier introduced in [2] ($PCRAR$) is completed, during the testing phase, two probabilities are computed for each new DNA sequence: P_+ - the probability that the sequence belongs to the positive class, i.e., it contains a promoter region and P_- - the probability that it belongs to the negative class, i.e., it does not contain a promoter. The way these probabilities are computed depends solely on the total number of generated association rules (positive and negative) and on the number of rules that the new sequence verifies or not, not taking into consideration the confidences of the rules.

We propose in this subsection a new way of computing the conditional probabilities for a new DNA sequence, which is based on the confidences of the generated relational association rules. It can be proven that the sum of the probabilities of the two possible outcomes (an instance to be classified as *positive* or *negative*) is 1.

We first introduce some notations that will be used in the following:

- S - a new DNA sequence, that must be classified as containing or not a promoter region.
- RAR_+/ RAR_- - the set of relational association rules having a minimum *support* and *confidence*, determined using the training data set containing the positive/negative instances.
- $RAR_+(S)/RAR_-(S)$ - a subset of RAR_+ ($RAR_+(S) \subseteq RAR_+$), respectively RAR_- ($RAR_-(S) \subseteq RAR_-$), containing the positive/negative rules that are verified in the sequence S. The set of relational association rules generated for the positive/negative instances, that are not verified in the sequence S will be denoted by $NRAR_+(S)$, respectively by $NRAR_-(S)$, where $NRAR_+(S) \subseteq RAR_+$ and $NRAR_-(S) \subseteq RAR_-$. It is obvious that $NRAR_+(S) \bigcup RAR_+(S) = RAR_+$ and that $NRAR_-(S) \bigcup RAR_-(S) = RAR_-$.
- $conf(\mathcal{R})$ - the confidence of an arbitrary relational association rule $\mathcal{R}$.

The steps we propose for computing the conditional probabilities are:

- Determine s_+ the total sum of the confidences of all the rules from the set RAR_+ and s_- the total sum of the confidences of all the rules from the set RAR_-:

$$s_+ = \sum_{\mathcal{R} \in RAR_+} conf(\mathcal{R}) \qquad s_- = \sum_{\mathcal{R} \in RAR_-} conf(\mathcal{R})$$

- Determine $s_+(S)$ the total sum of the confidences of the rules from $RAR_+(S)$ and $sn_+(S)$ the total sum of the confidences of the rules from $NRAR_+(S)$:

$$s_+(S) = \sum_{\mathcal{R} \in RAR_+(S)} conf(\mathcal{R}) \qquad sn_+(S) = \sum_{\mathcal{R} \in NRAR_+(S)} conf(\mathcal{R})$$

- Determine $s_-(S)$ the total sum of the confidences of the rules from $RAR_-(S)$ and $sn_-(S)$ the total sum of the confidences of the rules from $NRAR_-(S)$:

$$s_-(S) = \sum_{\mathcal{R} \in RAR_-(S)} conf(\mathcal{R}) \qquad sn_-(S) = \sum_{\mathcal{R} \in NRAR_-(S)} conf(\mathcal{R})$$

- Calculate the probability $P_+(S)$ to classify the instance S as a *positive* one as:

$$P_+(S) = \frac{1}{2}\left(\frac{s_+(S)}{s_+} + \frac{sn_-(S)}{s_-}\right) \tag{1}$$

The probability $P_-(S)$ to classify the instance S as a *negative* one could be computed in the same way, but it can be easily proven that the sum of the probabilities of the two possible outcomes (an instance to be classified as *positive* or *negative*) is 1. Therefore, if $P_+(S) \geq 0.5$ then the instance S will be classified as a *positive* instance, otherwise it will classified as a *negative* instance.

3.2. ***K*-length Rules Generation.** The *PCRAR* is based on an algorithm for the discovery of interesting ordinal association rules, called *DOAR* and introduced in [1]. This algorithm identifies ordinal association rules using an iterative process that consists in length-level generation of candidate rules, followed by the verification of the candidates for minimum support and confidence compliance.

In [2] we considered that a certain rule with a length greater than two is verified if all its binary subrules are verified and for computing the probability to classify a new instance as positive or negative we took into account only the number of verified/unverified rules. Therefore, for the *PCRAR* classifier, it was sufficient to generate only the binary interesting rules. This also led to a very fast training for this classifier. Here we propose, as another version of the algorithm, the generation of rules of any length k, the maximum length being, obviously, the number of attributes of an instance (for any instance S, let us denote its number of attributes $|S|$). A k-length rule is verified by an

instance if all its $k-1$ binary subrules are verified. As soon as the training phase is completed, meaning that all k-length rules ($k \in \{2, 3, \ldots, |S|\}$) have been generated, when a new DNA sequence must be classified, we compute the positive and negative probabilities for this sequence, as described in the previous subsection.

4. Experiments

In this section we provide experimental evaluations of the algorithms described in Section 3.

4.1. **Case study.** The data set we used to test the performances of the *PCRAR*-derived [2] algorithms is the one that was used in [2]. It is entitled "E. coli promoter gene sequences (DNA) with associated imperfect domain theory" and it was taken from the UCI Repository [3]. The data set is composed of 106 DNA sequences. Half of these represent positive instances, i.e. they contain promoter regions, while the other 53 are negative instances. As mentioned before, the relation definition and data pre-processing phases of the algorithm remain unchanged, only the training and testing phases being modified, as described in the Section 3.

The algorithms are compared by examining the classification accuracies and validation times and analysis for each one are made with different values for the confidence threshold: $\{0.4, 0.42, 0.45, 0.47, 0.48, 0.5, 0.52, 0.55, 0.6\}$. The minimum support threshold is fixed at 0.9. As described in [2], in order to decrease the number of considered attributes, we eliminate those attributes that have a very small correlation with the target output - whose correlation value is below a small positive threshold ϵ. To identify the optimal value of the threshold ϵ, a grid search method is applied, for each algorithm. The chosen values for ϵ are: $\{10^{-3}, 5 \cdot 10^{-3}, 10^{-2}, 5 \cdot 10^{-2}\}$. For each value of ϵ a cross-validation using a "leave-one-out" methodology is performed during the training phase of every algorithm, the best value of the threshold being indicated by the best accuracy (smaller error) obtained. We mention that the experiments were carried out on a PC with an Intel Core i5 Processor, at 2.53 GHz with 4 GB of RAM and that the validation time includes the computation time of the grid search procedure.

4.2. **Comparative results.** In the following the extensions of the *PCRAR* classifier [2] introduced in Section 3 will be referred to using the following abbreviations:

- *BRPC* - Binary Rules, with Probability computation based on the Confidence of the rules (Subsection 3.1)

- *KRPC* - K-length Rules, with Probability computation based on the Confidence of the rules (Subsection 3.2)

First observation we make is concerned to the number of generated relational association rules (for the entire positive and negative data sets), for each of the three algorithms. As both *PCRAR* [2] and *BRPC* determine binary rules, that only depend on the input data set, it is obvious that these two algorithms will always generate the same number of rules, for a certain value of the confidence threshold and of ϵ. However, the number of rules generated by *KRPC* will be significantly greater, as this algorithm determines rules of any length, starting from the set of generated binary rules. The maximum possible length for a k-length rule is $k = 57$ (the number of attributes in an instance), but the maximum length of rules that were actually generated was $k = 8$, for the confidence thresholds 0.4 and 0.42. Clearly, in all three cases, the number of rules increases as the confidence threshold decreases.

As reported in [2], the best result for *PCRAR*: **104** correctly classified instances, out of 106 instances (which means an error of 0.018867) was obtained for a confidence threshold of 0.4 and for $\epsilon = 10^{-2}$. The best result obtained by *BRPC* is the same: **104** out of 106, for the same minimum confidence, but it was obtained for $\epsilon = 10^{-3}$. *KRPC*, on the other hand, has proven a worse performance, as the best obtained result is **97** correctly classified instances, out of 106 (an error of 0.084905), for a confidence of 0.6 and with $\epsilon = 10^{-3}$. Figure 1 illustrates a comparison of the accuracies obtained by these three algorithms, for the minimum confidence 0.4 and for $\epsilon = 10^{-2}$ and $\epsilon = 10^{-3}$.

From Figure 1 we can observe that *BRPC* outperforms *PCRAR* [2], for $\epsilon = 10^{-3}$, which means that it is also important to consider the confidences of the rules in the classification process. Regarding the accuracies obtained by *KRPC*, we can observe that they are worse than those obtained by the other two algorithms. The higher classification error that appears in the case of *KRPC* may be due to the fact that the number of generated rules is significantly higher than in the other cases and consequently even if an instance verifies a certain number of stronger rules (with a high confidence), there may still remain a very large number of unverified rules (whose confidences, even if smaller, when summed up, could far exceed the sum of the confidences of the verified rules). Another thing we need to mention about the *KRPC* is that the accuracies it obtains increase with the confidence threshold. This is normal, because as the confidence threshold increases, the number of generated rules decreases and therefore the problem that we mentioned above is less likely to appear.

As the training time is also a relevant feature for comparing different classifiers, we will now refer to the running times of the algorithms. It is important

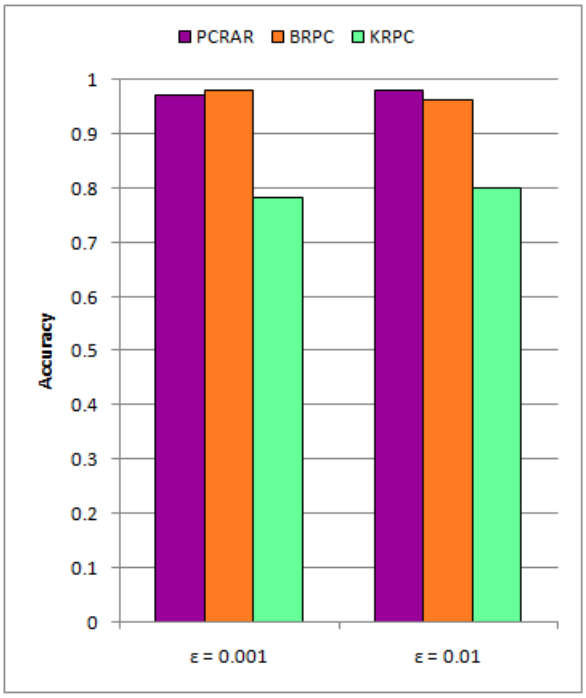


FIGURE 1. Comparative results: confidence threshold = 0.4

to know that these are actually validation times, i.e. overall times in which each version of the *PCRAR* classifier [2] performs the validation (this including the training time). We mention that both the algorithms that only consider binary rules have very low computational times, while the one that generates rules of any length clearly runs much slower. *PCRAR* [2] and *BRPC* have similar running times for all tested confidence thresholds: except for the minimum confidence 0.45, where they differ with approximately 3 seconds, for all the other values of the confidence thresholds, these two algorithms have at most 1.5 seconds difference. Figure 2 comparatively illustrates the running times for *PCRAR* [2] and *BRPC*. Although we cannot say that *PCRAR*'s [2] confidence-time function is monotonic (more specifically, decreasing), we may observe a decreasing tendency. On the other hand, the confidence-time function for *BRPC* is strictly decreasing. Concerning *KRPC*, its running times are significantly higher and, as expected, they decrease as the confidence threshold increases. The minimum validation time for this algorithm, for the confidence threshold of 0.6 is more than three times higher than that of the other two

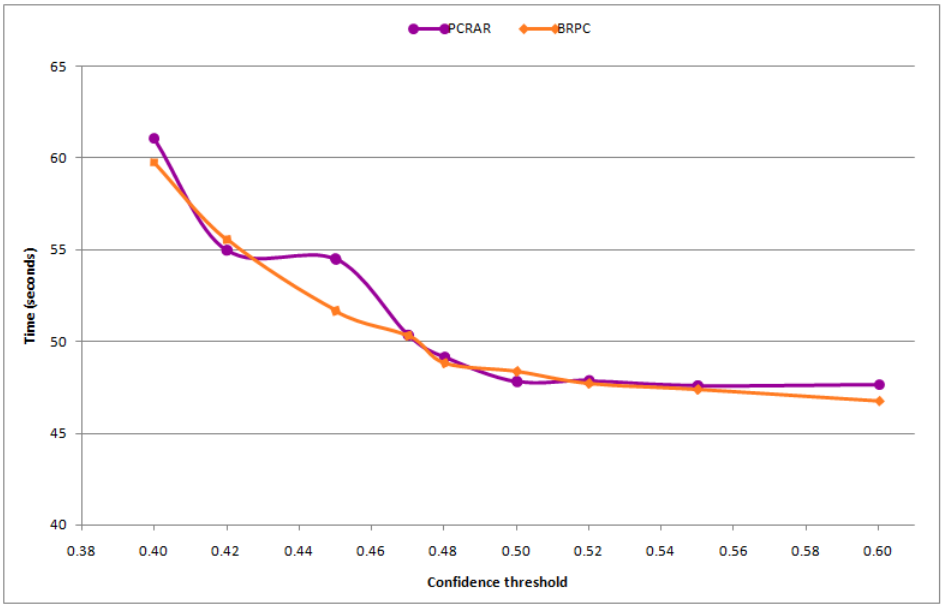


FIGURE 2. Comparative running times: *PCRAR* [2] and *BRPC*

algorithms. The confidence-time function that is generated for *KRPC* is exponential, as shown in Figure 3. In this figure, the Y axis represents hours, not seconds, as in Figure 2.

4.3. **Discussion.** We have experimented with three relational association rules based algorithms in order to obtain results for the promoter prediction problem. The original model - *PCRAR* [2] generates binary relational association rules in the training phase and then in the testing phase uses the number of rules that are verified by the new instance in order to classify it. The first extension that we introduced also generates binary relational association rules in the training phase, but then uses the confidences of the rules in order to classify a new instance. Finally, the second extension generates rules of any length, and uses the same method of classification as the first extension.

The obtained results demonstrate that the algorithms that generate and use only binary relational association rules perform better than the one generating rules of any length, both in terms of classification accuracy and validation times. This leads us to the conlusion that, for the considered problem, binary rules are sufficiently relevant in order to obtain a good classification of a DNA sequence as a *promoter* or a *non-promoter*.

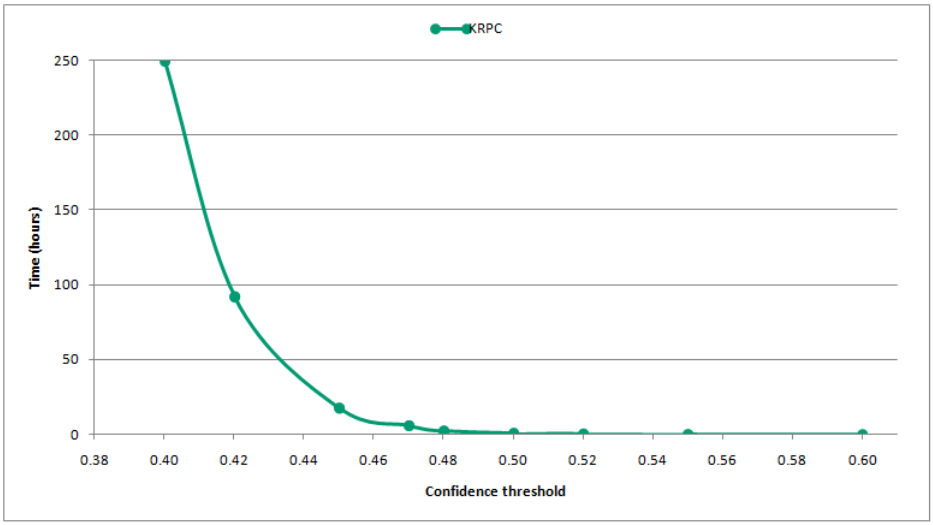


FIGURE 3. Running time for *KRPC*

5. Conclusions and Further Work

In the present study we introduced two extensions to a relational association rules based classification model for the problem of promoter sequences prediction and we experimentally evaluated and compared the three algorithms.

The algorithms were tested on a data set containing 106 *E. coli* DNA sequences [3], among which 53 contained promoter regions (positive instances) and 53 did not (negative instances). The tests showed that the two algorithms that generate only binary rules obtain very good performances, *better* than those reported by all other classifiers already applied in the literature for promoter sequences recognition and they need very low training times (less than two minutes). However, the third algorithm proved to obtain worse accuracies, when compared to the other two relational association rules classifiers, but still, when compared to the other classifiers already applied in the literature (see [2]), it is the *third best*. The drawback of this last algorithm is that its training times are very high.

Further work will be made in order to improve the accuracy of the relational association rules based classifiers by using supervised learning to identify the most appropriate values for the confidence threshold and for the attribute elimination threshold ϵ, i.e. the values that minimize the classification error as well as the execution time. We will also focus on hybridizing this classification

model, by combining it with other machine learning based predictive models [6].

ACKNOWLEDGEMENT

I thank my thesis advisor, Professor Gabriela Czibula for the ideas and suggestions for this paper. This work was possible with the financial support of the Sectoral Operational Programme for Human Resources Development 2007-2013, co-financed by the European Social Fund, under the project number POSDRU/107/1.5/S/76841 with the title Modern Doctoral Studies: Internationalization and Interdisciplinarity.

References

[1] Câmpan, A., Serban, G., Truta, T. M., Marcus, A., *An Algorithm for the Discovery of Arbitrary Length Ordinal Association Rules*, In DMIN'06, The 2006 International Conference on Data Mining, Las Vegas, USA, 2006, pp. 107–113.

[2] Czibula, G., Bocicor, M. I., Czibula, I. G., *Promoter Sequences Prediction Using Relational Association Rule Mining*, Evolutionary Bioinformatics **8**, 2012, pp. 181-196.

[3] Frank, A., Asuncion, A., *UCI Machine Learning Repository*, University of California, Irvine, School of Information and Computer Sciences, 2010, URL:http://archive.ics.uci.edu/ml.

[4] Han, J., *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers Inc., 2005, San Francisco, CA, USA.

[5] Marcus, A., Maletic, J. I., Lin, K. I., *Ordinal association rules for error identification in data sets*, Proceedings of the tenth international conference on Information and knowledge management, CIKM '01, ACM, 2001, pp. 589–591.

[6] Mitchell, T. M., *Machine Learning*, New York: McGraw-Hill, 1997.

[7] Saenger, W., *Principles of Nucleic Acid Structure*, Springer-Verlag, 1984.

[8] Serban, G., Câmpan, A., Czibula, I. G., *A Programming Interface for Finding Relational Association Rules*, International Journal of Computers, Communications and Control **I/2006**, Proceedings of the International Conference on Computers, Communications and Control, ICCCC 2006, Oradea, 2006, pp. 934-944.

[9] Tan, P. N., Steinbach, M., Kumar, V., *Introduction to Data Mining, (First Edition)*, Addison-Wesley Longman Publishing Co. Inc., 2005, Boston, MA, USA.

[10] Tung, N.T., Yang, E., Androulakis, I.P, *Machine Learning Approaches in Promoter Sequence Analysis*, In Machine Learning Research Progress, Nova Science Publishers, Inc, 2008.

Babeş-Bolyai University, Department of Computer Science, 1, M. Kogălniceanu street, 400084 Cluj-Napoca, Romania

E-mail address: iuliana@cs.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

CEPSTRAL-BASED SPEAKER RECOGNITION

BALŢOI IULIA-MONICA, TODEREAN ANDREEA-MARIA, AND STERCA ADRIAN

Abstract. This paper presents a simple speaker identification and classification algorithm. The algorithm uses features from the frequency domain and simple Euclidian distance for comparison. More specifically, the features for each voice sample are 5 pitch estimators constructed using cepstral coefficients. The algorithm was tested in a trial test with a collection of 10 speakers and achieved acceptable results.

1. Introduction

Human speech is a complex signal and this complexity is due to the large number of characteristics of human speech which can be viewed on different levels: acoustic, semantic, linguistic, psychological. Every person has an unique voice and even when the same person speaks the same words, the resulting sounds are not identical. The field of speech analysis has received much attention from the scientific community since the 60s up to the present day. Among important directions in speech analysis research are speech recognition and speaker recognition. Speech recognition refers to translating a an utterance into computer text. Important applications for speech recognition are voice command interfaces for mobile phones or other electronic devices, providing comfort for persons with disabilities or a more natural way of recording text for writers etc. As examples of commercial speech recognition software we can name SIRI of Apple iOS [4], Google Voice Search [5] or Windows Speech Recognition integrated in Windows Vista and Windows 7.

Speaker recognition is the activity of recognizing the person who is speaking. Speaker recognition technology takes two forms, speaker verification and speaker identification. While speaker verification means discovering the best match of an unknown speaker's identity from a list of known speakers (i.e.

Received by the editors: May 10, 2012.

2010 *Mathematics Subject Classification.* 94A12, 68U99.

1998 *CR Categories and Descriptors.* H.5.5 [**Information Systems**]: Information Interfaces and Presentation – *Sound and Music Computing*; I.5.4 [**Computing Methodologies**]: Pattern Recognition – *Applications*.

Key words and phrases. speaker recognition, voice recognition, cepstral-based sound analysis.

comparing one voice sample to voice samples of other speakers), speaker identification means verifying that a speaker is who he/she claims he/she is (i.e. comparing the speaker's voice sample to a previously recorded verified voice sample). Speaker recognition, whether verification or identification can be text-dependent (where the speaker is asked to say a specific text and the program can take advantage of the phonetic invariability of the voice samples) or text-independent (in which case the speaker can say any phrase and it is much harder for the program to find similarities between voice samples of several speakers). Applications of speaker recognition are found in forensics, microphone surveillance and various forms of authenticating services.

Related to speech and speaker recognition, in the general field of sound recognition, is music recognition which has several forms: melody recognition like performed by the Shazam software [6] music genre classification [7] or instrument separation.

The paper continues with sound recognition fundamentals in the following section, then section 3 describes our speaker recognition algorithm which is evaluated in section 4 and the paper ends with conclusions in section 5.

2. Sound recognition fundamentals

Every speaker recognition system has two components: feature extraction and classifier or speaker model. The feature extraction part refers to extracting data from the raw speech signal that identifies and differentiates the speech signal among other sound signals. Feature extraction is inevitable because a time-domain signal contains too much redundant data to use it directly for classification. Good features should encapsulate the main energy of the signal and should not contain redundant information. They also should not exhibit to much variability when extracted from another voice sample of the same speaker.

In order to apply statistical techniques on the speech signal, the signal is usually separated into frames of several tens of milliseconds long, so that the signal in a frame becomes quasistationary. Then each frame is usually multiplied by a window function (e.g. Hamming window, Hanning window etc.) to reduce the spectral leakage from applying the Discrete Fourier Transform on a finite interval.

One of the most used features are DFT coefficients. The Discrete Fourier Transform [8] of a speech signal draws the spectral envelope of that signal. Usually, only the magnitude of the spectrum is used and the phase information of the DCT coefficients is ignored.

Another much used feature is cepstrum and cepstral coefficients. The Cepstrum [8] is calculated by taking the Fourier Transform of a signal, then

absolute value of the coefficients, the logarithm and finally, the Inverse Fourier Transform. The resulting complex numbers are cepstral coefficients. Cepstrum has very good information-packing properties and the coarse spectral shape is modeled by the first cepstral coefficients, so not all coefficients from the frame must be considered. Cepstrum is also useful because a convolution of two signals in the time domain is equivalent to an addition of their cepstrum in the frequency domain.

Mel-frequency cepstral coefficients(MFCC) are another useful feature for speech recognition. MFCC are very similar to normal cepstral coefficients, but they approximate better the human auditory system's response due to the mel scale.

Other used features are linear frequency cepstral coefficients [9].

The classifier or the speaker model can be nonparametric where the classifier gets two feature vectors and it determines directly (without further tests) the similarity between them or it can be parametric where prior to determining the similarity between two feature vectors, the speaker model must be trained so that various parameters of the model can be fine-tunned for the specific speaker. As example of nonparametric classifiers we mention Dynamic Time Warping (DTW) and Vector Quantization (VQ) [1] and as examples of parametric speaker models we note Gaussian Mixture Models [11] and Hidden Markov Models [12]. Dynamic Time Warping is an algorithm that measures the similarity between two vectors of different time dimensions so that the similarity result is independent of small non-linear variations of the vectors in the time domain [1]. In Hidden Markov Models for speaker recognition, a Hidden Markov model is trained to match the speech utterance to some previously known utterance. In the training phase the model's parameters are adjusted so that they maximize the probability that the model outputs the training data.

For a very good overview of speaker recognition research please see [10].

3. Cepstrum-based algorithm for speaker recognition

The speaker recognition algorithm we present and evaluate in this paper uses the cepstrum transformation as the basis for the feature selection process and a simple euclidean metric for classification and matching. The algorithm is not text-depending meaning that it does not require the speaker to pronounce a specific phrase (although in the evaluation section, we have tested it using a specific text phrase). In order to recognize a speaker from a set of previously recorded speakers, the algorithm uses as input a voice sample of the unknown speaker and compares it to voice samples of the known speakers (previously recorded) and returns the one whose voice sample is the most similar to the

voice sample of the unknown speaker. The features extracted from the voice sample are pitch estimators of that voice. More specifically, we extract 5 such pitch estimators.

The workflow of the feature extraction phase of the algorithm is depicted in Figure 1. After the silence period is removed from the beginning of the voice sample, we take 5 sample windows from the voice sample, each sample window having 2048 samples and having a 50% overlap period. The purpose of choosing 2048-long sample windows is to have a quasistationary signal in a single sample window and we use 5 such sample windows in order to capture information than is not contained in the first 2048-long sample window. Each of the 5 sample windows is then passed through a Hamming window and then 2048 cepstral coefficients are determined from each sample window. The Hamming window is used to reduce the spectral leakage generated by applying the Fourier transform to a finite time interval of samples for which the period exhibits discontinuity at the edges of the interval. The cepstrum is used because it has good information-packaging properties and is very good for pitch determination of speech. From the resulting 5 cepstral coefficient windows, we take from each window the first 256 coefficients containing the frequency values with the highest energy of the sound signal (i.e. lowest frequency components) and we compute the maximum from those 256 cepstral coefficients. This maximum is an estimator of the pitch of the initial 2048-long sample window. In the end, we obtain 5 maximal cepstral coefficients from each of the 5 sample windows.

In order to compare two voice samples, we first compute the 5 aforementioned features (i.e. 5 maximal cepstral coefficients) for each voice sample and compute the Euclidean distance between those 2 vectors of 5 features each. If the Euclidean distance is bellow a threshold, then the two voice samples are similar. The overall algorithm is outlined bellow:

Algorithm isSimilar(*voicesample* source, *voicesample* candidate):
src_features[] = getFeatures(source); //get the 5 source features
cand_features[] = NULL; // initialize vector of candidate features
$(w_1, w_2, w_3, w_4, w_5)$ = getFiveSampleWindows(candidate);
for i=1 to 5 do
Hamming(w_i); // apply Hamming window on each sample window
Cepstrum(w_i); // compute Cepstrum
f = $max_{k=1..256} w_i[k]$; // get max from the first 256 cepstral coef.
cand_features[i] = f; // add the feature to the candidate feature set
end for;

```
if EuclidDistance(src_features, cand_features) ≤ threshold
    then return true;    // voice samples are similar
    else return false;
end if;
```

4. Evaluation of the algorithm

In order to evaluate the speaker recognition algorithm we have recorded voice samples from 10 speakers, 4 women and 6 men [2, 3]. Voice samples were recorded at a sample rate of 8000 Hz and 16 bit quantization. Each speaker was asked to say "salut" twice. The first voice sample of each speaker was saved in a database and the second voice sample was used to compare it against all voice samples stored in the database. The speaker recognition algorithm was implemented in the Java programming language. The results are shown in Table 1. The first column of each line from the table depicts the ID of the speaker together with his/her voice sample (i.e. the second voice sample for all speakers) that is compared against the first voice sample of each of the 10 speakers shown in the top line of the table. In general, on line i, column j the similarity between the second voice sample of speaker i and the first voice sample of speaker j it is shown. The bolded numbers from each line shows the sample voice most similar to the one used for the current line in the table.

We can see from the table that the algorithm has an error rate of 30%. This error rate is good, but not great, but if we look more careful at the table, we see that the algorithm was very close to correctly identify speakers 5 and 8 (i.e. the difference between their own voice samples is very close to the minimum difference found). There were also several other tests performed, considering only the first 128 cepstral coefficients, but the algorithm depicted in Figure 1 was the most successful from the ones we have tested.

5. Conclusions and Future work

We have presented in this paper an algorithm for speaker identification and classification. The algorithm uses features from the frequency domain for classification, cepstral coefficients to be more specific. For the classification part, is uses a simple Euclidean metric. To assess the effectiveness of the algorithm we have tested it using 10 speakers and found an identification error of 30%. In order to get a more precise assessment, we should test the algorithm

TABLE 1. Evaluation results

-	S 1.1	S 2.1	S 3.1	S 4.1	S 5.1	S 6.1	S 7.1	S 8.1	S 9.1	S 10.1
S 1.2	**0.023**	0.108	0.078	0.051	0.063	0.261	0.041	0.057	0.061	0.034
S 2.2	0.087	**0.053**	0.129	0.106	0.115	0.271	0.082	0.065	0.118	0.056
S 3.2	0.082	0.152	**0.039**	0.081	0.051	0.268	0.081	0.105	0.065	0.105
S 4.2	0.043	0.138	0.091	**0.023**	0.048	0.325	0.087	0.097	0.040	0.087
S 5.2	0.056	0.150	0.090	**0.033**	0.049	0.320	0.085	0.098	0.041	0.106
S 6.2	0.034	0.235	0.187	**0.024**	0.224	0.085	0.187	0.206	0.230	0.086
S 7.2	0.035	0.105	0.071	0.039	0.036	0.281	**0.030**	0.040	0.041	0.141
S 8.2	0.062	0.086	0.069	0.077	0.062	0.266	**0.031**	0.035	0.079	0.067
S 9.2	0.053	0.141	0.086	0.042	0.040	0.291	0.074	0.084	**0.016**	0.105
S 10.2	0.042	0.124	0.078	0.082	0.071	0.321	0.072	0.056	0.043	**0.025**

on a larger database of speakers. Also, the classification phase of the algorithm is rather simplistic and can be improved by using learning techniques.

6. Acknowledgments

This work was partially supported by CNCSIS-UEFISCSU, through project PN II-RU 444/2010.

References

[1] S. Theodoridis, K. Koutroumbas, *Pattern Recognition*, Academic Press, 4th edition, 2008.
[2] Baltoi Iulia, *Voice Recognition in the Frequency Domain*, Diploma thesis, Babes-Bolyai University, 2011.
[3] Toderean Maria, *Sound Classification in the Frequency Domain*, Diploma thesis, Babes-Bolyai University, 2011.
[4] ***, *Speech Interpretation and Recognition Interface*, http://www.apple.com/iphone/features/siri.html.
[5] ***, *Google Voice Search*, http://www.google.com/mobile/voice-search.
[6] ***, *Shazam*, http://www.shazam.com.
[7] G. Tzanetakis, P. Cook, *Musical Genre Classification of Audio Signals*, in IEEE Transactions on Speech and Audio Processing, vol. 10, no. 5, 2002.
[8] A. V. Oppenheim, R. W. Schafer, *Discrete-Time Signal Processing*, 3rd edition, Prentice Hall, 2009.
[9] Zhou X., Garcia-Romero D., Duraiswami R., Espy-Wilson C., Shamma S., *Linear versus Mel-Frequency Cepstral Coefficients for Speaker Recognition*, in Automatic Speech Recognition and Understanding Workshop, 2011.
[10] Kinnunen T., Li H., *An Overview of Text-Independent Speaker Recognition: from Features to Supervectors*, in Speech Communication, 2009.
[11] Reynolds D., Rose R., *Robust text-independent speaker identification using Gaussian mixture speaker models*, in IEEE Transactions on Speech and Audio Processing, vol. 3, no. 1, pp. 7283, 1995.

[12] Naik J., Netsch L., Doddington G., *Speaker verification over long distance telephone lines*, in Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, pp. 524527, 1989.

University of Bucharest
E-mail address: giuliasw@yahoo.com

Babes-Bolyai University, Computer Science Department
E-mail address: tais0549@scs.ubbcluj.ro

Babes-Bolyai University, Computer Science Department
E-mail address: forest@cs.ubbcluj.ro

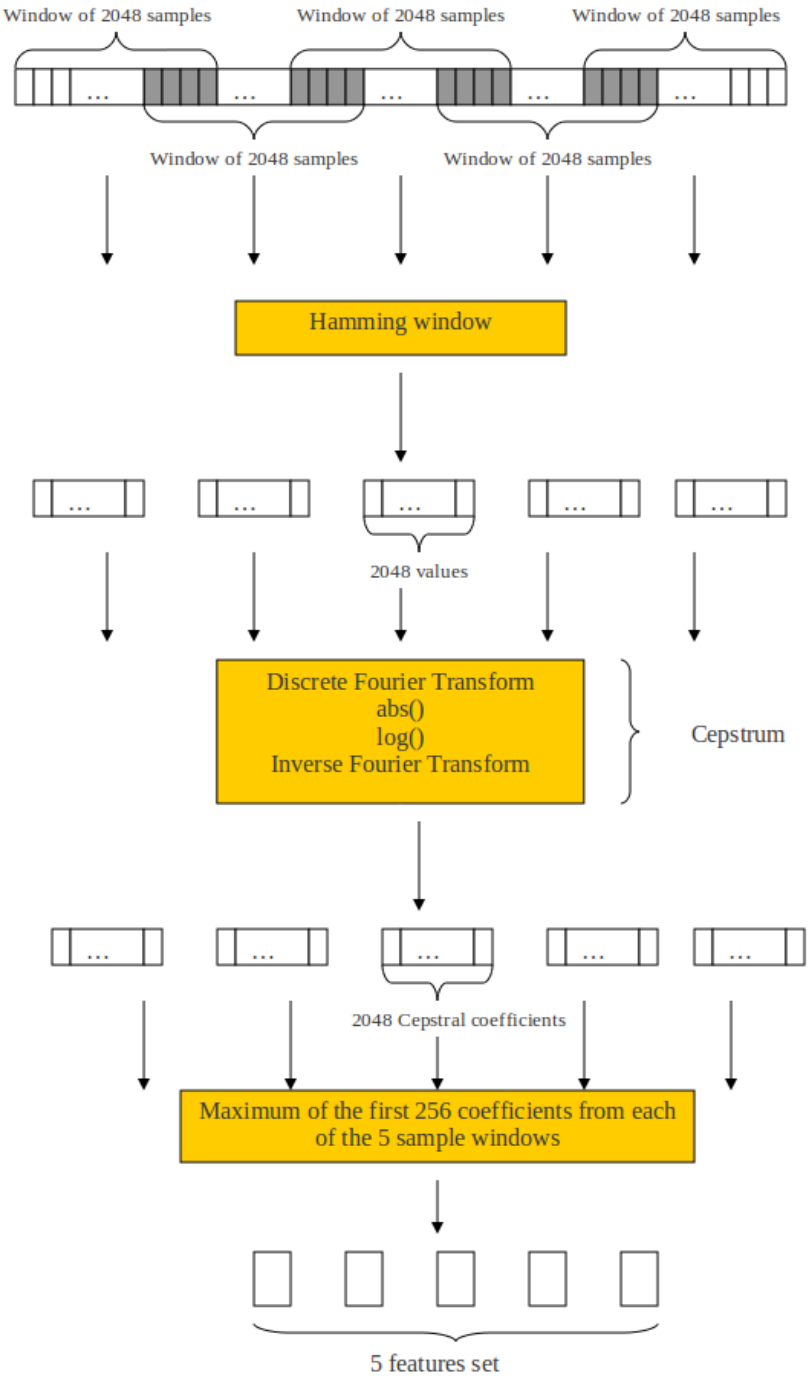


FIGURE 1. The feature extraction part of the speaker recognition algorithm

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

OPTIMIZATIONS IN PERLIN NOISE-GENERATED PROCEDURAL TERRAIN

ALEXANDRU MARINESCU

Abstract. The following article wishes to be the first of a series focused on different aspects involving procedural generation, with its ultimate goal being that of building an entire realistic 3D world from a single number (known in literature as the "seed"). It should allow for free exploration and render at interactive frame rates on mid to high-end graphics hardware. To begin with, we will discuss the manner in which we have generated procedural landscapes and some techniques we have borrowed from rendering algorithms in order to optimize the terrain generation and drawing process. Regarding the rendering framework, we have gone for Microsoft XNA Game Studio, the key factors of our choice being its simplicity plus the fact that we can focus more on the structure and realization of the algorithms and less on implementation and API details. As in our previous work [10], we have considered that this outweighs its limitations and that the concepts presented here should very well fit any programming language/drawing API.

1. About Procedural Generation

We feel it is our duty to first familiarize the reader with the field of procedural generation which, surprisingly, pre-dates the era of computers. The idea behind procedural generation is quite simple: what if we could take a complex object (and in our context, by "object" we usually refer to textures, vertices, polygons and meshes) and approximate it in a coarser or finer manner (depending on needs and hardware) by a function or procedure. In other words, when discussing procedural generation we must investigate to what extent it is possible to parametrize an object [16, 15].

Received by the editors: May 30, 2012.

2010 *Mathematics Subject Classification.* 68U05.

1998 *CR Categories and Descriptors.* A.1 General Literature [**INTRODUCTORY AND SURVEY**]; I.3.7 Computing Methodologies [**COMPUTER GRAPHICS**]: Three-Dimensional Graphics and Realism – *Virtual reality* .

Key words and phrases. procedural generation, seed, perlin noise, terrain, heightmap, LOD, chunk.

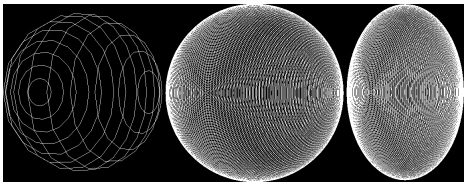


FIGURE 1. Different parametrizations for a procedural ellipsoid.

To better understand this process of parameter refinement, one should visualise a simple sphere. It is described by the coordinates of its centre in 3D world space and its radius, with all the points which create the "shell" of the sphere, (x, y, z) following the equation $(x - x_C)^2 + (y - y_C)^2 + (z - z_C)^2 = R^2$.

This is what we call the Sphere(Centre, Radius) parametrization. However simple it may seem, for the sphere to be drawn on the screen, the graphics pipeline must be fed primitives (triangles). This means that we must take discrete points (vertices) on the shell of the sphere, process which can return less or more vertices, depending on the division step by which we discretized the sphere. So now, apart from the centre and radius we have an extra parameter – a step increment which, for larger and smaller values yields, respectively, coarser and finer approximations of a sphere – the Sphere(Centre, Radius, StepDivision) parametrization. Additionally, we can substitute different increment values for the XY circle planes and for the Z axis, allowing for more control over the final shape. We can associate an elongation factor to a certain direction, effectively turning the sphere into an more general ellipsoid – Sphere(Centre, Radius, StepDivision, ElongationFactor, ElongationDirection). It is vital to notice that each new iteration is a generalization of the previous, so we can still generate everything we could before, while at the same time being able to provide new classes of objects (Figure 1).

At this point, the reader may already speculate that we can add an arbitrary number of parameters to our objects, each of them tweaking a different aspect of the final outcome, but still one should keep in mind some important aspects. The more parameters, the more control we have over the resulting object, but a more complex parametrization usually comes at the expense of speed and storage, since the resulting object needs to reside somewhere in memory.

The philosophy behind procedural generation states that, for the same combination of parameters we must have the exact same end result, making it possibly one of the most efficient methods of data compression to date. This turns it into an invaluable tool for game developers building extreme large scale environments, where the use of art assets sculpted by 3D modellers is

virtually impossible. Then again, this is the main reason why the authors have chosen it for the task of building a large scale realistic environment. We list here some sources of inspiration for our project: [1] – a free-roam racing game set in a post-apocalyptic environment with the total land surface estimated to an equivalent $14400km^2$, [3] – a galactic-scale god game with procedurally generated animations for creatures the player can model at his own will, [4] – a 96 kb FPS with all assets procedurally generated, from sound effects to weapon models, and last but not least, the many demo groups that push the limits of what can be done with a tiny amount of code – [2, 24]. The following sections will present an overview of the perlin noise generation algorithm and our attempt at implementing a synchronized CPU/GPU perlin noise generator (or PNG) focusing afterwards on extracting terrain heightmap data from the PNG.

2. Perlin Noise

Perlin noise is a type of gradient noise first developed by Ken Perlin in 1985 [17]. It relies on interpolating between the values of a lattice of random gradients to produce hyper-textures of pseudo-random appearance. The interpolant is usually a function having first and preferably second derivatives at endpoints 0. In our case, we used the improved interpolant $6t^5 - 15t^4 + 10t^3$ suggested by Ken Perlin in 2002 [18]. For further reading regarding the algorithm itself, please refer to [19, 5, 22, 14, 6].

The essential fact for us is that perlin noise is an application from $\mathbb{R}^n$ to $\mathbb{R}$, that is, for each point in an n-dimensional hyperspace, $PNG(x_1, x_2, ..., x_n)$ returns the evaluation of the perlin noise primitive in that point. Moreover, the PNG is consistent, meaning that, if $(x_1, x_2, ..., x_n) = (y_1, y_2, ..., y_n)$ then $PNG(x_1, x_2, ..., x_n) = PNG(y_1, y_2, ..., y_n)$, or the value returned by the perlin noise generator for the same point in space is always the same. This makes it very well suited to the context of procedural generation.

Because the PNG will be used extensively both by the CPU and the graphics pipeline, we have attempted to create a hybrid PNG that produces "similar" results, for up to 4-dimensional coordinate space. We say similar, because from a numerical point of view, the values evaluated in the same points by the CPU and GPU will never coincide, owing to different architectural implementations [6]. Moreover, the results are not even the same between different GPUs, but as we've stated before, we are interested in the visual appearance of the generated hyper-texture. We have not gone beyond 4D space because there was no interest for it in our application. Below one can find an outline of the hybrid CPU/GPU PNG:

- Initialize a random number generator with a given seed;

- Construct a 1D 256 permutation table containing a random permutation of the numbers 0 through 255;
- Construct a 2D 256*256 permutation table having 4 fields per element containing all the possible combinations of 2 indexes taken 0 through 255 and then hashed using the previous permutation table (all operations being performed modulo 256):

$$
\begin{aligned}
&\forall i = \overline{0, 255}, \forall j = \overline{0, 255} \\
&A = Permutation[i] + j, B = Permutation[i + 1] + j \\
&Permutation2D[i, j].Field_1 = Permutation[A] \\
&Permutation2D[i, j].Field_2 = Permutation[A + 1] \\
&Permutation2D[i, j].Field_3 = Permutation[B] \\
&Permutation2D[i, j].Field_4 = Permutation[B + 1]
\end{aligned} \tag{1}
$$

- Construct hashed permutation tables for the static 1D, 2D, 3D and 4D gradients. At this moment we have all we need for the CPU side.
- Generate (only once per application since these do not change) 1D textures coding the gradients using XNA's NormalizedByte4 surface format;
- Generate (every time the PNG is re-seeded) the 1D and 2D textures containing the permutation tables using XNA's Color surface format;
- Generate (again, with each re-seed) the 4 textures corresponding to the hashed permutation tables using XNA's NormalizedByte4 surface format;
- Finally, feed these textures to the graphics pipeline each time a shader utilizes procedural noise primitives.

The full documentation for the above mentioned surface formats and their use cases can be found at the following resources: [12, 13, 11, 9].

The higher dimensionality of gradient noise is interpretable, for example, 2D noise yields a texture which can be applied on a model with UV texture coordinates or used as a heightmap to generate 3D terrain (no folding of terrain surface such as caves or archways). 3D noise can be used to texture a 3D model without UV texture coordinates, based solely on the coordinates of its vertices, it can generate fully fledged 3D terrain, or it can be used to create animated textures, such as fire (with time being the extra 3-rd dimension). One can already realize why we have considered 4D noise sufficient for our purposes. We have also brought our personal contribution to this area, by fixing a minor bug in the HLSL implementation of 4D perlin noise from Nvidia GPU Gems 3 [14] and finding an optimization for 4D noise which uses fewer texture addresses and is consequently faster.

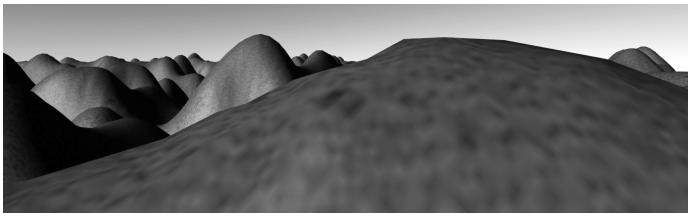


FIGURE 2. The lack of precision of a classic heightmap is responsible for the rough aspect of the terrain.

3. 3D Terrain Generation

The first thing that usually comes to mind when building a 3D world is the terrain. It is the basic support for other systems such as vegetation, water, buildings and so on. There are a number of ways in which we can import the terrain into our virtual reality application, heightmap-based generation being by far the simplest [7, 8]. Terrain can also be modelled as a 3D mesh with the use of 3rd party software, but this is out of the question for large scale environments.

A heightmap is normally a 2D grayscale image for which each pixel stores the height of the corresponding vertex (usually in the R channel). For example, a 64*64 image will yield 4096 vertices, with the width and height dimensions becoming X and Z coordinates respectively, possibly multiplied with some scale factors afterwards. The greatest drawback of this approach is that the color channels of the heightmap are very limited with respect to the range of values they can store (256 discrete values per channel). Visually, this lack of precision manifests itself as the "jagged" aspect of the terrain, which is noticeable even for small differences in height for adjacent vertices (Figure 2).

In order to fix this issue, we have used the PNG which we have discussed in the previous chapter to generate very smooth terrain. From an abstract point of view, we have considered our world, albeit it being in 3D, as "sitting" on a 2D lattice that extends infinitely on the X an Z axes in the XZ plane. All lattice points have integer coordinates, and the square delimited by 4 points will be denoted in our terminology as a "chunk". As a consequence, we have divided our "world" into chunks, each chunk being identified solely by the integer coordinates of its upper left corner (Figure 3). This idea was inspired by the manner in which [21] handles its large scale environment.

Afterwards we generate the actual terrain by splitting the intervals $(UpperLeft_X, UpperLeft_X + 1)$ and $(UpperLeft_Y, UpperLeft_Y + 1)$ into a desired number of subintervals. In our actual implementation we divided both intervals to obtain a smaller lattice of 64 by 64 vertices whose height can now

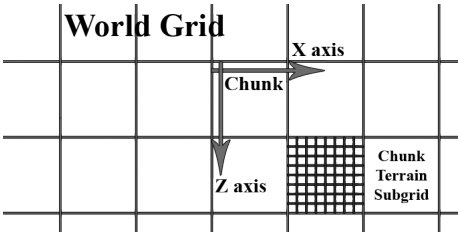


FIGURE 3. The lattice over which the 3D world resides.

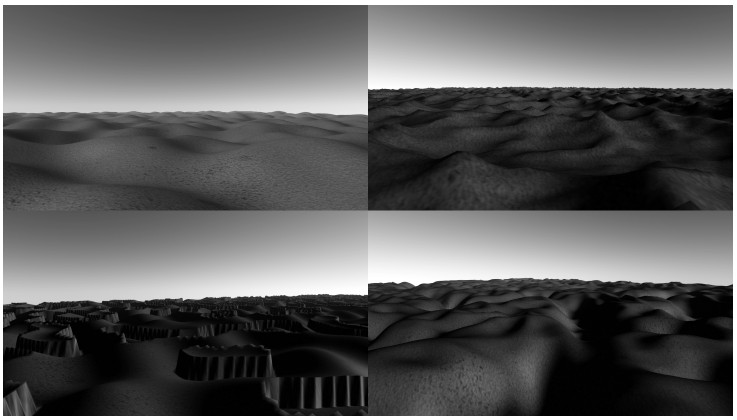


FIGURE 4. Examples of procedural terrain obtained using various combinations of perlin noise primitives. For each vertex, the normal component was computed based on the approach from [7, 8] and the tangent/binormal frame was calculated using the algorithm from [25] in order to apply more advanced lighting.

be determined by simply querying the value of the PNG at each given vertex (we already know the X and Z coordinates). Remark that, because of the properties of perlin noise and because we have split the 3D world in such a manner, the resulting terrain will be seamless. One should experiment with different combinations of noise primitives, visualize the outcome and try to understand the behaviour of various compositions of noise functions (Figure 4).

4. Optimizations

At this point there is still a lot of room for improvement and this section will present some of the steps we took in order to optimize the application,

some of them targeting the algorithmic side whilst others address efficiency of the rendering pipeline.

4.1. **Architectural Robustness.** We have already mentioned the chunking mechanism for generating our procedural world: each chunk is an abstract container, not only for terrain, but for everything that lies in that specific sector of our virtual world. In order to keep this high level of decoupling between chunks, all that should be publicly available to them at any moment must be only the PNG which is shared by the entire virtual world, and the coordinates of the upper-left corner which uniquely identify it. Since there are no cross-references between chunks, it is very easy to transition from single to multi-threaded chunk generation. Chunks inherit from XNA's DrawableGameComponent base class, which allows them to behave as standalone game components [23, 13, 11, 9]; when overriding the Update(GameTime) and Draw(GameTime) methods we have added a boolean flag which specifies whether the chunk is fully loaded and ready for rendering. If the world seed does not change, the chunk data does not change, so it is not necessary to generate the chunks with each run of the application. We have employed a serialization/deserialization type mechanism for saving generated chunks and loading those that have already been processed. Interesting enough, at this early phase of the application it is cheaper to always re-generate everything than to deserialize from disk, but it is expected that, as chunk data becomes larger and procedural content generation becomes more computationally expensive, saving and loading data will benefit the application.

4.2. **Tweaking the Rendering Process.** Considering these aspects, the virtual world becomes a collection of chunks. Notice that we do not need to draw all chunks with the same level of detail. Of course, chunks that are closer to the player camera must be drawn in higher detail than those further away. If this were the case of terrain sculpted by an artist, it would be quite difficult to obtain these different levels of detail, but with procedurally generated terrain, we simply apply the algorithm of splitting the lattice into smaller subintervals, but we will substitute a SkipFactor proportional to the required level of detail. To further exemplify, remember we split at 64 discrete points on each side; this would correspond to a SkipFactor of 1 and a level of detail (LOD) of 0 – the highest detail. Now, imagine we need an LOD of 1, this means a SkipFactor of $2^1 = 2$ which would correspond to the same lattice now split into 32*32 vertices. As the LOD increases, the number of vertices continues to halve, until it reaches 4 – a single quad (we cannot draw a single vertex). As a general rule, $SkipFactor = 2^{LOD}$ and thus $MaxLOD = \log_2 TerrainSize$. Since we have access to the terrain vertices, we can easily compute some auxiliary shapes that are extremely useful for

hidden surface removal. First of all we can compute the centroid of all the vertices for the terrain chunk data, which is done by taking their component-wise arithmetic mean: $G = \frac{1}{VertexCount} \sum_{i=1}^{VertexCount} Vertex_i$. Moreover, we can compute the BoundingBox and BoundingSphere surrounding the terrain as being the smallest box and respectively sphere that surround the entire chunk. This enables us to perform a simple camera frustum containment check and reject chunks outside the player's view, which also allows for a simple form of hardware occlusion query where the much lighter bounding boxes are drawn instead of the entire geometry and fully occluded chunks are also rejected.

By using the centroid of the terrain as a coarse identifier for the chunk, we can determine, based on the distance to the camera, which LOD to draw for the terrain. In the actual implementation we have found that

$$\max(\min(\lfloor \frac{DistanceToCamera}{e * TerrainDiagonal} \rfloor, MaxLOD), 0) \tag{2}$$

suits our purposes (where TerrainDiagonal is the diagonal of the terrain lattice scaled to world units). Most GPUs have what is known as an early depth test, meaning they can discard fragments from the pixel shader stage if another object has already been drawn in front of them. This allows us to speed up the rendering process by simply drawing closer (and presumably larger) chunks first, which is also done by taking into account the distance to the player camera. Furthermore, the GPU prefers drawing larger numbers of polygons in a small number of Draw calls to drawing fewer polygons with a high count of Draw invocations; it is best to gather the polygons from the furthest chunks into a single batch to be sent to the graphics pipeline [6, 7, 8].

One could implement simplex noise instead of perlin noise, which was also discovered by Ken Perlin later in 2001 [20] and has been proven to have a much lower complexity (n^2 compared to 2^n where n is the dimensionality of the noise function). Another useful aspect regarding generating chunks on the fly is that we can attach a camera predictor to attempt to guess in advance where the player might move next based on his previous actions and invoke the procedural generation mechanism even before the player reaches his destination.

5. Conclusions

In the closing chapter of this article we wish to point out our achievements so far, these being the implementation of a hybrid CPU/GPU perlin noise primitive generator and the realization of a large scale virtual world chunking mechanism at an abstract level, with optimized procedural terrain generation, having a pleasant visual appearance (Figure 5). Regarding the improvements

FIGURE 5. Our application running at little over 500 FPS 1366*768 fullscreen, on an Intel Core i5-460M 2.53 GHz CPU and Nvidia GeForce GTX 460M GPU system after all these optimizations have been applied. The world consists of 100 chunks.

over a similar architecture from [21] we should mention that their respective chunking system has voxels as the smallest unitary entity, effectively limiting the engine to drawing volumes made up of cubes, whilst our implementation allows for any type of surfaces/meshes. There have not been many attempts in the gaming industry at creating huge outdoor environments mainly because it is difficult to give the player a goal in such a world; ours has purely research value. Also, a transition to the state-of-the-art DirectX 11 is probable in the near future, and making use of its enhanced API and improved shader capabilities would surely benefit our application, but we feel that there are optimizations which can still be carried out in the older DirectX 9 version supported by XNA.

Finally, we hope that our work has raised the reader's interest on the exciting subject that is procedural generation and we wish to provide some directions for further development which we will also explore: the generation of a procedural sky – star fields, real-time atmospheric scattering, cloud cover formation, movement and dissipation, volumetric clouds; procedural vegetation (including aging of plants) and plant movement according to wind fields; procedural texture generation for different types of materials. As our application's expenses grow larger we will investigate the possibility of offloading all or some of the procedural computations to a dedicated server/web service. With the ever-increasing communication bandwidth we consider this might be feasible.

6. ACKNOWLEDGEMENTS

This research is supported by Grant No. 30068/19.01.2012, Procedural Generation: Building a 3D world from a single seed, funded by Babeş-Bolyai

University, Cluj-Napoca, Romania, and has been carried out under the tutorship of Assoc. Prof. Ph.D. Simona Motogna[1].

References

[1] Codemasters, Fuel, http://www.codemasters.com/uk/#/uk/fuel-uk/360/
[2] Conspiracy, Binary Flow, http://conspiracy.hu/
[3] Electronic Arts, Spore, http://eu.spore.com/home.cfm
[4] Farbrausch, .kkrieger, http://www.farb-rausch.de/
[5] R. Fernando, GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, Pearson Higher Education, 2004.
[6] GameDev, GameDev.net Developer Community, http://www.gamedev.net/
[7] R. Grootjans, Riemer's 2D & 3D XNA Tutorials, http://www.riemers.net/
[8] R. Grootjans, XNA 3.0 Game Programming Recipes: A Problem-Solution Approach, Apress, 2009.
[9] S. Hargreaves, Shawn Hargreaves Blog - Game programming with the XNA Framework, http://blogs.msdn.com/b/shawnhar/
[10] A. Marinescu, Achieving Real-Time Soft Shadows Using Layered Variance Shadow Maps (LVSM) in a Real-Time Strategy (RTS) Game, Studia Universitatis Babeş-Bolyai Informatica, 2011, p. 85-94.
[11] Microsoft, App Hub - XNA main developer portal for Windows Phone & Xbox 360, http://create.msdn.com/en-US/
[12] Microsoft, DirectX SDK June 2010.
[13] Microsoft, MSDN Library - Documentation for developers using Microsoft technologies and tools, http://msdn.microsoft.com/en-us/library
[14] H. Nguyen, GPU Gems 3, Addison-Wesley Professional, 2007.
[15] D. Oliver, FractalVision: put fractals to work for you, Sams, 1992.
[16] PCG, Procedural Content Generation Wiki, http://pcg.wikidot.com/
[17] K. Perlin, An image synthesizer, SIGGRAPH '85 Proceedings of the 12th annual conference on Computer graphics and interactive techniques, 1985, p. 287-296.
[18] K. Perlin, Improving noise, SIGGRAPH '02 Proceedings of the 29th annual conference on Computer graphics and interactive techniques, 2002, p. 681-682.
[19] K. Perlin, Making Noise, GDCHardCore, 1999, http://www.noisemachine.com/talk1/
[20] K. Perlin, Noise hardware, Real-Time Shading SIGGRAPH Course Notes, 2001.
[21] M. Persson, Minecraft, http://www.minecraft.net/
[22] M. Pharr, R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley Professional, 2005.
[23] A. Reed, Learning XNA 4.0: Game Development for the PC, Xbox 360, and Windows Phone 7, O'Reilly Media, 2010.
[24] RGBA, Elevated, http://pouet.net/prod.php?which=52938
[25] Terathon Software, C4 Engine, http://www.terathon.com/

Department of Computer Science, Babeş-Bolyai University, Kogălniceanu 1, 400084 Cluj-Napoca, Romania
E-mail address: mams0507@scs.ubbcluj.ro

[1] motogna@cs.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

AN INCREMENTAL APPROACH TO THE SET COVERING PROBLEM

RADU D. GĂCEANU AND HORIA F. POP

Abstract. The set covering problem is a classical problem in computer science and complexity theory and it serves as a model for many real-world applications especially in the resource allocation area. In an environment where the demands that need to be covered change over time, special methods are needed that adapt to such changes. We have developed an incremental clustering algorithm in order to address the set covering problem. The algorithm continuously considers new items to be clustered. Whenever a new data item arrives it is encapsulated by an agent which will autonomously decide to be included in a certain cluster in the attempt to either maximize its cover or minimize the cost. We have introduced the soft agent model in order to encapsulate this behaviour. Initial tests suggest the potential of our approach.

1. Introduction

The Set Covering Problem (SCP) is a classical problem in computer science and complexity theory and it serves as a model for many applications in the real world like: facility location problem, airline crew scheduling, resource allocation, assembly line balancing, vehicle routing, information retrieval etc. Let us consider a set X and a family F of subsets of X such that every element from X belongs to at least one subset from F. The set covering problem is the problem of finding a minimum number of subsets from F (or subsets of minimum cost) such that their union is the set X.

A straightforward solution is the greedy approximation algorithm [**?**]. This method selects at each step a set from F that covers most of the still uncovered elements. In [**?**] the set covering problem is addressed using an ant colony optimization algorithm together with a new transition rule. The authors have also used a look-ahead mechanism for constraint consistency checking such

Received by the editors: March 8, 2012.

2010 *Mathematics Subject Classification.* 68T05, 62H30.

1998 *CR Categories and Descriptors.* I.2.6 [**Computing Methodologies**]: Artificial Intelligence — *Learning*, I.5.3 [**Pattern recognition**]: Clustering — *Algorithms*.

Key words and phrases. Set covering problem, Incremental methods, Clustering, SCP datasets.

that new elements are added to the solution if they do not produce conflicts with the next element to be chosen.

The set covering problem can be formulated as a clustering problem where the within cluster sum of squared errors to be minimized corresponds to the cost associated to a certain set covering that needs to be minimal. We have developed an incremental clustering algorithm in order to address the set covering problem. The algorithm continuously considers new items to be clustered. Whenever a new data item arrives it is encapsulated by an agent which will autonomously decide to be included in a certain cluster in the attempt to either maximize its cover or minimize the cost.

The rest of the paper is structured as follows. In Section 2 the related work is presented. Section 3 contains the theoretical background. The proposed model is described in Section 4. The advantages and drawbacks of the approach together with some concluding remarks are presented in the closing Section 6.

2. Related work

The set covering problem is NP-hard and it has been addressed in many ways over time [**?**, **?**, **?**, **?**, **?**, **?**]. A greedy approximation approach [**?**] is a straightforward way to address this problem. This method selects at each step a set from F that covers most of the still uncovered elements (where F denotes a family of subsets of the given set X such that every element from X belongs to at least one subset from F).

In [**?**] the set covering problem is addressed using a genetic algorithm by using an n-bit binary string as the chromosome structure, where n is the number of columns from the SCP dataset. In order to mark that column i is in the solution the i^{th} bit is set to 1. The authors designed heuristic operators that transform invalid solutions (obtained after applying genetic operators) to valid ones. The binary tournament selection was chosen as the method for parent selection and for crossover the authors propose a so-called fusion operator taking into account both the structure and the relative fitnesses of the parents. A variable mutation rate specified in [**?**] is used arguing that the genetic algorithm is more effective. Computational experiments on a large set of randomly generated problems show that the genetic algorithm based approach is capable of producing high quality solutions.

In [**?**] the online version of SCP is considered. In the online version an adversary gives elements to the algorithm one by one. Whenever a new element is arriving, the algorithm has to cover it. The elements of X and the members of F are known in advance to the algorithm, but the set $X^{'} \subseteq X$ of elements

given by the adversary isn't and the task is to minimize the total cost of the sets chosen by the algorithm.

In [**?**] the set covering problem is addressed using an ant colony optimization algorithm together with a new transition rule. The authors have also used a look-ahead mechanism for constraint consistency checking such that new elements are added to the solution if they do not produce conflicts with the next element to be chosen.

In [**?**] a clustered variant of the SCP is defined. In the Clustered-SCP the subsets are partitioned into k clusters and a fixed cost is associated to each cluster. So the objective is to find a cover that minimizes the sum of subsets costs plus the sum of fixed cluster costs.

In [**?**] the authors introduce a new class of set covering heuristics, based on clustering techniques. They begin by partitioning the set of columns into clusters based on some column similarity measure and then they select the best column from each cluster. If the selected columns cover the set then this cover is pruned and the search stops here. Otherwise the partitioning is modified and the process is restarted. Experiments performed on randomly generated test problems indicate promising results.

3. Theoretical background

In machine learning, clustering is an example of unsupervised learning because it does not rely on predefined classes and class-labelled training examples. So it could be said that clustering is a form of learning by observation, rather than learning by examples. In data analysis, efforts have been conducted on finding methods for efficient and effective cluster analysis in large databases. The main requirements for a good clustering algorithm would be the scalability of the method, its effectiveness for clustering complex shapes and types of data, dealing with high-dimensional data, and handling mixed numerical and categorical data in large databases.

There is a great variety of clustering algorithms to choose from each with its own strengths and weaknesses. In [**?**] an incremental clustering algorithm is presented. Incremental clustering algorithms in general do not rely on the in-memory dataset and they build the solution gradually, with every new incoming data item. The idea behind is that it is possible to consider one instance at a time and assign it to existing clusters without significantly affecting the already existing structures. Only the cluster representations need to be kept in memory so not the entire dataset and thus the space requirements for such an algorithm are very small. Whenever a new instance is considered an incremental clustering algorithm would basically try to assign it to one of

the already exiting clusters. Such a process is not very complex and therefore the time requirements for an incremental clustering algorithm are also small.

An agent is an entity that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [**?**]. An agent that always tries to optimize an appropriate performance measure is called a rational agent. Such a definition of a rational agent is fairly general and can include human agents (having eyes as sensors, hands as effectors), robotic agents (having cameras as sensors, wheels as effectors), or software agents (having a graphical user interface as sensor and as effector). Usually agents coexist and interact forming Multi-agent Systems (MAS). In computer science, a MAS is a system composed of several interacting agents, collectively capable of reaching goals that are difficult to achieve by an individual agent or monolithic system.

The set covering problem is a classical problem in computer science and it serves as a model for many real world applications. Let us consider a set X and a family of subsets of X

$$F = \{S_1, S_2, \ldots, S_n\}. \tag{1}$$

The classical set covering problem is the problem of finding a minimum cardinality $J \subseteq \{1, \ldots, n\}$ such that

$$\bigcup_{j \in J} S_j = X. \tag{2}$$

The minimum cost set covering problem considers a cost c_j for each S_j and the problem is to find a cover for X and to minimize the sum of costs $\sum_{j \in J} c_j$.

4. Incremental SCP

In our model the input is an $m \times n$ incidence matrix A, where $m =\mid X \mid$ and each column corresponds to a set S_j with $j \in \{1, \ldots, n\}$. Each column j has a corresponding cost $c_j > 0$. We say that a column j covers a row i if $a_{ij} = 1$. Let x_j be a binary decision variable which has the value 1 if column j is chosen and 0 otherwise. Then the set covering problem can be defined as minimize (3) subject to (4) [**?**].

$$f(x) = \sum_{j=1}^{n} c_j x_j, \tag{3}$$

$$\sum_{j=1}^{n} a_{ij}x_j \geq 1, \forall i = \overline{1,n}, \tag{4}$$

Clustering can be seen as the problem of finding "meaningful" groups in data and a way to do this is by minimizing a certain objective function. The set covering problem can be formulated as a clustering problem in the following way: assign columns from A to clusters such that the function from (3) is minimized and the cluster is valid, i.e., the relation (4) holds.

We have developed an incremental clustering algorithm in order to address the set covering problem. The algorithm considers one instance at a time and assigns it to one of the existing clusters without significantly affecting the already existing structures. The algorithm continuously considers new items to be clustered. Whenever a new data item arrives it is encapsulated by an agent which will autonomously decide to join a certain cluster in the attempt to either maximize the cluster cover or minimize its the cost. These two objectives are rather conflicting and this brings a great deal of imprecision and uncertainty in the whole reasoning process. That is why we have employed *soft agents* in this matter. A *soft agent* is an intelligent agent that has to deal with imprecision, uncertainty, partial truth and approximation during its execution as a reactive agent or goal oriented agent or both.

Definition 4.1. *A soft agent is a function which assigns actions to state-reward pairs:*
$agent : (S \times \mathbb{R})^* \to A$,
where S represents the set of all possible states and A is the set of all possible actions an agent may choose from.

So, roughly speaking, a soft agent chooses its next action based on its previous experience, i.e., previous environment states and it receives a reward $r \in \mathbb{R}$ as a result of this choice.

The interaction between the agent and the environment is thus a sequence of environment state-reward pairs and actions:

$$h : (s_0, r_0) \xrightarrow{a_0} (s_1, r_1) \xrightarrow{a_1} \ldots \xrightarrow{a_{u-1}} (s_u, r_u) \tag{5}$$

We consider a state transformer function

$$env : S \times \mathbb{R} \times A \to \mathcal{P}(S \times \mathbb{R}) \tag{6}$$

in order to represent the effect of an agent's actions over the environment.

According to this definition environments are assumed to be history dependent. So the next state of an environment is determined by the action performed by the agent in the current state of the environment, the reward

and also by the earlier actions made by the agent. This behaviour is thus non-deterministic. In other words, the result of performing an action in some state is governed by uncertainty. Also we may notice that the agent's behaviour in this situation is highly *reactive* to the local changes in the environment. However using only purely reactive agents is not always a fortunate choice because they could easily get trapped in local minima. This is why soft agents are designed to also act *proactively*, ensuring that they attempt to accomplish what they are supposed to. Therefore in order to obtain an optimal performance from an agent it should be able to find the proper balance between its *reactive* and *proactive* behaviour.

The environment can be formally written as a triple

$$Env = \langle S, (s_0, r_0), env \rangle, \tag{7}$$

where S is the set of environment states, s_0 is the initial state, r_0 is the initial reward and env is the state transformer function.

Proposition 4.1. *If* $agent : (S \times \mathbb{R})^* \to A$ *is an agent,* $env : S \times \mathbb{R} \times A \to \mathcal{P}(S \times \mathbb{R})$ *is an environment,* s_0 *is the initial state and* r_0 *is the initial reward then the sequence:*

$$h : (s_0, r_0) \xrightarrow{a_0} (s_1, r_1) \xrightarrow{a_1} \ldots \xrightarrow{a_{u-1}} (s_u, r_u) \tag{8}$$

is a possible history of the given agent in the given environment if and only if the following conditions hold:

(1) $\forall u \in \mathbb{N}, a_u = agent(((s_0, r_0), (s_1, r_1), \ldots, (s_u, r_u)))$
(2) $\forall u \in \mathbb{N}$ *such that* $u > 0, (s_u, r_u) \in env(s_{u-1}, r_{u-1}, a_{u-1})$

The set of all such possible histories (8) will be denoted with H.

In order to specify the agent's *proactive* behaviour a certain performance measure needs to be specified and in the case of soft agents fitness functions are associated to states of the environment. The agent has to maximize its fitness. The fitness function is defined in the following way: $F : H \to \mathbb{R}$, where H is the set of histories. So a fitness function associates a real value to every history.

The task that an agent has to accomplish is to maximize its fitness so an optimum is then reached for:

$$\arg\max_{agent} \sum_{h \in H(agent, Env)} F(h) P(h | agent, Env), \tag{9}$$

where $P(h|agent, Env)$ denotes the probability that the history h occurs when the *agent* is placed in environment Env.

While the fitness function evaluates what is good on a long term, the reward function evaluates the quality of a certain state. So the reward function shows which would be the good and the bad actions to be taken from a certain state. The reward function could be used to modify an agent's policy π, i.e, the agent's behaviour. Thus the reward function is defined as $Q : S \times A \to \mathbb{R}$ and maps a real value, a reward, to a state-action pair.

For the set covering problem we define the fitness of an agent a_i given a cluster c_k in the following way:

Definition 4.2. *The fitness of an agent a_i given a cluster c_k is:*

$$f(a_i, c_k) = \frac{\mid rows(a_i) - rows(c_k) \mid}{m}, \tag{10}$$

where $rows(a_i)$ denotes the set of rows covered by agent a_i, $rows(c_k)$ denotes set of rows covered by cluster c_k and m is the total number of rows.

The algorithm considers one column at a time and encapsulates it in an agent. In the first part an initial valid cluster will be built using a greedy approach (a cluster is valid if it covers the considered set). After this step every newly considered agent will decide weather to try to maximize the cover of one of the existing clusters or to minimize the cost using the following control function: $f^{\lambda}(a_i, c_k)$. The parameter λ is initialized with 1 and is increasing in time thus leading to agents that act upon minimizing the cost rather than maximize the cover. A pseudo-code of the algorithm is sketched in Algorithm 1.

Algorithm 1 Incremental SCP

```
1: initialize parameters
2: find a proper cluster c_0 by randomly selecting agents
3: C ← C ∪ {c_0}
4: while condition() do
5:     U ← U ∪ {createAgent()}
6:     while U ≠ ∅ do
7:         if reactive(a_i) then
8:             assign a_i to a non-valid cluster c_k or create a new cluster
9:         else
10:            {S_j, c_k} ← tryReplace(a_i)
11:            U ← U ∪ S_j
12:        end if
13:    end while
14:    U ← U ∪ discardWorseCluster()
15:    update parameters
16: end while
```

The algorithm continuously receives new items (columns) to be clustered and an agent encapsulates each item. The agent is placed in the collection U of unclustered items. Starting from line 6 the algorithm repetitively considers agents from the collection U. The agent decides to behave in a reactive or proactive manner based on its control function. If it decides to behave reactively, i.e., maximize the cover then the agent attempts to find a non-valid

cluster to be included in. If no such cluster is found then a new cluster is created containing this agent. The new cluster is added to C and a_i is removed from U. A cluster is valid if all the rows are covered (4). If on the other hand the agent wants to minimize the cost then it will try to replace agents from a valid cluster c_k. If the replace operation took place then the set of replaced agents S_j is added to the unclustered collection. After all agents from U have been added to some clusters the cluster with the worse cost is discarded and its agents are added to the unclustered collection. Parameters like λ or acceptable cost threshold may be updated at this point. The λ parameter influences at any moment the decision of behaving one way or another, i.e., maximizing cover or minimizing the cost. It may be adaptively updated — when the solution stabilizes for a certain number of iterations, λ is reinitialized. The update strategy of the λ parameter influences the efficiency of the algorithm. Future work will focus on the study of fine tuning the λ parameter in order to find the right balance between the two objectives. The whole process starts over from line 4 and ends when some condition is met (like a good-enough solution is found or a certain number of iterations have been completed).

5. Case studies

We have conducted experiments on five datasets from the $OR - Library$ [**?**]: *scp*410, *scpa*1, *scpa*2, *scpa*3, *scpa*4. The $OR - Library$ is a collection of test data sets for a variety of operation research (OR) problems, including SCP. The *scp*410 dataset has 200 rows and 1000 columns in the following format: number of rows (m), number of columns (n), the cost of each column $c(j), j = 1, \ldots, n$ and then for each row $i(i = 1, \ldots, m)$: the number of columns which cover row i followed by a list of the columns which cover row i. The other datasets (*scp*410, *scpa*1, *scpa*2, *scpa*3, *scpa*4) have 300 rows and 3000 columns and they use the the same format which was described above.

We have obtained near-optimum solutions as shown in Table 1 and in Figure 2.

Reference [**?**] mentions that in case of the *scp*410 dataset the optimum cost is 514 (see Table 1). After 5000 iterations we have obtained the cost 569. By one iteration we mean the execution of the loop from *line* 4 of Algorithm 1. At each iteration we read one new agent until all agents have been read. The *scp*410 dataset may produce at most 1000 agents. At iteration 1000, i.e., by the the time all agents are loaded we have already obtained the cost 774. So at the beginning the algorithm finds good solutions fast. Unfortunately, in time, it slows down in improving the cost and it only reaches near optimum solutions (see Figure 1). Nevertheless, compared to the ACS [**?**] and $ACS + FC$ [**?**] algorithms we have obtained better results in case of the *scp*410 dataset.

Dataset	m	n	Opt	AS	ACS	AS+FC	ACS+FC	IncSCP
scp410	200	1000	514	539	669	556	664	569
scpa1	300	3000	253	592	348	288	331	372
scpa2	300	3000	252	531	378	285	276	486
scpa3	300	3000	252	473	319	270	295	382
scpa4	300	3000	234	375	333	278	301	370

TABLE 1. m — number of rows (constraints); n — number of columns (decision variables); Opt — the best known cost value (taken from [**?**]); result when applying Ant algorithms, AS and ACS, and combining them with forward checking (taken from [**?**]); IncSCP — our result.

We have also obtained near optimum results in our tests on datasets *scpa*1 to *scpa*4, outperforming the *AS* [**?**] algorithm (see Table 1).

Comparative evaluation of the execution of our approach and the results reported in [**?**] is shown in Figure 2. In the case of *scp*410 dataset we outperform the *ACS* [**?**] and $ACS + FC$ [**?**] algorithms and in the *scpa* datasets we outperform the *AS* [**?**] algorithm.

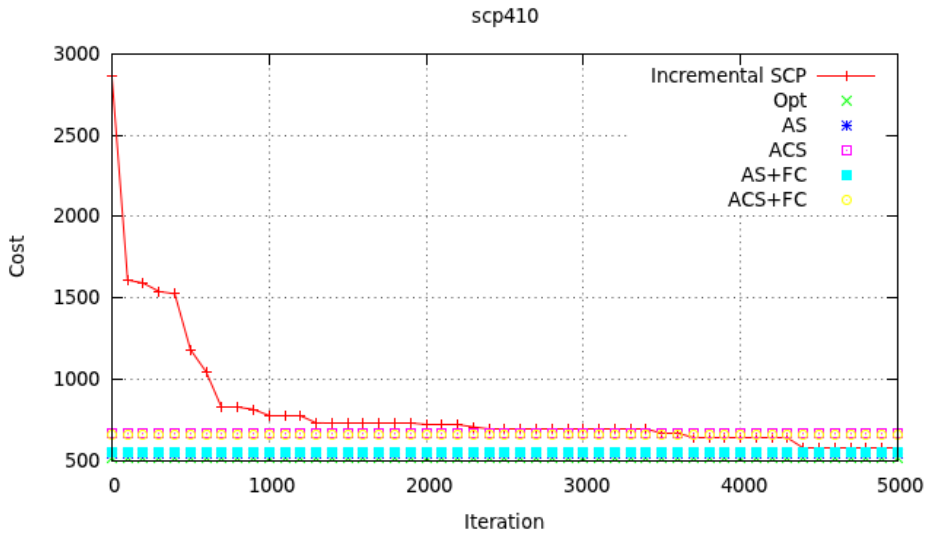


FIGURE 1. Comparative evaluation of algorithms for the scp410 dataset.

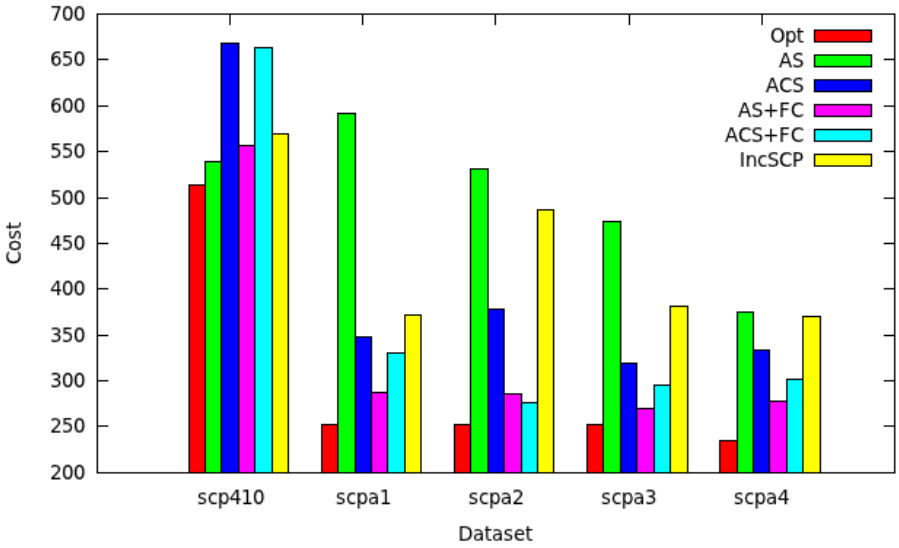


FIGURE 2. Comparative evaluation of algorithms on the considered datasets.

5.1. **Discussion.** We have addressed the set covering problem by using an incremental clustering approach. Incremental methods are quite a novel topic in cluster analysis research. They are essentially different from online and offline learning methods. With offline learning the whole data set is assumed available at all times, and with online learning the learning procedure becomes iterative and considers one data item at a time in repetitive turns. On the contrary, incremental learning procedures assume that at each time step the decision is based on updating the data structures based on the data structures constructed at the previous time step and this approach should increase robustness as compared with traditional learning methods.

In [**?**] the authors address the on-line version of SCP. In the online version an adversary gives elements to the algorithm one by one. Whenever a new element is arriving, the algorithm has to cover it. Even though it is on-line, this approach is essentially different from the one considered in this paper since the elements of X and the members of F are known in advance to the algorithm while the set $X^{'} \subseteq X$ of elements given by the adversary isn't and the task is to minimize the total cost of the sets chosen by the algorithm. Even though numerical experiments are not presented in the paper, the authors perform a thorough complexity analysis of their algorithm.

The set covering problem approach introduced in [**?**] uses an ant colony optimization algorithm together with a new transition rule. In our initial experiments performed on some of the datasets also used in [**?**] we have obtained similar results as in [**?**] without managing to outperform their results in any of the so far considered datasets. A careful analysis of our approach regarding the fine tuning of the algorithm parameters is needed in order to improve the obtained costs.

In [**?**] the authors define a clustered variant for the set covering problem. This is different from our approach since in the Clustered-SCP the subsets are partitioned into k clusters and a fixed cost is associated to each cluster. So the objective is to find a cover that minimizes the sum of subsets costs plus the sum of fixed cluster costs. In spite of not presenting numerical experiments in the paper, the authors have performed in depth complexity analysis of their approach.

In [**?**] the authors introduce a new class of set covering heuristics, based on clustering techniques. They begin by partitioning the set of columns into clusters based on some column similarity measure and then they select the best column from each cluster. The authors have performed experiments on randomly generated test problems and they indicate promising results. As opposed to our approach, the method presented in [**?**] is not incremental.

There are a large number of examples suggesting that incremental learning and reasoning are some of the intelligent methods most used by humans in their real life. Such an example is speech recognition, where the listener recognizes and understands the speech of the speaker in incremental steps, before actually having the whole statement available. As well, when incrementally clustering, humans have the ability to dynamically recognize that the extra data item considered actually contributes to a local reorganization of the data clusters, leading to, for instance, an increase or decrease in the total number of clusters.

6. Conclusions and future work

We have developed an incremental clustering algorithm in order to address the set covering problem. The algorithm continuously considers new items to be clustered. Whenever a new data item arrives it is encapsulated by an agent which will autonomously decide to join a certain cluster in the attempt to either maximize the cluster cover or minimize its the cost. We have used soft agents in order to deal with the two conflicting objectives: maximize cover and minimize cost. As in any approximation algorithm an optimal solution is not guaranteed to be found, the purpose being to find reasonably good solutions fast enough. Tests on other datasets from [**?**] are ongoing. We are also investigating possibilities for speeding up the convergence. Ongoing tests

suggest promising results and lead us to also consider similar problems like the set partitioning problem.

Acknowledgement

The authors wish to thank for the financial support provided from programs co-financed by The Sectorial Operational Programme Human Resources Development, Contract POSDRU 6/1.5/S/3 "Doctoral studies: through science towards society".

References

Babeş-Bolyai University, Department of Computer Science, 1 M. Kogălniceanu Street, 400084, Cluj-Napoca, Romania

E-mail address: {rgaceanu,hfpop}@cs.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

ALGEBRAIC APPROACH TO IMPLEMENTING AN ATL MODEL CHECKER

LAURA FLORENTINA STOICA AND FLORIAN MIRCEA BOIAN

Abstract. The Alternating-Time Temporal Logic (ATL) is interpreted over the game structures. An open system interacts with its environment and its behavior depends on the state of the system as well as the behavior of the environment. In this paper we will use an algebraic approach to implement an ATL model checker. Our tool is implemented in client-server paradigm: ATL Designer, the client tool, allows an interactive construction of concurrent game structures as a directed multi-graph and the ATL Checker, the core of our tool, represents the server part and is published as Web service. The ATL Checker includes an algebraic compiler implemented with ANTLR (Another Tool for Language Recognition) support. The main function of the Web service is to parse a given formula, find the set of nodes in which the formula is satisfied and return the result to the user.

1. Introduction

An *open system* is a system that interacts with its environment and whose behaviour depends on the state of the system as well as the behaviour of the environment. In order to construct models suitable for open systems, the Alternating-time Temporal Logic (ATL) was defined [1].

A Computation Tree Logic (CTL) specification is interpreted over Kripke structures, which provide a model for the computations of a closed system. In a *closed system* the behaviour is completely determined by the state of the system. In order to capture compositions of open systems, we present an extension of CTL, the alternating-time temporal logic (ATL), which is interpreted over game structures.

ATL extends CTL by replacing the path quantifiers $\exists$ (existential quantification) and $\forall$ (universal quantification) by cooperation modalities $\langle\langle A\rangle\rangle$,

Received by the editors: May 5, 2012.

2010 *Mathematics Subject Classification.* 68Q60, 68Q85, 68N20.

1998 *CR Categories and Descriptors.* **C.2.4** [Computer-Communication Networks]: Distributed Systems - *Client/server*; **D.2.4** [Software Engineering]: Software/Program Verification - *Model checking*; **F.3.1** [Logics and meanings of programs]: Specifying and Verifying and Reasoning about Programs - *Model checking*.

Key words and phrases. algebraic compiler, ATL model checking, ANTLR, Web services.

where A is a team of agents. A formula $\langle\langle A\rangle\rangle\ \varphi$ expresses that the team A has a collective strategy to enforce φ.

ATL is a branching-time temporal logic that naturally describes computations of multi-agent system and multiplayer games. It offers selective quantification over program-paths that are possible outcomes of games [1]. ATL uses alternating-time formulas to construct model-checkers in order to address problems such as receptiveness, realizability, and controllability.

Model checking is a technology used for verification and validation of automated system.

Over structures without fairness constraints, the model-checking complexity of ATL is linear in the size of the game structure and length of the formula, and the symbolic model-checking algorithm for CTL [2] extends with few modifications to ATL.

In the following the ATL language is defined. The operator scheme Σ_{atl} is defined as a triple $\langle S_{atl}, O_{atl}, \sigma_{atl}\rangle$ where set S_{atl} contains the representations of the ATL formulas, $O_{atl} = \{\neg, \vee, \wedge, \rightarrow, \Diamond, \circ, \Box, U\}$ is the set of operators, and the $\sigma_{atl} : O_{atl} \rightarrow S^*_{atl} \times S_{atl}$ is a function which defines the signature of the operators. The $\Diamond$ ('future'), $\circ$ ('next'), $\Box$ ('always'), and U ('until') are temporal operators. The ATL model checker can be defined as the Σ_{atl} - language [4] given in the form $L_{atl} = \langle Sem_{atl}, Syn_{atl}, \mathfrak{L}_{atl} : Sem_{atl} \rightarrow Syn_{atl}\rangle$ where Syn_{atl} is the word algebra of the operator scheme Σ_{atl} generated by the operations from O_{atl} and a finite set of variables, representing atomic propositions, denoted by AP. Sem_{atl} represents ATL semantic algebra defined over the set of ATL formulas which are satisfied by the ATL model. $\mathfrak{L}_{atl}$ is a mapping which associates the set of satisfied formulas from Sem_{atl} to ATL expressions from Syn_{atl} which satisfy these formulas.

Having well-defined ATL language, implementation of an ATL model checker will be equated with an algebraic compiler which translates an ATL formula of the ATL model to set of nodes over which that formula is satisfied.

We have chosen an algebraic approach to implement the ATL symbolic model checking algorithm and we used a Web services technology to make our model checker tool available to various clients. We provide as part of our tool an example of GUI Client for the Web service.

The paper is organized as follows. In section 2 we present the definition of the concurrent game structure with its syntax and semantics. In section 3 is presented the implementation of an algebraic compiler used by our tool to verify satisfiability of ATL formulas for given models. Invocation of the compiler will be accomplished through a Web service described in section 4. In section 5 we test our new model checking tool on a simple ATL model. Conclusions are presented in section 6.

2. ATL logic with its syntax and semantics

Alternating-time Temporal Logic (ATL) is a branching-time temporal logic that naturally describes computations of multi-agent system and multiplayer

games. It offers selective quantification over program-paths that are possible outcomes of games [1].

Concurrent game structures can be used to model compositions of open systems. Unlike in Kripke structures, in a concurrent game structure, the environment is involved in a state transition. The environment is modelled by a set of agents. Each agent may perform some actions and at least one action is available to the agent at each state.

The concurrent game structure in [1] is defined as a tuple $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ with the following components:

- A natural number $k \geq 1$ of *players*. We identify the players with numbers $1, ..., k$.
- A finite set Q of states.
- A finite set Π of *propositions* (also called *observables*).
- For each state $q \in Q$, a set $\pi(q) \subseteq \Pi$ of propositions true at q. The function π is called *labelling* (or *observation*) *function*.
- For each player $a \in \{1, ..., k\}$ and each state $q \in Q$, we identify the moves of player a at state q with the numbers $1, ..., d_a(q)$, where $d_a(q) \geq 1$ represents the number of available moves. For each state $q \in Q$, a *move vector* at q is a tuple $\langle j_1, ..., j_k \rangle$ such that $1 \leq j_a \leq d_a(q)$ for each player a. Given a state $q \in Q$, we write $D(q)$ for the set $\{1, ..., d_1(q)\} \times ... \times \{1, ..., d_k(q)\}$ of moves vector. The function D is called *move function*.
- For each state $q \in Q$ and each move vector $\langle j_1, ..., j_k \rangle \in D(q)$, $\delta(q, j_1, ..., j_k) \in Q$ represents the state that results from state q if every player $a \in 1, ..., k$ chooses a move j_a. The function δ is called the *transition function*.

For a computation starting at state q we refer to it as a *q-computation*. For a computation λ and a position $i \geq 0$, we use $\lambda[i]$ to denote the i-th state of λ, $\lambda[0, i]$ to denote the finite prefix $q_0, q_1, ..., q_i$ of λ and $\lambda[i, \infty]$ to denote the infinite suffix $q_i, q_{i+1}, ...$, of λ.

The s*yntax of a temporal logic ATL* is defined with respect to a finite set Π of propositions and a finite set $\Lambda = \{1, ..., k\}$ of players [1].

An ATL formula is one of the following:

(s1): p, for propositions $p \in \Pi$;
(s2): $\neg\varphi$, $\varphi_1 \vee \varphi_2$, where φ and $(\varphi_i)_{i=\overline{1,2}}$ are ATL formulas;
(s3): $\langle\langle A \rangle\rangle \circ \varphi$, $\langle\langle A \rangle\rangle \Box \varphi$, or $\langle\langle A \rangle\rangle \varphi_1 U \varphi_2$, where $A \subseteq \Lambda$ is a set of players, and φ, $(\varphi_i)_{i=\overline{1,2}}$ are ATL formulas.

The operator $\langle\langle\rangle\rangle$ is a path quantifier, and $\circ$ ('next'), $\Box$ ('always'), and U ('until') are temporal operators.

A *strategy* for player $a \in \Lambda$ is a function $f_a : \mathcal{Q}^+ \to \mathbb{N}$ that maps every nonempty finite state sequence $\lambda = q_0 q_1 ... q_n$, $n \geq 0$, to a natural number such that $f_a(\lambda) \in \{1, ..., d_a(q_n)\}$. Thus, the strategy f_a determines for every finite prefix λ of a computation a move $f_a(\lambda)$ for player a in the last state of λ.

Given a set $A \subseteq \{1, ..., k\}$ of players, a state $q \in Q$ and a set $F_A = \{f_a | a \in A\}$ of strategies, one for each player in A, the outcome of F_A is defined to be the set $out(q, F_A)$ of q-computations that the players from A are enforcing when they follow the strategies in F_A. A *computation* $\lambda = q_0, q_1, q_2, ...$ is in $out(q, F_A)$ if $q_0 = q$ and for all positions $i \geq 0$, there is a move vector $\langle j_1, ..., j_k \rangle$ such that $j_a = f_a(\lambda[0, i])$ for all players $a \in A$ and $\delta(q_i, j_1, ..., j_k) = q_{i+1}$ [1].

Formal definition of ATL semantics given in [1] is defined over a game structure $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$. We write $S, q \models \varphi$ to indicate that the state q satisfies the formula φ in the structure S. When S is clear from the context, we omit it and write $q \models \varphi$.

The satisfaction relation $\models$ is defined for all states q of S inductively as follows:

- $q \models p$, for propositions $p \in \Pi$, iff $p \in \pi(q)$;
- $q \models \neg\varphi$ iff $q \not\models \varphi$;
- $q \models \varphi_1 \vee \varphi_2$ iff $q \models \varphi_1$ or $q \models \varphi_2$;
- $q \models \varphi_1 \wedge \varphi_2$ iff $q \models \varphi_1$ and $q \models \varphi_2$;
- $q \models \varphi_1 \rightarrow \varphi_2$ iff $q \models \neg\varphi_1$ or $q \models \varphi_2$;
- $q \models \langle\langle A \rangle\rangle \circ \varphi$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$, we have $\lambda[1] \models \varphi$;
- $q \models \langle\langle A \rangle\rangle \Box\varphi$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$, and all positions $i \geq 0$ such that $\lambda[i] \models \varphi$;
- $q \models \langle\langle A \rangle\rangle \Diamond\varphi$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$, there exists a position $i \geq 0$ such that $\lambda[i] \models \varphi$;
- $q \models \langle\langle A \rangle\rangle \varphi_1 U \varphi_2$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$, there exists a position $i \geq 0$ such that $\lambda[i] \models \varphi_2$ and for all positions $0 \leq j < i$, we have $\lambda[j] \models \varphi_1$

3. Java implementation of ATL model checker

Our algebraic compiler $\mathcal{C}$ translates a formula φ of the ATL model to the set of nodes Q' over which formula φ is satisfied. That is, $\mathcal{C}(\varphi) = Q'$ where $Q' = \{q \in Q | q \models \varphi\}$.

For the implementation of our algebraic compiler we choose the ANTLR [3]. ANTLR is a compiler generator which takes as input a grammar - an exact description of the source language, and generates a recognizer for the language defined by the grammar.

The algebraic compiler $\mathcal{C}$ implements the following ATL symbolic model checking algorithm given by [1]. We add three more symbolic operators $(\wedge, \rightarrow, \Diamond)$ in the ATL symbolic model checking algorithm to show all operators form O_{atl}.

```
Function Eval_A(φ) as set of states ⊆ Q
  case φ = p:
      return [p];
  case φ = ¬θ:
      return Q \ Eval_A(θ);
  case φ = θ1 ∨ θ2:
      return Eval_A(θ1) ∪ Eval_A(θ2);
  case φ = θ1 ∧ θ2:
      return Eval_A(θ1) ∩ Eval_A(θ2);
  case φ = θ1 → θ2:
      return (Q \ Eval_A(θ1)) ∩ Eval_A(θ2);
  case φ = ⟨⟨A⟩⟩ ∘ θ:
      return Pre(A, Eval_A(θ));
  case φ = ⟨⟨A⟩⟩□θ:
      ρ := Q; τ := Eval_A(θ); τ0 := τ;
      while ρ ⊈ τ do
          ρ := τ;
          τ := Pre(A, ρ) ∩ τ0;
      wend
      return ρ;
  case φ = ⟨⟨A⟩⟩◇θ:
      ρ := ∅; τ := Eval_A(θ);
      while τ ⊈ ρ do
          ρ := ρ ∪ τ;
          τ := Pre(A, ρ) ∩ Q;
      wend
      return ρ;
  case φ = ⟨⟨A⟩⟩θ1Uθ2:
      ρ := ∅; τ := Eval_A(θ2); τ0 := Eval_A(θ1);
      while τ ⊈ ρ do
          ρ := ρ ∪ τ;
          τ := Pre(A, ρ) ∩ τ0;
      wend
      return ρ;
End Function
```

The $Pre(A, \rho)$ function, where $A \subseteq \Lambda$ and $\rho \subseteq Q$, returns the set of states q such that from q, the players in A can cooperate and enforce the next state to be in ρ. $Pre(A, \rho)$ contains state $q \in Q$ if for every player $a \in A$ there exists a move $j_a \in \{1, ..., d_a(q)\}$ such that for all players $b \in \Lambda \backslash A$ whatever are their moves we have $\delta(q, j_1, ..., j_k) \in \rho$.

In order to translate a formula φ of an ATL model to the set of nodes Q' over which formula φ is satisfied, the attachment of specific actions to grammatical constructions within specification grammar of ATL is necessary.

The actions are written in target language of the generated parser (in our case, Java). These actions are incorporated in source code of the parser and are activated whenever the parser recognizes a valid syntactic construction in the translated ATL formula. In case of the algebraic compiler $\mathcal{C}$, the actions define the semantics of the ATL model checker, i.e., the implementation of the ATL operators.

The model checker generated by ANTLR from our specification grammar of ATL, takes as input the concurrent game structure S and formula φ, and

provides as output the set $Q' = \{q \in Q | q \models \varphi\}$ the set of states where the formula φ is satisfied.

The corresponding *action* included in the ANTLR grammar of ATL language for implementing the $\circ$ operator is:

```
'<<A>>@' f=formula
 {
   HashSet rez = Pre($f.set);
   $set = rez;
   trace("atlFormula",3);
   printSet("<<A>>@ " + $f.text,rez);
 }
```

For $\circ$ ATL operator in ANTLR we use the @ symbol.

The *formula* represents a term from a production of the ATL grammar, and *f*, *rez* variables are sets used in internal implementation of the algebraic compiler.

The $Pre(\$f.set)$ is a function that returns the set of states *rez* such that from each state of *rez*, the players in A can cooperate and enforce the next state to be in the set of states in which the formula f is satisfied.

The algebraic compiler $\mathcal{C}$ translates formula f of the ATL model S to set of nodes Q' over which formula f is satisfied. The implementation of the algebraic compiler $\mathcal{C}$ is made in two steps. First, we need a syntactic parser to verify the syntactic correctness of a formula *f*. Then, we should deal with the semantics of the ATL language (Sem_{atl}), respectively with the implementation of the operators from set $O_{atl} = \{\neg, \vee, \wedge, \rightarrow, \Diamond, \circ, \Box, U\}$.

Writing a translator for certain language is difficult to be achieved, requiring time and a considerable effort. Currently there are specialized tools which generate most of necessary code beginning from a specification grammar of the source language.

In figure 1 is represented the algebraic compiler implementation process, based on our specification grammar of ATL language.

4. Publishing the ATL model checker as a Web service

Web Services, as a distributed application technology, simplifies interoperability between heterogeneous distributed systems. Clients can access Web services regardless of the platform or operating system upon which the service or the client is implemented [5]. In order to make available our implementation of algebraic compiler as a reusable component of an ATL model checking tool, we published it as a Web service. The Web service will receive from a client the XML representation of an ATL model S and an ATL formula φ. The original form of the ATL model S is passed then to the algebraic compiler $\mathcal{C}$ generated by ANTLR using our ATL extended grammar. For a syntactically correct formula φ, the compiler will return as result $\mathcal{C}(\varphi) = \{q \in Q | q \models \varphi\}$, the set of states in which the formula is satisfied.

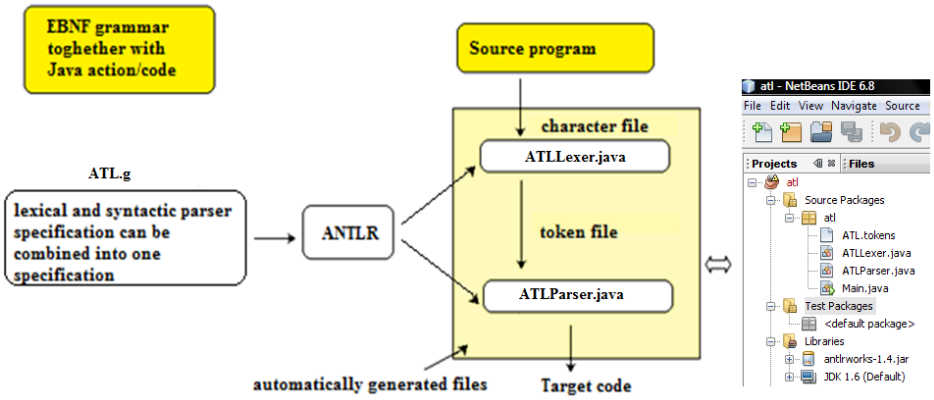


FIGURE 1. Algebraic compiler implementation

Obviously, the formula φ may contain syntactical errors. In order to notify the client about these possible errors, we must override the default behavior of the ANTLR error-handling.

The architecture of the ATL model checker Web Service is showed in figure 2.

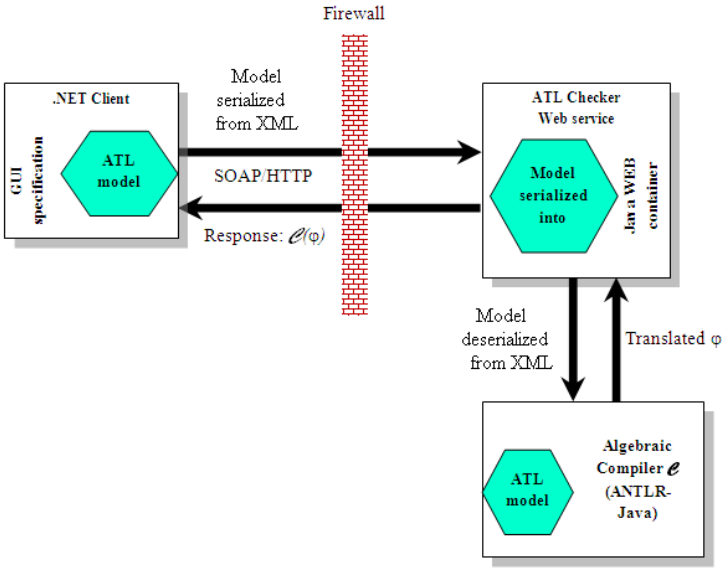


FIGURE 2. Architecture of the Web Service

In conclusion, for a given ATL model and an ATL formula φ, the Web service will parse the formula and will return to client the set of states in which the formula is satisfied if formula is syntactically correct, or a message describing the error from an erroneous formula.

5. Testing the new ATL model checker tool

Consider a system with two processes, Px and Py. The process Px assigns values to the Boolean variable x. When x=*false*, then Px can leave the value of x unchanged or change it in true. When x=*true*, then Px leaves the value of x unchanged. The process Py assigns values to the Boolean variable y, in the same way as the process Px.

We model the synchronous composition of two processes by the S_{xy} concurrent game structure, where $S_{xy} = \langle k, Q, \Pi, \pi, d, \delta \rangle$:

- k=2 (first player denoted process Px, second player denoted process Py);
- $Q = \{q_0, q_1, q_2, q_3\} - q_0$ means $x = y = false$, q_1 means $x = true$ and $y = false$, etc;
- $\Pi = \{x, y\}$;
- $\pi(q_0) = \emptyset, \pi(q_1) = \{x\}, \pi(q_2) = \{y\}, \pi(q_3) = \{x, y\}$;
- $d_1(q_0) = d_1(q_2) = 2$ (means in state q_0 and q_2 move 1 of the first player leaves the value of x unchanged, and move 2 changes the value of x); $d_1(q_1) = d_1(q_3) = 1$ (means in states q_1 and q_3 first player has only one move, namely, to leave the value of x unchanged); $d_2(q_0) = d_2(q_1) = 2, d_2(q_2) = d_2(q_3) = 1$;
- state q_0 has four successors: $\delta(q_0, 1, 1) = q_0, \delta(q_0, 1, 2) = q_2, \delta(q_0, 2, 1) = q_1, \delta(q_0, 2, 2) = q_3$.

A concurrent game is played on a state space. Every player chooses a move. The combination of choices determines a transition from the current state to a successor state.

The model checking tool is based on a *C Sharp GUI* client who allows interactive graphical development of the ATL models.

The model is sent as a XML document to the Web service, together with the formula to be verified. The response from server is displayed in a separate window, as we will see in the following section.

Given the ATL formula $\varphi = \langle\langle A \rangle\rangle \circ (x \wedge y)$ for game structure from figure 3 with $A = \{2\}$, the output of the model checker is $Q' = \{1, 3\}$. From state q_1 if second player chooses the move 2 the next state is q_3 whatever is the move selected by the first player. From the state q_2 for the move 1 of the second player, the first player can choose the move 1. Thus the game remains in state q_2. For that reason the state $2 \notin Q'$.

The answer from the server is showed into separate windows.

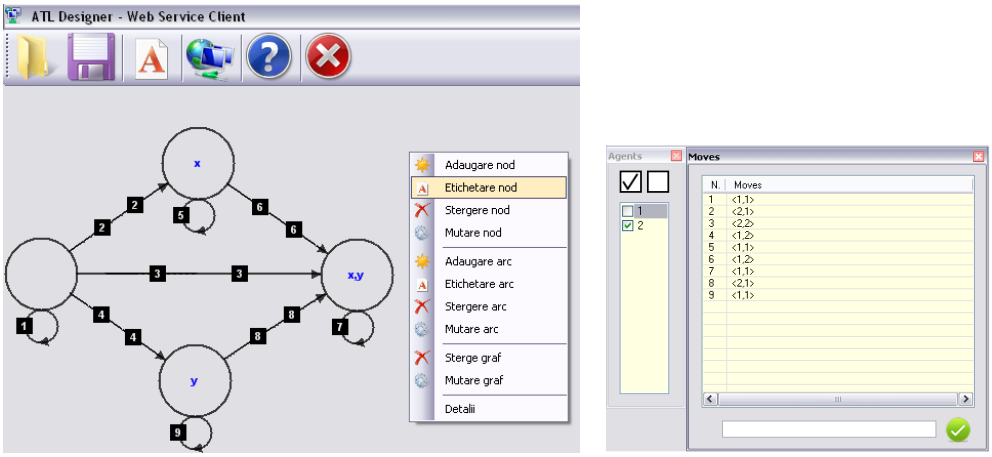


FIGURE 3. CTL Designer - the *C Sharp* client

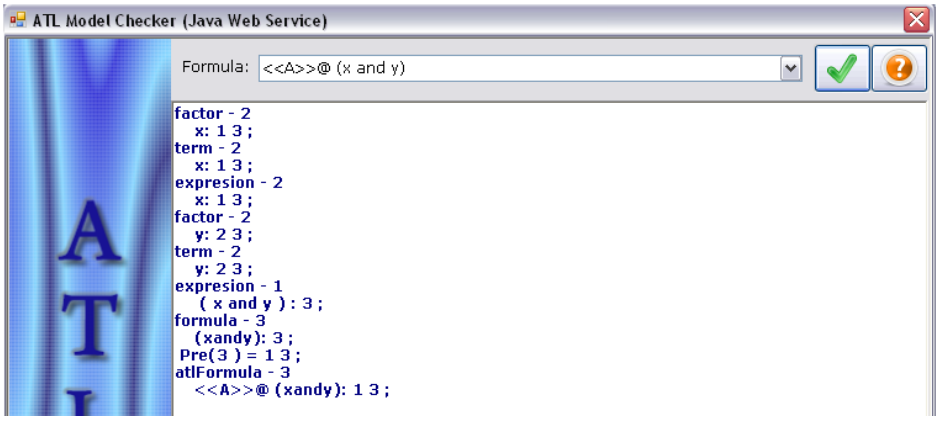


FIGURE 4. Invoking the algebraic compiler (Web service) from the ATL Designer (client)

6. CONCLUSIONS

For a given ATL model (concurrent game structure) and an ATL formula, the Web service will parse the formula and will return to client the set of states in which the formula is satisfied if formula is syntactically correct, or a message describing the error from an erroneous formula.

We built an ATL model checking tool, based on robust technologies (Java, .NET) and well-known standards (XML, SOAP, HTTP).

As a great facility we mention the capability of interactive graphical specification of the ATL model, using the client tool (ATL Designer).

References

[1] R. Alur, T. A. Henzinger , O. Kupferman, *Alternating-Time Temporal Logic*, Journal of the ACM, Vol. 49, No. 5, September 2002, pp. 672713.
[2] L. Cacovean, F. Stoica, *Algebraic Specification Implementation for CTL Model Checker Using ANTLR Tools*, 2008 WSEAS International Conferences, Computers and Simulation in Modern Science - Volume II, Bucharest, Romania, 2008, pp. 45-50.
[3] Terence Parr , *The Definitive ANTLR Reference, Building Domain-Specific Languages*, version: 2007.
[4] E. Van Wyk, Specification Languages in Algebraic Compilers, Theoretical Computer Science, 231(3), 2003, pp. 351385.
[5] Florian M. Boian, *Servicii Web; Modele, Platforme, Aplicatii*. Seria 245 PC, Editura Albastra, Cluj-Napoca 2011, ISBN 978-973-650-266-8.

"Lucian Blaga" University of Sibiu, Faculty of Sciences, Department of Mathematics and Informatics, 5-7 Dr. Ion Ratiu St, 550012, Sibiu, Romania
E-mail address: laura.cacovean@ulbsibiu.ro

Babes-Bolyai University of Cluj-Napoca, Faculty of Mathematics and Informatics, Department of Computer Science, 1 M Kogalniceanu St, 400084, Cluj-Napoca, Romania
E-mail address: florin@cs.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

DRAS: DERIVED REQUIREMENTS GENERATION

DAVID BAR-ON AND SHMUEL TYSZBEROWICZ

Abstract. A system specification may include many interdependencies between the specified requirements. Requirements may conflict with one another and they may impact other requirements as well. We present the DRAS (Derived Requirements generation by Actions and States) methodology that helps to identify Functional Requirements (FRs) that are in conflict with other FRs DRAS also assists with generating the derived requirements that are inferred from the conflicting requirements. DRAS is based on the observation that using the same action in two requirements indicates that those requirements may conflict. In order to find which requirements potentially are in conflict with a given requirement, DRAS considers actions stated by the requirements, their implied actions, modes (also called states), and action modifiers.

The DRAS methodology is explained by means of a comprehensive example, which uses a subset of simplified requirements from an industrial project that one of the authors participated in its requirements definition and analysis.

1. Introduction

System and product requirements often contain inconsistent and conflicting requirements. Requirements may impact other requirements as well. Inconsistency between requirements usually means that either existing requirements should be enhanced, or new requirements should be written. The DRAS methodology (***DRAS*** — *Derived Requirements generation by Actions and States*) that is described in this paper helps to identify conflicting Functional Requirements (FRs) and to resolve the conflicts. The method is not effective for Non-Functional Requirements (NFRs).

Consider the functional requirements "close the window when the outside temperature is below 10 degrees" and "open the window in the morning".

Received by the editors: April 25, 2012.

1991 *Mathematics Subject Classification.* 68N99.

1998 *CR Categories and Descriptors.* code [**D.2.1**]: Requirements/Specifications – *Methodologies* ;

Key words and phrases. Early Aspects, Requirements Engineering, Conflicting Requirements, Crosscutting Requirements, Derived Requirements, Match-Point.

The requirement to close the window conflicts with the requirement to open the window when the temperature in the morning decreases below 10 degrees. The outcome of analyzing the interaction between the requirements should include a decision whether or not to open the window in the morning, when the temperature is below 10 degrees.

The resolution of conflicting requirements influences the definition and the architecture of the system. Therefore, it is very important to identify conflicting requirements as soon as possible in the software development process, for example during requirements analysis. Otherwise there may be a major overhead work during later development phases. While identifying and handling conflicting requirements, both functional-requirements and non-functional requirements should be considered. A rigorous analysis and understanding of conflicting requirements and their interactions is essential to derive a balanced architecture. Ignoring it may result in an incomplete understanding of specified requirements and, consequently, poorly informed architectural choices.

One way to resolve the conflicts between requirements is to define Derived Requirements (DRs). These are requirements that we infer, or derive, from other user requirements. They represent the outcome of resolving interactions and conflicts between requirements. DRs may be either new requirements or enhancements to existing requirements.

The identification of conflicting requirements is a difficult process, especially in large systems; consequently, methods and tools that can identify conflicting requirements and define the resulting DRs are needed. This is because the system may be designed according to one of the requirements, while the other conflicting requirements should have precedence, so a redesign may be required later in the process.

The DRAS methodology is used to generate textual DRs from stakeholders' textual requirements. DRAS aims to assist the process of identifying conflicting functional requirements and supports resolving these conflicts. The resolution may include changing, enhancing, or overriding some of the conflicting FRs, including the generation of textual derived requirements which are part of the resolution. Note that DRAS does not handle non-functional requirements, as it relies on action words to identify conflicts between requirements.

We have addressed the main concepts of DRAS methodology in [5, 4]. This paper expands on the previous papers by providing more details about the methodology, procedures used for its implementation (using a prototype tool), and adding an Appendix with a comprehensive example of the DRAS process.

DRAS includes some ideas that have been adopted from existing methods for identifying and handling conflicting requirements, especially from [2, 31,

7]. DRAS uses actions as the primary means for identifying conflicting FRs, similar to ý [2]. Actions are the functions specified by the FRs. In the example above, "open window" is the action used by both requirements. Note that using the same action by both requirements indicates that one of them may conflict with the other.

The main contribution of the DRAS methodology is the way conflicting requirements are identified. In order to find which requirements may conflict a given requirement *Req*, DRAS considers *Req's actions, modes* (also called states), and *action modifiers*. Actions can be explicitly stated in *Req*, implied by actions in *Req*, or imply actions in *Req*. For example, "refresh the room"implies the action "open a window" and "unlock the window" is implied by "open the window" (and indirectly, is also implied by "refresh the room").

The activation of some actions depends on the mode of the system *Req* refers to. For example, there might be different requirements for opening the window depending on the season - summer or winter. Requirements that are relevant only for the summer usually are not conflicting requirements that are relevant only for the winter.

We also distinguish between a restricting action, that forbids some activities (e.g., "open a window" is restricted in "do not open the window when the temperature is below 10 degrees") and ease a restriction for an action (e.g., "the window may be opened when leaving the room, even if the temperature is below 10 degrees"). This distinction is used to determine whether to consider implied or implying actions.

In designing a system that reengineers the requirements, one dilemma is whether the output should be expressed textually or a more formal method should be used. The initial stakeholders' requirements for a system or a product are usually textual. Only later in the development process are the requirements specifications transformed into a more formal, technical representation (such as UML diagrams ý [16]). When the data is formally represented, it is easier to identify conflicting requirements and the requirements they conflict. However, non-technical stakeholders, such as customers and marketing representatives, usually are not trained to read formal specifications. Therefore, it is advantageous to be able to generate DRs in a textual form and to integrate them with other requirements. This enables non-technical stakeholders to review and understand the specifications. Therefore we decided to create textual requirements as the output of DRAS.

In order to generate textual DRs from stakeholders' requirements, DRAS first identifies *match-points* (defined in ý [7]) between requirements. A *Match-point* in requirements is identifying tentative conflicts between the requirements (e.g., a common action). This is performed by identifying common

actions that are used by the requirements (inspired by ý [2]), and by identifying common system modes and states (when these actions are used). The DRs are then created based on the conflicting requirements. This enables review and evaluation process by both technical and non-technical stakeholders.

As an example, following is a simple set of requirements for initiating a call from a cellular system:

- *Ra. When a phone user dials a number, the phone shall initiate a call to the dialed number.*
- *Rb. The phone shall allow initiating calls to the police (911 in the US, 112 in Europe) under any condition.*
- *Rc. The phone shall be allowed to initiate calls and receive calls only after checking that the user is allowed to use it (bills paid, phone not stolen, etc.).*

The first observation is that all of these requirements are FRs, and the action "call initiation" is mentioned in each of them. Therefore, the requirements may conflict with each other. Further analysis reveals that *Rb* and *Rc* are tentatively conflicting, as *Rc* restricts call initiation while *Rb* eases restriction for initiating a call. Analyzing each pair of requirements shows that *Rc* conflicts *Ra*, restricting its specification. Assuming that *Rc* has a higher priority than *Ra*, the result is an enhancement (change) to *Ra*. The enhanced derived requirement may be:

- *Rd. When a user of a phone dials a number, the phone shall initiate a call to the dialed number only if the user is legitimate.*

In this case *Rc* is redundant since *Rd* includes it. It still may be important to keep such a requirement, as usually not all the requirements it conflicts are identified in the early stages of development.

Another derived requirement is required since *Rb* and *Rc* are in conflict with each other. Illegitimate users are allowed to dial the police according to *Rb* and disallowed according to *Rc*. A common solution in cellular systems gives *R2* higher priority, allowing also illegitimate users to dial the police:

- *Re. Illegitimate user should be allowed to dial the police.*

Alternatively, this can be an enhancement to *Rd*:

- *Rd (enhanced). The phone should not allow dialing by an illegitimate user, unless the user dials the police.*

This paper is structured as follows. Section 2 gives an overview of related work. In Section 3 we discuss the problem of generating DRs and outline how DRAS handles this issue. Section 4 concludes the paper, and discusses

further work and enhancements to DRAS. The Appendix contains a detailed description of DRAS, which is demonstrated using detailed examples.

2. Related Work

Many methods exist to identify crosscutting and conflicting requirements. Since DRAS uses ideas of Aspect Oriented Requirements Engineering (AORE), we reference AORE-based methods for identifying crosscutting concerns, and use some of those ideas to identify conflicting requirements.

AORE complements existing requirements engineering approaches by offering additional abstraction and composition mechanisms for systematically handling crosscutting, or aspectual, requirements. AORE methods include techniques for explicitly modeling aspects or concerns, in the context of requirements specifications. These methods allow identifying concerns in requirements, identifying crosscutting aspects between requirements, evaluating the aspects, and resolving conflicts caused by aspectual requirements. An extensive review of AORE methods can be found in [5, 1].

Several of the referenced papers are using the term ***crosscutting requirements***, referring to requirements that are inconsistent with other requirements they crosscut, requirements that enhance or change the functionality of other requirements, and sometimes are also in conflict with these requirements. The DRAS methodology mainly deals with ***conflicting requirements***. However, several of the methods used for handling crosscutting requirements are also applicable for conflicting requirements and are adopted by DRAS.

Section ďž″2.1 elaborates on AORE methodologies that includes ideas that are used DRAS. We describe some of the methodologies that have been evaluated during the development of DRAS in Section ďž″2.2, including a brief discussion on their relevancy to DRAS.

2.1. Methods with ideas used by DRAS.

Goal Oriented Requirements Analysis [26] explores the alternatives for achieving the goals as described in a given set of high level requirements, in order to achieve high-level requirements and select the proper alternatives. The method correlates *Softgoals* (NFRs) with goals and other Softgoals, which is similar to analyzing crosscutting requirements. An enhancement of this method is ***Aspects in Requirements Goal Models*** (Yu et-al [36], summarized in [11]) proposes a process for discovering aspects early in the software development process, based on relationships between functional and non-functional goals. V-shape graphs are used to decompose goals from Softgoals, and to present the goal/softgoal hierarchies. The *Discovering Aspects from Requirements Goal Models* method described below is based on these methods.

Discovering Aspects from Requirements Goal Models method [36] is an enhancement of the Goal Oriented Requirements Analysis method. It proposes a process for discovering aspects early in the software development process, from relationships between functional and non-functional goals. DRAS enhancements over this method include correlating between system specifications to identify conflicting functional requirements and then resolving those conflicts.

Modularization and Composition of the Aspectual Requirements method, described in [31], defines an AORE process model for resolving conflicts between requirements, and determining the influence of the conflicts resolution later in the architecture and design development stages. It focuses on the modularization and on the composition of requirements-level concerns that cut across other requirements. The method is primarily useful for Non Functional Requirements (NFRs), such as: availability, security, response time, accuracy, reliability, and other requirements that cannot be encapsulated by a single viewpoint or use-case [18]. The method supports the separation of crosscutting NFR properties and helps to identify the mapping and influence of the requirements level aspects on artifacts at later development phases. The method identifies critical tradeoffs before the architecture is derived. It includes steps for identifying concerns and viewpoints relationships, identifying candidate aspects, identifying match-points, defining composition rules, and the composition of viewpoints and aspects using composition rules. DRAS adopted the same idea from the *Modularization and Composition of Aspectual Requirements* method, for analyzing the requirements in order to find out whether they are conflicting or whether a requirement enhances other ones, and how. However, as noted earlier, this method is mainly relevant for crosscutting NFRs while DRAS handles FRs; it also does not use actions as a means to identify crosscutting and conflicting functional requirements.

Composition Process for Aspect Oriented Requirements *(AOR)*, described in [7], is a process for composing crosscutting concerns with the concerns (requirements) they conflict. The method is used primarily for NFRs, but it also includes techniques that are applicable for functional requirements. The process includes the following steps: a) identify concerns; b) identify candidate aspects; c) compose candidate aspects with concerns. DRAS adopted from this methodology the identification of match points and the use of contribution and composition-rules. Note that this method mainly handles NFRs; it does not work well to identify match-points between FRs. Brito, et al. [8] presents enhancements to this approach; the composition rules are refined with new operations inspired by LOTOS [6] operators.

Theme and ***Theme/Doc*** are defined in [2] and [3]. These methods handle FRs, rather than mainly NFRs as in most of the other methods described

earlier. *Theme* is a method and set of tools developed for early identification of aspects in the software development life cycle. The *theme* notion represents a system feature. Themes can be either *base themes,* which may share some behavior structure with other base themes, or *crosscutting themes-aspects*, which have a behavior that overlays the base themes functionality. *Theme/Doc* can identify aspects from interrelated behaviors of FRs, not just from NFRs as most other methods identify. The *Theme/Doc* approach and tool is used to view the relationship between behaviors in requirements documents and to identify and isolate aspects in the requirements. That is, it is primarily used to identify and separate aspects or concerns from the requirements, but not to combine aspects with other requirements. The approach provides requirements specifications views, exposing relationships between behaviors in the system. The method helps to determine which elements of functionality are *base* and which are *aspects*. The *Theme/Doc* approach assumes that if two behaviors are described in the same requirement, they are related. According to this approach, behaviors can be related to each other in three ways: *erroneously/coincidentally*, *hierarchically,* and *crosscutting*. The *Theme/Doc* process includes creating a list of actions that are used by the system. The actions are then used to identify crosscutting requirements, based on how these requirements use the actions. Those ideas were adopted by DRAS. *Theme/Doc*, however, does not take into account implied actions, actions that are the result of initiating other actions and states/modes related to the requirements. DRAS handles these issues and this enables identifying conflicts between requirements that are using related actions, but not using the same actions.

2.2. **Other Methods Evaluated.** ***Viewpoints*** [15] are used to specify the system from the perspectives (viewpoints) of each of its users (Actors in Use Case diagrams). Usually each perspective is partial and incomplete, because of the different roles each user has. A separate evaluation for each viewpoint is needed in order to define the full system's specification. For a complex system, using viewpoints allows Separation of Concerns (SoC) between different viewpoints, and provides a more manageable means of handling the system's specifications. Viewpoint-oriented methods do just that. Nuseibeh [28] presents a viewpoint as collecting and partitioning knowledge about representations, processes, and products of software development - all from the user's perspective. Several Viewpoints-Based Requirements Engineering (VBRE) methods exist. For example, Silva [33] introduces an approach for classifying and diagnosing discrepancies between viewpoints. The main purpose of Viewpoint methods is to verify that requirements cover all perspectives. Although they deal with

SoC, they do not specifically treat the issue of identifying crosscutting or conflicting requirements. Therefore, ideas from these methods were not used for DRAS (at least not directly).

Adaptation of the NFR-Framework to AORE [34] is an enhancement to the method defined in [31], which also includes some methods defined in [26], and to the NFR-Framework [12]. The NFR-framework is goal-oriented and is similar to the method described in [26], using the notion of *softgoals*. The *Adaptation of the NFR-framework to AORE* method uses the general AORE process framework defined by Rashid, et al. [31]. The main enhancement over [31] is identifying and selecting *operationalizations*. To *operationalize* a requirement means to provide more concrete and precise mechanisms (such as operations, processes, data representations, or constraints) to solve a problem. To *operationalize a softgoal* is to define possible solutions, or design alternatives, to help achieve the NFRs. This method is applicable to DRAS, but the related ideas used in DRAS were taken from [31] instead.

Goal Oriented Requirements Methodology [14] enhances the *Adaptation of the NFR-Framework to AORE* [34], and is based on the SoC principle. The method provides a way to represent crosscutting requirements separately from the requirements they affect and to specify the composition between them.

Crosscutting Quality Attributes [25, 9] propose a model to identify and specify *Quality Attributes* (QA) that crosscut other requirements at the requirements analysis stage. A QA is a non-functional concern, such as response time, accuracy, security, and reliability. This is the same as in a NFR, but from the point-of-view of the functional requirement. The aim of the method is to improve the separation of crosscutting requirements during analysis, providing a better means for identifying and managing conflicts. The QA method enables the handling of the NFR aspect of the FR together with the FR, instead of handling each of them separately. This method is only partly applicable for generating derived requirements from the requirements defined in this work. However, it mainly handles NFR and QA requirements. Also, the main methods it uses are also included in other methods [31]. Therefore, ideas from this method were not incorporated into DRAS.

3. The DRAS Methodology: Generating Derived Requirements

The DRAS methodology is used to identify and handle conflicting functional requirements. DRAS first identifies the actions used by each requirement, including implied actions, the modes (or states) that are relevant for the requirement, and the action modifiers per action. Based on this information, DRAS identifies the functional conflicting requirements, the requirements they

are in conflict with, and helps with generating the resulting *derived requirements (DRs)*. The generated requirements are textual, so that all stakeholders, including those with no technical background, can review and understand the requirements.

We have chosen to identify conflicting FRs based on the *actions* used by the requirements, similar to the Theme/Doc method [2, 3], as the functional requirements are used to specify the functions, or actions, that the system should provide.

3.1. **Input FRs.** Following are functional requirements that will be used to describe the DRAS process. The requirements are a simplified version of a very large project that one of the authors participated in the definition and the analysis of its requirements. The requirements are mainly about the Push-to-Talk (PTT [37]) action that is used to initiate calls to a pre-selected user or target number in walky-talkies, by pressing a button, also called PTT. As in walky-talkies, these calls are half-duplex, and only one participant can transmit voice at a given time.

- *R1. When the PTT is pressed, the phone shall initiate a call.*
- *R2. When another phone transmits, the phone shall not initiate voice transmission.*
- *R3. During a call, the phone shall not initiate a voice transmission when another phone transmits.*
- *R4. In the Emergency mode, the phone should always be allowed to initiate a voice transmission.*
- *R5. Illegitimate user should not be allowed to use the cellular system.*
- *R6. Illegitimate users shall not be allowed to initiate calls.*
- *R7. All users should be allowed to initiate a call to the police (an emergency number).*

As described in the introduction, DRAS purpose is to identify conflicting requirements and to generate derive requirements to resolve the conflict, based on the actions in the requirements and the relative requirements priorities.

For example, in this set of requirements *R1* and *R2* may conflict, since both are about *voice transmission*, as a call initiation implies voice transmission. In cellular systems, when the PTT is used to initiate a call, usually *R1 and R2 are conflicting requirements.* Since usually *R2* has a higher priority, i.e., a phone will not try to transmit if another phone already transmits, the conflict resolution may be as follows [the E in *R1(E)* means enhanced]:

- *R1(E). When PTT is pressed, the phone shall initiate a call unless another phone transmits.*

Note that with *R1(E)*, *R2* may be redundant. However, it is important to keep such conflicting requirements. Usually not all requirements that are in conflict with each other are identified in the early stages of development; moreover, new conflicting requirements may be added later (changes and additions to the systems' requirements usually never end).

3.2. **DRAS Outline.** The process map for the DRAS methodology is shown in Figure 1. The methodology steps are briefly described in the following sections, using as an example the requirements defined above. In the Appendix we provide a detailed example of using DRAS.

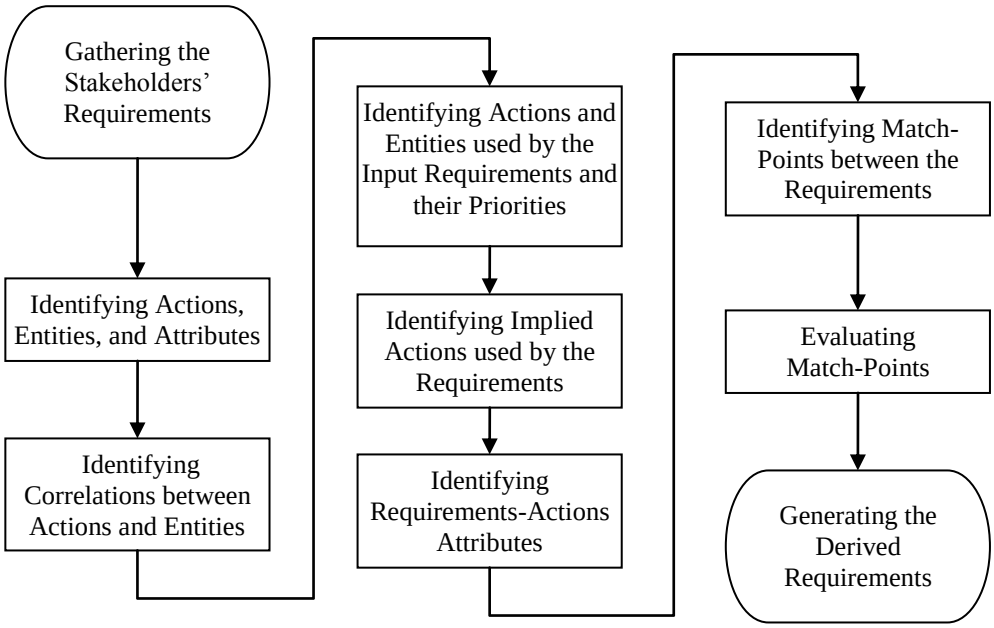


FIGURE 1. DRAS: Process Map

3.3. **Identifying Actions and Attributes Lists.** After the stakeholders' requirements are gathered and formulated, the lists of *actions*, *entities*, and *attributes* used in the system are identified. Attributes are mainly the *Modes* (and States) list of the system.

A list of contradicting pairs of modes values is also generated. This list is later used to remove tentatively identified conflicts between requirements that usually cannot occur in reality. E.g., a requirement related to call in process

usually cannot contradict with a requirement related to the case when the phone is not registered to the system.

In addition, correlations between actions and entities are identified, i.e. which actions are relevant for each entity. This is used to identify which actions are relevant to a requirement that refers to an entity, e.g. the actions related to *R5* that refers to the *cellular system* entity. However, in this section we will not refer to entities, and the relevant information can be found in the Appendix.

3.4. **Identifying Requirements and Actions Attributes.** The attributes assigned to the requirements and to the requirements' actions for assisting with the DRAS process are:

(1) Requirement's *Priority* (Section 3.4.1).
(2) Requirement's *Modes* (and *States*) (Section 3.4.2).
(3) Requirement's Action *Action Modifier* (Section 3.6.1).

Note that attributes assigned to a requirement are also assigned to the actions it uses (directly or indirectly).

3.4.1. *Requirements Priority.* The resolution of conflicting requirements depends on the relative priority of the requirements; i.e., deciding which of the conflicting requirements takes precedence. The specification of a requirement with a higher priority should override the specifications of requirements with a lower one. The use of relative priorities between requirements for handling conflicting requirements is inspired by methods such as [2, 31].

Note that it is time consuming to decide for each pair of requirements which requirement has a higher priority. In order to simplify the process, DRAS assigns a unique priority to each requirement. A functional requirement priority is based both on the importance of the actions and on the system-state referred to by the requirement. For example, a requirement about emergency actions will usually have higher priority than a requirement about other actions. Also, requirements that restrict operation in emergency states will usually have higher priority than requirements for general states.

The decision about a requirement priority is not always deterministic and the final decision is usually done (or at least validated) manually, based on experience, domain knowledge, understanding the customer needs, etc.

For example, the analysis of requirements priorities can be used to resolve the conflict between *R6* and *R7*:

- *R6. Illegitimate users shall not be allowed to initiate calls.*
- *R7. All users should be allowed to initiate a call to the police (an emergency number).*

The resolution whether an illegitimate user can dial the police or not, can only be performed manually. That is, we must determine which of these two requirements has a higher priority to define the proper DR.

It should be noted that assigning a unique priority per requirement is a simplification, as the requirements priorities are not necessarily totally ordered. That is, even if *Req A* has a higher priority than *Req B* and *Req B* has a higher priority than *Req C*, *Req C* still may have higher priority than *Req A*. Thus, using a unique priority per requirement can only suggest which requirement has a higher priority. A main reason for this is that many of the requirements are unrelated, so it difficult to compare their relative priority. Also, requirements may refer to more than one action, and each reference to an action may have its own priority.

For example:

- *R14. When pressing PTT, the phone shall initiate a call.*
- *R15. Illegitimate users shall not be allowed to transmit.*
- *R16. The phone shall send its location to the system every minute.*
- *R17. During a call, the phone shall not transmit its location.*

As initiating a call requires transmission, *R15* is assigned a higher priority than *R14*. However, although sending location to the system requires transmission as well, it still may be allowed for illegitimate users, e.g., to allow locating the phone in case of emergency. Therefore, *R16* is assigned a higher priority than *R15*, and thus also an higher priority than *R14*. However, due to *R17* (which is the result of a technical limitation of the system), initiating a call will stop sending the location for the duration of the call. If *R17* has a higher priority than *R16*, *R14* should also have higher priority than *R16* to allow initiating calls. Otherwise, *R17* has no meaning. Thus, different considerations lead to different relative priorities of *R14* and *R16*. Hence, the relative priorities between the requirements are not totally ordered.

3.4.2. *Requirements Modes and Conflicting Modes.* DRAS uses identified *modes* (and *states*) that each requirement refers to. Normally, when two requirements relate to two orthogonal modes, the requirements are not in conflict. That is, even if the two requirements use the same (implied) action, we can still assume that they do not have match-points. Examples for modes of a cellular phone are: a) is it in a call or not; b) is the user in the dialing a number; c) is the user reading SMS messages. As shown in Figure 2, the *Idle* and *Call* modes are orthogonal. A phone can either be in a call session or idle. Therefore a requirement that refers to the *Idle* mode will usually not conflict with a requirement referring to the *Call* mode.

However, the *Emergency* mode crosscuts both the *Idle* and the *Call* modes, since *Emergency* mode can be initiated no matter if the phone is in a call or not.

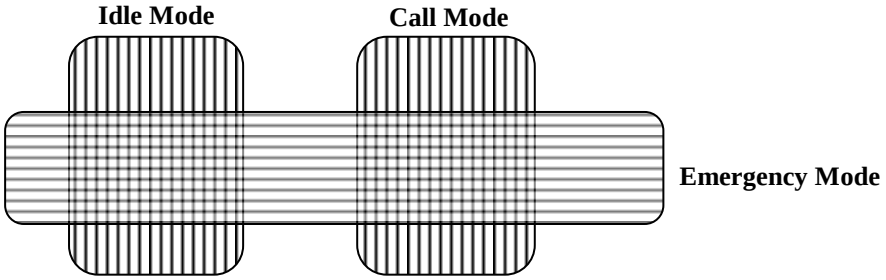


FIGURE 2. Crosscutting Modes

For example:

- *R3.* ***During a call****, the phone shall not initiate a* ***voice transmission*** *when another phone transmits.*
- *R4. In the* ***Emergency mode****, the phone should always be allowed to initiate a* ***voice transmission****.*

In this example both requirements use the same action (*transmit*), *R3* refers to the *Call* mode, and *R4* refers to the *Emergency* mode. Therefore, requirement *R4* tentatively conflicts with *R3*. Following is a possible solution to the conflict, assuming requirements related to emergency cases have higher priority than other requirements:

- *R3(E). In the Call mode, the phone shall not initiate a voice transmission when another phone transmits, unless it is in the Emergency mode.*

On the other hand, if requirement *R1* is modified to include a reference to a specific mode ("(M)" refers to "mode"), it will not conflict with requirement *R3*:

- *R1(M). In the* ***Idle mode****, the phone shall* ***initiate a call*** *when the PTT is pressed.*

Although both requirements imply the use of "transmit", none of them conflicts the other because the modes are orthogonal, i.e. there is no state of the phone when the two requirements are both relevant.

3.5. **Generating the Implied Actions List.** When searching for requirements which are based on actions that may conflict other requirements, DRAS

not only performs comparisons between actions directly used by the requirements, but also takes *implied* and *implying actions* into account.

3.5.1. *Implied and Implying Actions.* In many cases, the use of an action *Act* by a requirement implies the use of other actions by that requirement. Those are the actions which are the consequence of using *Act*. For example, the action "pressing the dial button on the phone" implies the use of the action ""transmitting voice". In addition, actions that imply the use of *Act* may also be relevant to the requirement. For example, when analyzing a requirement about "transmitting voice", the action "pressing the dial button" may also have to be considered.

We have to decide which actions to consider: those that are implied by the action *Act* or those that imply the use of *Act*. The decision depends on whether the requirement restricts the use of *Act* or whether it eases restrictions for the use of *Act*. Restricting the use of *Act* means that all actions that imply its use should also be restricted. Ease of restrictions for the use of *Act* means that all the actions that it implies should also be allowed.

Consider the case where an action (e.g., *transmit*) is restricted, i.e. it is not allowed in certain cases. The action that implies *transmit* (i.e., initiate a call) is also restricted:

- *R1. When the* ***PTT is pressed****, the phone shall* ***initiate a call****.*
- *R2. When* ***another phone transmits****, the phone shall not initiate voice* ***transmission****.*

In this case, since initiating a call requires the phone to transmit, a phone should not try to initiate a call if another phone is already transmitting. Note that this conclusion requires knowing that initiating a call results in a transmission.

3.5.2. *DRAS use of Implied and Implying Actions.* To identify the implied and implying actions for a certain action *Act*, the DRAS methodology uses predefined implied actions-list of all actions that are directly used by each action. That is, the actions-list specifies for each action A which other actions are implied by A. The implied actions-list is defined based on previous knowledge and during initial analysis of the system's requirements. Recursive use of the list allows it to identify all actions that are *implied* by the use of that action. For each implied action, the list also specifies whether it is always activated when *Act* is performed.

For example, for identifying that requirements *R1* and *R2* conflict, the list of actions implied by *call initiation*, as specified by the implied actions-list, is analyzed recursively to check if *transmit* is a result of *call initiation*. This

is shown in Figure 3 (*Tx* abbreviates transmit, see the Appendix for details about these actions).

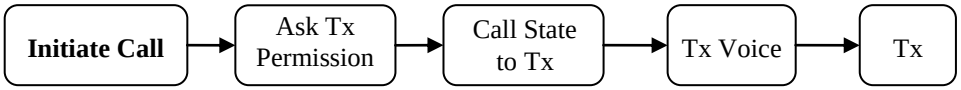


FIGURE 3. Implied Actions for Call Initiation

Given that *R2* has a higher or equal priority than *R1*, then the conflict resolution may be:

R1(E) When PTT is pressed, the phone shall initiate a call, unless another phone transmits.

A more complex example of implied actions is shown in Figure 4. It shows that several threads of actions can imply the same action; e.g., both *Power On* and *Initiate Call* imply *transmission.* Therefore, for example, not allowing transmission (Tx) means restricted functionality of call initiation and Power On.

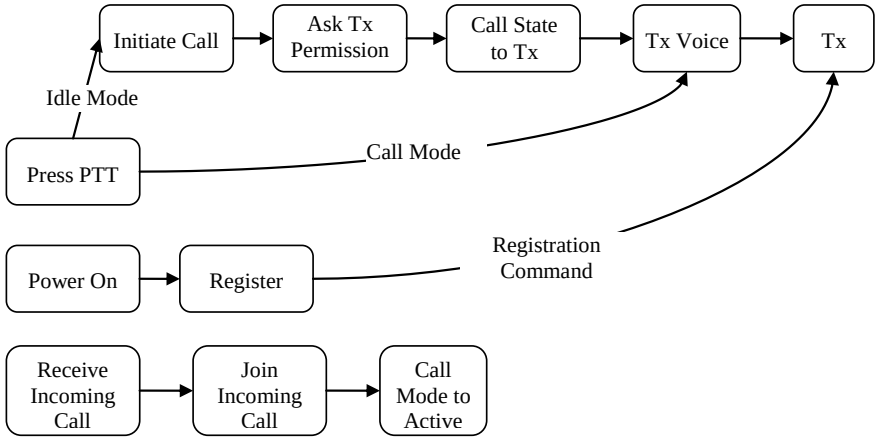


FIGURE 4. Implied Actions (a partial list)

To create the list of actions that are implied by other actions, the following steps are taken:

(1) The **list of actions** that are used by the requirements is identified.
(2) For each action, the **list of actions that they directly imply** (i.e., that are directly used by the action) are defined.

(3) The list of **all actions used by each action (either directly or indirectly)** is generated from the list of actions directly implied, as specified in the following pseudo code:

```
For each record in "Actions Implied by an Action" table{
    Create a record in "All Implied Actions" table with
    Action, Implied Action}
Repeat until no new record is added (no duplicates){
    For each two records in "All Implied Actions",
        where R1.Implied_Action = R2.Action{
           Create record in "All Implied Actions" table
           with R1.Action, R2.Implied_Action}}
```

3.6. **Identifying Implied Actions used by the Requirements.** In this step, the list of all actions used by the requirements, whether directly or indirectly (implied and implying), are identified, to ensure all actions common to the requirements are considered when checking whether a requirement conflicts another requirement. To identify whether implied or implying actions should be considered, the *action modifier* of the action in the requirement is used, as described below.

3.6.1. *Action Modifier.* Functional requirements usually conflict when they restrict normal functionality or ease other restrictions. For example, *R1* conflicts with *R2* since *R2* **restricts** the functionality of *R1*. On the other hand, *R4* **ease** the restriction of *R3* when the phone is in *Emergency* mode.

To allow identifying the type of possible conflict between requirements based on the actions they use, for each action used by a requirement, DRAS identifies its *action modifiers*. In general, an action modifier determines whether the requirement specifies when the action is used or when the action should not be used.

DRAS distinguishes between 3 action-modifiers:

- **Restrict**: an action is restricted or not allowed.
- **Unconditional**: an action is always allowed, even if it was restricted by other requirements (ease of restriction).
- **None**: an action is not specifically allowed or restricted in certain modes or states. Usually, an action with a non-action-modifier does not need to determine whether the FR conflicts other FRs or not.

The information regarding action-modifiers helps in determining whether two requirements are in conflict. If the use of an action is not restricted or a restriction for its use is not eased, then the use of the action does not necessarily mean the requirements are conflicting with other requirements. (An

exception is when there is a mistake in the requirements, such as: two contradicting requirements that are erroneously defined.) The action-modifiers are also propagated to the implied-actions.

It depends on the action modifier whether to consider the actions that are implied by an action or to consider the actions that imply the action. If a requirement **restricts** the use of action *Act*, all actions that **imply** *Act* are also restricted. For example, not allowing transmitting also means not allowing call-initiation, but not allowing call-initiation does not mean not allowing transmitting.

On the other hand, if a requirement **eases** the restrictions for using *Act* or allows using it unconditionally, then all actions **implied** by *Act* are also allowed. For example, permitting unconditional call-initiation in *Emergency* mode means also unconditional permission to transmit in this mode. Permitting unconditional transmission, however, does not mean unconditionally permitting call-initiation. The action-modifier of an action is therefore used to determine the *direction* for identifying implied-actions (see Figure 5). If *Act* is restricted, then the actions that imply *Act* are also restricted. If restrictions are eased (Unconditional), then restrictions for using the actions (implied by the action) are also eased.

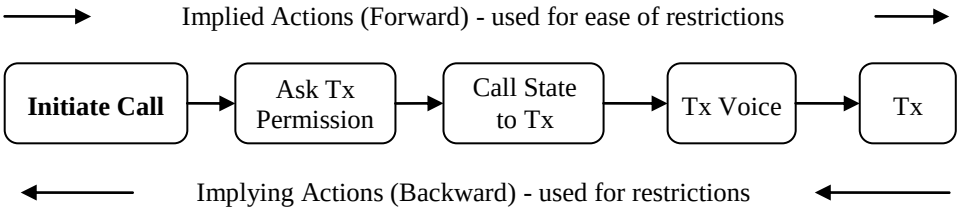


FIGURE 5. Restriction and Ease of Restriction for Implied Actions

For example, since *R2* restricts transmission, according to Figure 5, *R2* also restricts call-initiation; therefore, *R2* conflicts with *R1*.

Having only an action-modifier does not mean that one requirement may conflict with the other. Usually, in order to have a conflict the action-modifier should also contradict the action-modifier of the other requirement. For example, the following two requirements are not in conflict with each other:

R1 When the PTT is pressed, the phone shall ***initiate a call****. R4 In* ***the Emergency mode****, the phone should always be allowed to initiate a voice* ***transmission****.*

Although *R4* eases a restriction for transmission, it does not contradict *R1*, because *R1* refers to permitting transmission (implied by call initiation) and not to restricting transmission.

3.6.2. *Identifying Implied and Implying Actions.* Generating the list of actions used by a requirement is done as follows:

(1) The actions directly used by it are identified. For ach action, the *Action Modifier* is also identified, i.e. whether the action is *restricted*, it can be performed *unconditionally* (restriction is eased), or *none*.
(2) For each *restricted* action, the use of all actions implying it may also be *restricted*. The following pseudo code describes these actions list generation:

```
For each restricted Action1 used by the requirement{
    For each Action2 implying Action1{
        Add Action2 to the list of restricted actions
            used by the requirement}}
```

(3) For each *unconditional* action, the use of all actions implied by it may also be *unconditional*. The following pseudo code describes these actions list generation:

```
For each unconditional Action1 used by the requirement{
    For each Action2 implied by Action1{
        Add Action2 to the list of unconditional actions
            used by the requirement}}
```

(4) For each identified action, the *Action Modifier* of the implying or implied action and the requirement's *Priority* are assigned.

3.7. **Identifying Match-Points between the Requirements.** In this step, conflicting requirements and the requirements they are in conflict with are identified. This is performed by identifying the ***match-points*** between requirements, using their common attributes. The common attributes are *actions*, *modes*, *action modifiers* and other attributes that are common to the requirements. We also include *priorities* for the requirements.

For example, match-point will be identified between *R1* and *R2* as both use *voice transmission* (in *R1* voice transmission is implied by call initiation), and between *R1* and *R6* as they are both about *call initiation*.

Match-point identification is performed in three steps:

(1) Identify the list of match-point candidates between requirements, according to the use of common actions and different values for attributes. The following pseudo code describes this process:

```
For each pair of requirements using the same Action
(and per each common Action used by these requirements),
where the priority of the first requirement is
lower than the priority of the second requirement
and where the first requirement uses Action directly
```

```
and Action Modifier of both Actions is different {
    Add record to "Match by Action" table with Action
    and the list of attributes from both requirements}
```

(2) Remove redundancies that were created by multiple matches between two requirements (mainly caused by several implied actions that match between requirements). The following pseudo code describes this process:

```
Delete each record where a second record exists that
results from the same two requirements and with the
same Action
```

(3) Remove match-points that cannot happen in reality, or do not have at least one different conflicting mode or state.

```
Delete each record where two of its attributes contradict
or where the two requirements do not have at least one
different attribute
```

3.8. **Evaluating Match-Points.** After identifying the requirements that may conflict with each other, the effect of the conflicts should be evaluated. This is performed by identifying two attributes for each pair of matched requirements: *contribution* and *composition rules*, and is based on Brito and Moreira [7].

- ***Contribution*** - indicates how the function defined by one of the requirement affect the other requirement's function:
  - **Conflicts** with the functionality of the other requirement ("**-**")
  - **Adds** to that functionality ("**+**")
  - Does not affect the functionality ("**None**")
- ***Composition Rules*** - based on the requirements' relative priority and the nature of the conflicting functionality, the conflicting requirement can be one of the following:
  - **Overlap Before/After** - add functionality before/after the functionality of the requirement it conflicts with.
  - **Override** - replace the functionality
  - **Wrap** - encapsulate the existing functionality within new functionality.

Identifying these attributes enables an improved understanding of the nature of the conflict and how it can be resolved. After these attributes are identified, they are used for deciding how the requirements should be modified to resolve the conflicts, and to identify which of the *match-points* should result in a derived requirement.

For example, requirement *R2 conflicts* with *R1* and *override* its functionality, i.e. the phone should not initiate a call or transmit when other phone is transmitting:

- *R1. When the PTT is pressed, the phone shall* ***initiate a call****.*
- *R2. When* ***another phone transmits****, the phone shall not initiate voice* ***transmission****.*

On the other hand, *R4 overrides R2*, meaning that *R1* is allowed in emergency mode, even if other phone is transmitting.

R4 In the ***Emergency mode****, the phone should always be allowed to initiate a voice* ***transmission****.*

3.9. **Generating the Derived Requirements.** Derived requirements are generated according to the attributes defined for each *match-point*. In addition, requirements specifications may be enhanced based on the match-points attributes. Note that several *match-points* may result in one derived requirement, as several conflicts may have the same resolution.

For example, the following derived requirements may be generated (Rx/Ry indicates the matched requirements that are the source for the DRs):

- *R1/R2 When the PTT is pressed, the phone shall initiate a call, unless other phone is transmitting.*
- *R6/R7 Illegitimate users shall only be allowed to initiate a call to the police (an emergency number).*

4. Conclusion and Future Work

4.1. **Conclusion.** We have presented *DRAS*, a methodology to help identify and handle *conflicting functional requirements*. We have also applied our methodology to few test cases.

DRAS identifies conflicting functional requirements based on the actions they use. It starts with identifying the lists of actions and entities used by the input requirements. The relative priority of each requirement is also identified. Then the list of actions (implied by each action) is defined. This list is later used to identify all of the actions a requirement refers to, directly or indirectly. Generating the list depends on the action's action-modifier, i.e. whether the requirement restricts the use of an action or eases a restriction for its use. If the use of an action *Act* is restricted, the use of all actions that use *Act* (i.e. the implied-actions) is restricted too. If a requirement eases the restrictions for using an action *Act*, the actions list will include actions that are the result of using *Act* (i.e. the implying actions).

For each requirement, the modes and states of the different entities it refers to are also identified. This information is later used to help to decide whether

functional requirements conflict with each other, because this usually depends on the modes and the states referred to by the requirements.

The actions (and their modifiers) and modes for each requirement are identified. Based on this information, match-points between the requirements are identified. To get to the final list of match-points, the list is further refined to remove redundancies and conflicts that cannot occur in real-life.

The final step of DRAS is to generate DRs, according to the list of match-points between requirements. This process usually requires consulting the stakeholders; because in several cases resolving conflicts is not straight forward, and the stakeholders should decide what direction to take. The requirements, both original and derived, can be reviewed by all stakeholders, making sure that resolutions to conflicts are performed properly.

Using DRAS provides a method to identify conflicting functional requirements and the requirements they are in conflict with. It also helps in deciding what derived requirements should be generated from the conflict, by identifying the match-points between the requirements. By that, DRAS helps to define more consistent requirements.

4.2. **Further Work.** Several enhancements are considered for the DRAS methodology; mainly further automating the process beyond the partial automation of the prototype tool described in the Appendix, so that the (tentative) DRs can be generated automatically. Before automating the process, some more case studies should be done. In addition, automation requires the ability to parse and analyze the text and the ability to set the relative priorities between requirements a match-point refers. Note that text analysis should allow identifying actions, even when they are written in different forms. For example, "call initiation" may be written in the requirement "initiate a call", "start call", etc.

The DRAS methodology, or part of it, may be integrated with existing requirements management tools (such as DOORS or RequisitePro). This will enhance their functionality and enable an easier definition of requirements (derived from conflicts between other requirements). Another possible enhancement to such tools is the definition of attributes per requirement, as used in this work. For each requirement, these attributes include the actions used with their action-modifiers and the mode/state attributes. With proper textual analysis, the requirements management tool may be able to generate these attributes automatically. Using these attributes, the tool can suggest to the user possible conflicts between the requirements, by implementing similar algorithms to the ones defined for DRAS.

DRAS methodology assumes that an identified match-point (between functional requirements) tentatively identifies that one requirement is conflicting

the other. That is, one of the requirements is a conflicting requirement. This assumption was not validated; further work is required to identify whether this is true, or for what cases this is true.

Using natural language processing methods to analyze the requirements (e.g. [21, 32]), it may be possible to semi-automate actions identification of actions (used by the requirements) and their different attributes. Writing the requirements in some formal form, such as Attempto controlled language [35], can assist this approach. Ideas from AbstFinder [17] may also be used to help identify aspects in the specifications text. Mining aspects methods [23] and tools may also be used for automatic or semi-automatic retrieval and identification of aspects. Automatic weaving (composing) of requirements (to generate the DRs) may use methods similar to the ones used by aspect oriented programming (see [20]). As a starting point for using natural language processing, we consider adapting two tools that we have developed for other purposes. The EasyCRC tool [30], which automates the processes of finding nouns and synonyms, as part of its activity, can be adapted to find actions and related actions in the requirements. The CodePsychologist [27] is used to assist the programmer to locate regression bugs in the source code. It uses some affinity evaluation algorithms, which we consider using in our tool.

Using queries to identify conflicting requirements and the requirements they are in conflict with, as defined in the Requirements Description Language (RDL) [10], is another possible approach for enhancing DRAS. RDL identifies aspectual (conflicting) requirements by defining constraint queries about actions and objects used by the requirements. The requirements that the aspectual requirements tentatively are in conflict with are identified by base queries.

The use of XML to internally represent requirements can also be considered. Note that XML cannot be used to represent input and output requirements, because these should be in textual format, so that all stakeholders can understand. XML representation can help automate the creation of DRs. Methods will be needed to translate the textual requirements from text to XML (or another format) and to translate back the XML representation for DRs to textual format. XML is already used for aspect-oriented methods (e.g., the ARCaDe tool [31, 19]) to compose requirements, or for supporting aspects plug-ins in software design [22]. Concepts from these and other approaches may be reused.

To allow automatic detection of relative priorities between requirements, priorities may be added to each attribute value (e.g., Normal=1, Emergency=2). In addition to requirements priorities, this can also enable having relative priorities between requirements (i.e., a partially ordered tree of requirements

priorities). There will be no absolute priority per requirement, and the relative priority of each pair of requirements should be evaluated separately. In addition, default values for each attribute should be defined. This will enable requirements handling, where partial attribute values are specified (e.g., set call priority default as "Normal").

Composition-rules can be enhanced to improve the automation process. In many cases, current composition-rules values are not useful. Different values for composition rules, which are more suited for generating DRs, may be more useful. One possible approach is to define temporal rules, such as "Override Temporarily", "Delayed After", "On Event" (e.g., when mode changes). Enhancements using ideas from LOTOS [6, 8] should also be considered.

Temporal logic may also be used to enhance the method [24]. Action Modifiers identified in DRAS, "Restrict" and "Unconditionally" seem to be similar to Temporal Logic Path Quantifiers/Operators A/G (all paths / always) and A/H (all paths/always in the past). It may be possible to develop a logic based on temporal logic, that will use such action-modifiers and specify (using a formula), the effect of these aspectual action-modifiers on other requirements (e.g., Emergency $\rightarrow$ [A(always) PTT $\rightarrow$ Initiate Call]). The logic may be defined as an extension to already existing methods which support temporal logic for requirements, such as Formal Tropos [29] or Kaos [13]. Using formal languages that use temporal logic may allow the use of model checking methods [24] to identify conflicting requirements. The ideas suggested by [19] for the use of temporal logic in the PROBE framework can also be used as input for enhancements.

References

[1] J. Araújo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdogan. Early aspects: The current landscape. *Technical Notes, CMU/SEI and Lancaster University*, 2005.

[2] E. Baniassad and S. Clarke. Finding aspects in requirements with Theme/Doc. In B. Tekinerdoğan, P. Clements, A. Moreira, and J. Araújo, editors, *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, pages 15–22, March 2004.

[3] E. L. A. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *ICSE*, pages 158–167. IEEE Computer Society, 2004.

[4] D. Bar-On and S. S. Tyszberowicz. Derived requirements generation: The DRAS methodology. In *SwSTE*, pages 116–126. IEEE Computer Society, 2007.

[5] D. Bar-On and S. S. Tyszberowicz. Aspects, dependencies and interactions: report on the workshop ADI at ECOOP 2007. In *Proceedings of the 2007 conference on Object-oriented technology*, ECOOP'07, pages 5–10, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, March 1987.

[7] I. Brito and A. Moreira. Towards a composition process for aspect-oriented requirements. In J. Araújo, A. Rashid, B. Tekinerdogan, A. Moreira, and P. Clements, editors, *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design*, Mar. 2003.

[8] I. Brito and A. Moreira. Integrating the NFR framework in a RE mode. In *EA-AOSD: Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with the 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK*, March 2004.

[9] I. Brito, A. Moreira, and J. Araújo. A requirements model for quality attributes. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, March 2002.

[10] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *Proceedings of the 6th international conference on Aspectoriented software development AOSD*, pages 36–48. ACM Press, 2007.

[11] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdoğan, S. Clarke, and A. Jackson. Survey of aspect-oriented analysis and design approaches. Technical Report D11 AOSD-Europe-ULANC-9, May 2005.

[12] L. Chung, B. Nixon, and E. Yu. *Non-Functional Requirements in Software Engineering*. Kluwer International Series in Software Engineering. Kluwer Academic, 2000.

[13] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. In *Science of Computer Programming*, volume 20, pages 3–50. Elsevier, April 1993.

[14] G. M. C. de Sousa and J. Castro. Towards a goal-oriented requirements methodology based on the separation of concerns principle. In L. E. G. Martins and X. Franch, editors, *WER*, pages 223–239, 2003.

[15] A. Finkelstein and I. Sommerville. The view point FAQ. *Software Engineering Journal*, 1(11):2–4, 1996.

[16] M. Fowler and K. Scott. *UML distilled - a brief guide to the Standard Object Modeling Language (2nd edition)*. Addison-Wesley-Longman, 2000.

[17] L. Goldin and D. M. Berry. AbstFinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Automated Software Engg.*, 4(4):375–412, October 1997.

[18] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[19] S. Katz and A. Rashid. From aspectual requirements to proof obligations for aspect-oriented systems. In *12th IEEE International Conference on Requirements Engineering (RE 2004), 6-10 September 2004, Kyoto, Japan*, pages 48–57. IEEE Computer Society, 2004.

[20] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[21] D. Lin and P. Pantel. Discovery of inference rules from text. In *Seventh ACM SIGKDD international conference on Knowledge discovery and data mining, USA*, pages 323–328, August 2001.

[22] C. V. Lopes and T. C. Ngo. The aspect oriented markup language and its support of aspect plugins. Technical Report UCI-ISR-04-8, October 2004.

[23] N. Loughran and A. Rashid. Mining aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, March 2002.

[24] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1992.

[25] A. Moreira, J. Araújo, and I. Brito. Crosscutting quality attributes for requirements engineering. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 167–174. ACM Press, 2002.

[26] J. Mylopoulos, L. Chung, S. S. Liao, H. Wang, and E. S. K. Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, 2001.

[27] D. Nir, S. S. Tyszberowicz, and A. Yehudai. Locating regression bugs. In K. Yorav, editor, *Haifa Verification Conference*, volume 4899 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2007.

[28] B. Nuseibeh. Crosscutting requirements. In G. C. Murphy and K. J. Lieberherr, editors, *AOSD*, pages 3–4. ACM, 2004.

[29] M. Pistore, A. Fuxman, Kazhamiakin, and M. R., Roveri. Formal Tropos: Language and semantics. Technical Report 4, November 2003.

[30] A. Raman and S. S. Tyszberowicz. The easycrc tool. In *ICSEA*, pages 25–31. IEEE Computer Society, 2007.

[31] A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In M. Akşit, editor, *Proceedings 2nd International Conference on Aspect-Oriented Software Development (AOSD-2003)*, pages 11–20. ACM Press, March 2003.

[32] A. Sampaio, R. Chitchyan, and P. Rayson. Ea-miner: a tool for automating aspect-oriented requirements identification. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *ASE*, pages 352–355. ACM, 2005.

[33] A. Silva. Requirements, domain and specifications: a viewpoint-based approach to requirements engineering. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 94–104, New York, NY, USA, 2002. ACM.

[34] G. Sousa, G. Silva, and J. Castro. Adapting the NFR framework to aspect-oriented requirements engineering. In *Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES)*, pages 177–192, 2003.

[35] H. Stefan. The syntax of Attempto controlled english: An abstract grammar for ace 4.0. Technical Report ifi-2004.03, 2004.

[36] Y. Yu, J. Leite, and J. Mylopoulos. From goals to aspects: Discovering aspects from requirements goal models. In *12th IEEE International Requirements Engineering Conference*, September 2004.

[37] ETSI EN 300 392-2. Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Part 2: Air Interface (AI), 2007.

Motorola Solutions Israel
E-mail address: David.Bar-On@motorolasolutions.com

School of Computer Science, The Academic College of Tel Aviv Yaffo
E-mail address: tyshbe@tau.ac.il