Volume 18 Number 3

ACTA CYBERNETICA

Szeged, 2008

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Manuscripts must be written in good English. Contributions are accepted for review with the understanding that the same work has not been published elsewhere. Papers previously published in conference proceedings, digests, preprints are eligible for consideration provided that the author informs the Editor at the time of submission and that the papers have undergone substantial revision. If authors have used their own previously published material as a basis for a new submission, they are required to cite the previous work(s) and very clearly indicate how the new submission offers substantively novel or different contributions beyond those of the previously published work(s). Each submission is peer-reviewed by at least two referees. The length of the review process depends on many factors such as the availability of an Editor and the time it takes to locate qualified reviewers. Usually, a review process takes 6 months to be completed. There are no page charges. Fifty reprints are supplied for each article published.

Manuscript Formatting Requirements. All submissions must include a title page with the following elements:

- title of the paper
- author name(s) and affiliation
- name, address and email of the corresponding author
- An abstract clearly stating the nature and significance of the paper. Abstracts must not include mathematical expressions or bibliographic references.

References should appear in a separate bibliography at the end of the paper, with items in alphabetical order referred to by numerals in square brackets. Please prepare your submission as one single PostScript or PDF file including all elements of the manuscript (title page, main text, illustrations, bibliography, etc.). Manuscripts must be submitted by email as a single attachment to either the most competent Editor, the Managing Editor, or the Editor-in-Chief. In addition, your email has to contain the information appearing on the title page as plain ASCII text. When your paper is accepted for publication, you will be asked to send the complete electronic version of your manuscript to the Managing Editor. For technical reasons we can only accept files in LaTeX format.

Subscription Information. Acta Cybernetica is published by the Institute of Informatics, University of Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. Subscription rates for one issue are as follows: 5000 Ft within Hungary, €40 outside Hungary. Special rates for distributors and bulk orders are available upon request from the publisher. Printed issues are delivered by surface mail in Europe, and by air mail to overseas countries. Claims for missing issues are accepted within six months from the publication date. Please address all requests to:

Acta Cybernetica, Institute of Informatics, University of Szeged
P.O. Box 652, H-6701 Szeged, Hungary
Tel: +36 62 546 396, Fax: +36 62 546 397, Email: acta@inf.u-szeged.hu

Web access. The above informations along with the contents of past issues are available at the Acta Cybernetica homepage http://www.inf.u-szeged.hu/actacybernetica/ .

Bálint Dömölki
IQSOFT
Budapest, Hungary
domolki@iqsoft.hu

Zoltán Ésik
Department of Foundations of
Computer Science
University of Szeged
Szeged, Hungary
ze@inf.u-szeged.hu

Zoltán Fülöp
Department of Foundations of
Computer Science
University of Szeged
Szeged, Hungary
fulop@inf.u-szeged.hu

Ferenc Gécseg
Department of Computer Algorithms
and Artificial Intelligence
University of Szeged
Szeged, Hungary
gecseg@inf.u-szeged.hu

Jozef Gruska
Institute of Informatics/Mathematics
Slovak Academy of Science
Bratislava, Slovakia
gruska@savba.sk

Tibor Gyimóthy
Department of Software Engineering
University of Szeged
Szeged, Hungary
gyimothy@inf.u-szeged.hu

Helmut Jürgensen
Department of Computer Science
Middlesex College
The University of Western Ontario
London, Canada
helmut@csd.uwo.ca

Alice Kelemenová
Institute of Computer Science
Silesian University at Opava
Opava, Czech Republic
Alica.Kelemenova@fpf.slu.cz

László Lovász
Department of Computer Science
Eötvös Loránd University
Budapest, Hungary
lovasz@cs.elte.hu

Gheorghe Păun
Institute of Mathematics of the
Romanian Academy
Bucharest, Romania
George.Paun@imar.ro

András Prékopa
Department of Operations Research
Eötvös Loránd University
Budapest, Hungary
prekopa@cs.elte.hu

Arto Salomaa
Department of Mathematics
University of Turku
Turku, Finland
asalomaa@utu.fi

László Varga
Department of Software Technology
and Methodology
Eötvös Loránd University
Budapest, Hungary
varga@ludens.elte.hu

Heiko Vogler
Department of Computer Science
Dresden University of Technology
Dresden, Germany
vogler@inf.tu-dresden.de

Gerhard J. Woeginger
Department of Mathematics and
Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
gwoegi@win.tue.nl

Conference of PhD Students in Computer Science

Guest Editor:

Tibor Csendes

Department of Applied Informatics
University of Szeged
Szeged, Hungary
csendes@inf.u-szeged.hu

Preface

The 5th Conference for PhD Students in Computer Science (CSCS) was organized by the Department of Computer Science of the University of Szeged (SZTE) and held in Szeged, Hungary from June 27-30, 2006. The members of the Scientific Committee were the following representants of the Hungarian doctoral schools in computer science: Mátyás Arató (DE), András Benczúr (ELTE), Miklós Bartha (SZTE), Tibor Csendes (SZTE), János Csirik (SZTE), János Demetrovics (SZTAKI), Sarolta Dibuz (Ericsson), József Dombi (SZTE), Zoltán Ésik (SZTE), Ferenc Friedler (VE), Zoltán Fülöp (SZTE), Ferenc Gécseg (chair, SZTE), Tibor Gyimóthy (SZTE), Balázs Imreh (SZTE), János Kormos (DE), László Kozma (ELTE), Attila Kuba (SZTE), Eörs Máté (SZTE), Gyula Pap (DE), András Recski (BMGE), Endre Selényi (BMGE), Katalin Tarnay (NOKIA), György Turán (SZTE), and László Varga (ELTE). The members of the Organizing Committee were Balázs Bánhelyi, Tibor Csendes (chair), Judit Jász, Mariann Kocsorné Sebő, Gabriella Nagyné Hecskó, and Péter Gábor Szabó.

There were more than 100 participants and 75 talks in several fields of computer science and its applications. Beyond the Hungarian PhD schools in computer science, 8 other European countries were represented. The talks were going in two parallel sections in artificial intelligence, automata and formal languages, computer networks, database theory, discrete mathematics, fuzzy decision support systems, information systems, optimization, picture processing, and software engineering. The talks of the students were completed by 4 plenary talks of leading scientists: Tibor Gyimóthy (Szeged), Marius Minea (Timisoara), Lajos Rónyai (Budapest), and Hermann Schichl (Vienna).

Three scientific journals, viz. Periodica Polytechnica (Budapest), Publicationes Mathematicae (Debrecen) and Acta Cybernetica (Szeged) offered students to publish the paper version of their presentations after a selection and review process. Altogether 19 manuscripts were submitted for publication. The present special issue of Acta Cybernetica contains 7 such papers.

The full program of the conference, the collection of the abstracts and further information can be found at `http://www.inf.u-szeged.hu/~cscs`.

On the basis of our repeated positive experiences, the conference will be organized in the future, too, hopefully with more foreign participants. According to the present plans, the next meeting will be held in July 2008 in Szeged.

Tibor Csendes

Acta Cybernetica **18** (2008) 367–377.

On the Ambiguity of Reconstructing *hv*-Convex Binary Matrices with Decomposable Configurations*

Péter Balázs†

Abstract

Reconstructing binary matrices from their row, column, diagonal, and antidiagonal sums (also called projections) plays a central role in discrete tomography. One of the main difficulties in this task is that in certain cases the projections do not uniquely determine the binary matrix. This can yield an extremely large number of (sometimes very different) solutions. This ambiguity can be reduced by having some prior knowledge about the matrix to be reconstructed. The main challenge here is to find classes of binary matrices where ambiguity is drastically reduced or even completely eliminated. The goal of this paper is to study the class of *hv*-convex matrices which have decomposable configurations from the viewpoint of ambiguity. First, we give a negative result in the case of three projections. Then, we present a heuristic for the reconstruction using four projections and analyze its performance in quality and running time.

Keywords: discrete tomography; *hv*-convex binary matrix; decomposable configuration; reconstruction algorithm

The reconstruction of binary matrices from their projections is a basic problem in discrete tomography. Binary matrices can represent two-dimensional cross-sections of an object made (or consisting) of homogeneous material, while one can think of projections as the numerical results of measuring the density of the object in the given cross-section along certain directions. Reconstruction algorithms have a wide area of applications in non-destructive testing, biplane angiography, crystallography, radiology, image processing, and so on. For a detailed overview of the main problems and applications of discrete tomography the reader is referred to [11, 12]. For practical reasons the projections in most cases can be taken only from few (usually at most four) directions. This often leads to ambiguous reconstruction, i.e., the reconstructed matrix can be quite dissimilar to the original one which is inappropriate for applications [1, 14]. One commonly used technique to reduce ambiguity is to use some a priori information of the matrix to be reconstructed. In this paper we investigate the problem of ambiguity if the matrix to

*This work was supported by OTKA grant T048476.

†Department of Image Processing and Computer Graphics, University of Szeged, 6720 Szeged, Árpád tér 2., Hungary, E-mail: pbalazs@inf.u-szeged.hu

be reconstructed is *hv*-convex and its components form a so-called *decomposable configuration*. First, we give a construction to prove that the use of only three projections is not sufficient to eliminate ambiguity, that is, for some inputs there can be exponentially many *hv*-convex decomposable binary matrices having the same horizontal, vertical and diagonal projections. In the case of four projections we are facing the following problem. Although all the *hv*-convex decomposable binary matrices with the given four projections can be reconstructed in polynomial time [2] this class is not explicitly defined. In more detail, one criterion for decomposability is that the components of the binary matrix are uniquely reconstructible from their horizontal and vertical projections. However, when reconstructing *hv*-convex matrices with the algorithm of [2] it cannot be decided in advance whether this criterion is satisfied. If so then, clearly, the algorithm gives correct solutions. However, in some cases the algorithm gives a solution even if the above criterion is not fulfilled, i.e, if one or more of the components are not uniquely determined by the horizontal and vertical projections. We conduct experiments to investigate whether the above criterion is often implicitly satisfied. Since components of an *hv*-convex binary matrix are necessarily *hv*-convex polyominoes, we investigate the possibility that an *hv*-convex polyomino is uniquely determined by its horizontal and vertical projections. We also study how the knowledge of a component's third projection affects ambiguity, and based on the observations we develop a fast and accurate reconstruction heuristic for the class of *hv*-convex binary matrices with decomposable configurations.

This contribution is structured as follows. First, the necessary preliminaries are introduced in Section 1. In Section 2 we show that in the class of *hv*-convex decomposables there could be a large number of ambiguous reconstructions, if we use only three projections. In Section 3 we extend the method published in [2] for reconstructing *hv*-convex binary matrices with decomposable configurations even if it is not guaranteed that the components are uniquely determined by the horizontal and vertical projections, and analyze the performance of the developed algorithm. Finally, in Section 4 we discuss our results.

1 Preliminaries

Discrete sets (the finite subsets of the two-dimensional integer lattice) are highly important in discrete tomography. A discrete set with the smallest containing discrete rectangle (SCDR) of size $m \times n$ can be represented by a binary matrix $F = (f_{ij})_{m \times n}$ where the 1s in the matrix are representing that the corresponding element of the 2D lattice belongs to the discrete set (see Fig. 1). Based on this correspondence, in the sequel, when we are using the term discrete set we always mean the set of positions of F having value 1. Analogously, the *size* of the discrete set is defined by the size of its SCDR (or equivalently, the size of its representing matrix F). To avoid confusion we stress that the size of the discrete set is not the number of its elements (see, again, the discrete set of Fig. 1 which is of size 5×5 but has 14 elements). The *horizontal* and *vertical projections* of F are the vectors

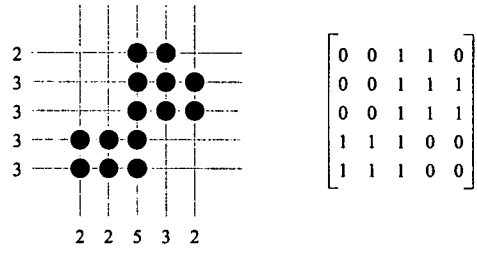


Figure 1: An *hv*-convex polyomino and the corresponding binary matrix. The elements of the discrete set are marked with black dots. The projections of the polyomino are the vectors $H = (2,3,3,3,3)$, $V = (2,2,5,3,2)$, $D = (0,2,3,2,1,1,2,2,1)$, and $A = (0,0,1,3,4,4,2,0,0)$.

$\mathcal{H}(F) = H = (h_1,\ldots,h_m)$, and $\mathcal{V}(F) = V = (v_1,\ldots,v_n)$, respectively, where

$$h_i = \sum_{j=1}^{n} f_{ij} \quad (i = 1,\ldots,m) \quad \text{and} \quad v_j = \sum_{i=1}^{m} f_{ij} \quad (j = 1,\ldots,n)\,. \tag{1}$$

Similarly, the *diagonal* and *antidiagonal projections* of F are defined by $\mathcal{D}(F) = D = (d_1,\ldots,d_{m+n-1})$, and $\mathcal{A}(F) = A = (a_1,\ldots,a_{m+n-1})$, respectively, where

$$d_k = \sum_{i+(n-j)=k} f_{ij} \quad \text{and} \quad a_k = \sum_{i+j=k+1} f_{ij} \quad (k = 1,\ldots,m+n-1)\,. \tag{2}$$

Two positions $P = (p_1,p_2)$ and $Q = (q_1,q_2)$ in a discrete set are said to be *4-adjacent* if $|p_1 - q_1| + |p_2 - q_2| = 1$. The positions P and Q are *4-connected* if there is a sequence of distinct positions $P_0 = P,\ldots,P_k = Q$ in the discrete set F such that P_l is 4-adjacent to P_{l-1} for each $l = 1,\ldots,k$. A discrete set F is *4-connected* if any two points in F are 4-connected. The 4-connected discrete set is also called *polyomino*. A maximal 4-connected subset of a discrete set F is called a *component* of F. In particular, every polyomino consists of a single component. The discrete set F is *hv-convex* if all the rows and columns of F are 4-connected (see Fig. 1).

Given an ordered pair of binary matrices (C,D) we say that we get the binary matrix F by *NorthWest-gluing* (or shortly, NW-gluing) C to D if

$$F = \begin{pmatrix} C & \mathbf{0} \\ \mathbf{0} & D \end{pmatrix}\,. \tag{3}$$

If C is a polyomino then we say that C is the *NW-component of* F. NE-, SE-, SW-gluings and -components are defined similarly. A discrete set F consisting of $k \geq 2$ components is *decomposable* if all of the following properties are fulfilled

(α) the components of F are uniquely reconstructible from their horizontal and vertical projections in polynomial time,

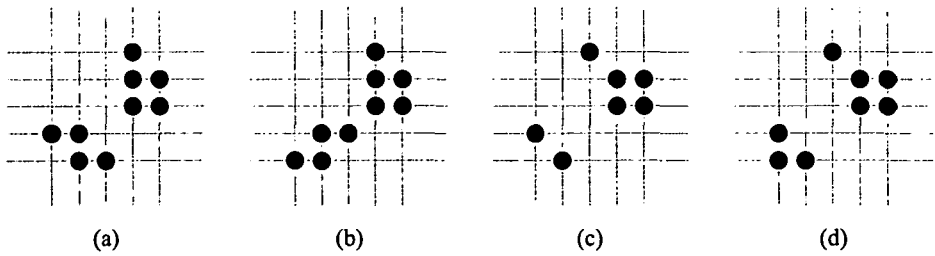


Figure 2: (a)-(c) Undecomposable and (d) decomposable *hv*-convex discrete sets.

(β) the sets of the row and column indices of the components' SCDRs are disjoint,

(γ) if $k > 2$ then we get F by gluing a single component to a decomposable discrete set consisting of $k-1$ components using one of the four gluing operators.

If F satisfies properties (β) and (γ) but not necessarily property (α) then we say that the components of F are in a *decomposable configuration.* Obviously, every *hv*-convex set satisfies property (β). The discrete set in Fig. 1 is undecomposable since it consists of only one component. Figures 2a and 2b show a situation where the components are in a decomposable configuration but property (α) does not hold since the bottom left components of both sets have the same horizontal and vertical projections. The discrete set in Fig. 2c does not satisfy property (γ) while the set shown in Fig. 2d is decomposable.

The reconstruction task aims to find a discrete set F such that $\mathcal{H}(F) = H$ and $\mathcal{V}(F) = V$ for given vectors H and V (in the case of four projections two more vectors D and A are also given, i.e., $\mathcal{D}(F) = D$ and $\mathcal{A}(F) = A$ must also hold). Not any pair (or 4-tuple) of vectors are projections of a discrete set (see, e.g., [16] for a necessary and sufficient condition in the case of two projections). However, in some cases there can be several (and also very dissimilar) solutions with the same projections (see, e.g, [14]). This latter feature of the reconstruction is the so-called ambiguity, a problem one tries to avoid during the reconstruction. One of the most frequently used techniques to reduce ambiguity is to suppose that the set to be reconstructed belongs to a certain class of discrete sets having some geometrical properties. There are classes of discrete sets where ambiguity is completely eliminated (see [3, 4, 8]). Furthermore, for certain classes it was shown that only a polynomial number of discrete sets with the same projections can belong to the given class [2, 5]. Finally, for some classes it is known that ambiguity in those classes can be exponentially large [4, 8, 10]. In this paper we are going to investigate the problem of ambiguity in the class of *hv*-convex discrete sets having decomposable configurations.

2 Three Projections: A Negative Result

In [2] it was shown that every *hv*-convex decomposable discrete set having the same horizontal, vertical, diagonal, and antidiagonal projections can be reconstructed in polynomial time. Clearly, this also means that the number of solutions is polynomial, too. However, the question was left open whether the use of four projections is necessary to achieve this result. The following theorem gives an answer to this question.

Theorem 1. *For some vectors H, V, and D there can be exponentially many hv-convex decomposable binary matrices with the same horizontal, vertical, and diagonal projections H, V, D, respectively.*

Proof. Consider the following matrices

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad M' = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} . \tag{4}$$

Clearly, M and M' are decomposable and have the same horizontal, vertical, and diagonal projections. Now, for a given $k \geq 1$ and for any $S \subseteq \{1, \ldots, k\}$ let the matrix X_k^S be defined as follows

$$X_k^S = \begin{pmatrix} M_1 & & & \\ & M_2 & & \\ & & \ddots & \\ & & & M_k \end{pmatrix} \quad \text{where} \quad M_i = \begin{cases} M & \text{if } i \in S \\ M' & \text{if } i \notin S \end{cases} \tag{5}$$

for every $i = 1, \ldots, k$. The matrices defined by (5) are, certainly, *hv*-convex and decomposable and have the same horizontal, vertical, and diagonal projections. S can be any subset of $\{1, \ldots, k\}$ which gives 2^k matrices with the described properties. □

As a consequence of this theorem we get

Corollary 1. *If there is an algorithm that reconstructs all the hv-convex decomposable binary matrices with the horizontal, vertical, and diagonal projections H, V, and D, respectively, then there are some vectors H, V, and D for which the time complexity of the algorithm is exponential.*

Remark 1. *Naturally, we get the same results replacing the diagonal projections with the antidiagonal projections.*

3 Four Projections: A Reconstruction Heuristic

The reconstruction of a discrete set from four projections is NP-hard [10]. Furthermore, the number of *hv*-convex discrete sets having the same four projections can

be extremely large. This can be shown, e.g., in a similar way as in the proof of Theorem 1 but using the matrices

$$M = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad \text{and} \quad M' = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{6}$$

However, as we mentioned in Section 2 if we assume that the set is *hv*-convex and decomposable then every discrete set with the given four projections can be found in polynomial time yielding a polynomial number of solutions. Before going further we describe a somewhat modified version of the algorithm given in [2].

Algorithm 1.

Input: Four vectors, $H \in \mathbb{N}^m$, $V \in \mathbb{N}^n$, $D, A \in \mathbb{N}^{m+n-1}$.
Output: The binary matrix F with $\mathcal{H}(F) = H$, $\mathcal{V}(F) = V$, $\mathcal{D}(F) = D$, and $\mathcal{A}(F) = A$ or the message "no solution".

1: $F = \emptyset$;
2: try to find the bottom right corner (i, j) of a component in the NW-corner;
3: **if** (Step 2 succeed) **then** reconstruct all the *hv*-convex polyominoes with horizontal and vertical projections $(h_1, \ldots, h_i)$ and $(v_1, \ldots, v_j)$, respectively; **else goto** Step 5;
4: select randomly a polyomino P with $\mathcal{A}(P) = (a_1, \ldots, a_{i+j-1})$ from the candidates generated in Step 3;
5: **if** (no component) **then** try to decompose a component in the NE-corner using the vectors H, V, and D similarly as in Steps 2, 3, and 4;
if (no component) **then** try to decompose a component in the SE-corner using the vectors H, V, and A similarly as in Steps 2, 3, and 4;
if (no component) **then** try to decompose a component in the SW-corner using the vectors H, V, and D similarly as in Steps 2, 3, and 4;
if (no component) **then goto** Step 6;
else { update H, V, D, and A according to the projections of P;
$F = F \cup P$;
goto Step 2; }
6: try to reconstruct the last component and update the vectors;
7: **if** $(H = V = D = A = 0)$ **then return** F **else** FAIL (no solution);

The algorithm in its original form reconstructs *hv*-convex decomposable discrete sets with the given four projections in polynomial time by decomposing a component (which is an *hv*-convex polyomino) in each step of the main loop. Since in that class property (α) is satisfied the components are reconstructed from their horizontal and vertical projections uniquely. However, when reconstructing a component we always have a third projection which is not used directly for reconstruction but only for testing whether the reconstructed polyomino in the given corner has

Table 1: The number of *hv*-convex polyominoes in the test data sets that are not uniquely determined by two, three, and four projections.

Size $n \times n$	H, V	H, V, D	H, V, A	H, V, D, A
4×4	1393	40	52	18
5×5	1442	33	36	16
7×7	967	13	8	2
10×10	586	4	6	1
20×20	312	2	1	1
40×40	210	1	0	0
60×60	162	1	0	0
80×80	148	0	0	0
100×100	171	0	0	0

the proper (diagonal or antidiagonal) third projection. Interestingly, in some cases the algorithm also gives a solution even if one or more of the components are not uniquely determined by the horizontal and vertical projections, i.e., if there are ambiguous reconstructions for some of the components (see [2] for further details). Algorithm 1 exploits this feature of the original algorithm to serve as a heuristic for the broader class of *hv*-convexes with decomposable configurations. The idea of our reconstruction heuristic is that we try to eliminate ambiguity by using directly the third projection in the reconstruction. In Step 3 of Algorithm 1 we reconstruct all candidates for a component from the horizontal and vertical projections. Then, in Step 4 we choose one of them that has the proper third projection. Note that if the discrete set to be reconstructed satisfies property (α), too, then Steps 3 and 4 together yield the original form of the algorithm presented in [2].

We have conducted experiments for investigating how ambiguity of the components (which are *hv*-convex polyominoes) can affect the performance of Algorithm 1. Using the methods given in [13], we have generated *hv*-convex polyominoes with different sizes sampled from a uniform random distribution. Each test data set consisted of 5000 discrete sets with the same size. The second column of Table 1 represents the number of polyominoes in the test data sets that have ambiguous solutions when only two projections are used to reconstruct them. Note that unless the size of the polyomino is small ambiguity occurs in only 3-6% of the cases (these results of this column are essentially the same that were established independently by a similar investigation in [7]). This means that if the components of an *hv*-convex discrete set form a decomposable configuration and they are relatively big then it is very likely that the set will be decomposable, i.e., it will satisfy property (α), too. Clearly, the more components the discrete set has, the less likely it is that all the components are uniquely determined by the horizontal and vertical projections. Moreover, if the set has small components then ambiguity can occur more likely – possibly causing the algorithm to fail.

The accuracy of Algorithm 1 depends on whether the third projection can effec-

tively eliminate the ambiguity. The third and fourth columns of Table 1 represent the number of polyominoes in the test data sets that are not uniquely determined by three projections (clearly, due to symmetry the two columns have nearly the same entries). These results show that if the size of the polyomino is greater than 3×3 then ambiguity occurs in less than 1% of the cases, and it reduces drastically as the size of the set increases. Again, the more components the discrete set has the more likely ambiguity occurs (since it can occur in any component). If we have several candidates with the same three projections for a certain component then the only thing that affects the remaining part of the reconstruction is the fourth projection of the component. The fifth column of Table 1 shows that even in the cases if there are several candidates with the same three projections, it still has a small probability that the fourth projection of the chosen set will be the same as the true component's one, and thus the algorithm will not fail.

The computational cost of Algorithm 1 mostly depends on whether Step 3 can be performed fast. In this step we reconstruct *hv*-convex polyominoes from their horizontal and vertical projections. Several algorithms have been developed for solving this problem (see [6] for a comparison of them). However, all of them can find only one of the solutions in polynomial time. Since the number of *hv*-convex polyominoes with the same horizontal and vertical projections can be exponentially large [9] executing Step 3 in some cases can take an exponential time. Fortunately, in average case this task can be performed in a few hundredths seconds even on a PC with a processor of only 533 MHz [6].

Based on the observations that all the *hv*-convex polyominoes having the same two projections can be found quite fast, and the number of ambiguous cases is very small if three projections are used to reconstruct them, we expect our newly developed heuristic to reconstruct *hv*-convex discrete sets having decomposable configurations (i.e., if property (α) might not hold) fast and in most cases accurate. In order to support this claim we have conducted some experiments. We have generated 5 data sets, each of them contained 1000 *hv*-convex sets with decomposable configurations having k components of size $n \times n$ for some fixed k and n. The generation method was the following. Again, using the methods given in [13], we have generated a sequence of k *hv*-convex polyominoes of size $n \times n$ sampled from a uniform random distribution. Then, we have generated a random sequence of the elements NW, NE, SE, and SW. If the discrete set to be generated consisted of k components then the length of the sequence was $k - 1$, and it represented the way and order of how the k components should be glued together. For the 5 test data sets we have chosen the parameters k and n as follows:

- Test 1: $k = 10$, $n = 5$;
- Test 2: $k = 20$, $n = 5$;
- Test 3: $k = 30$, $n = 5$;
- Test 4: $k = 10$, $n = 10$;
- Test 5: $k = 20$, $n = 10$.

Table 2: Accuracy and average running time of Algorithm 1 on the test data sets.

Test	#correct sol.	#incorrect sol.	#no sol.	time (s)
Test 1	939	14	47	0.600
Test 2	891	27	82	0.847
Test 3	851	41	108	2.322
Test 4	998	0	2	0.660
Test 5	994	0	6	5.676

For example, Test 1 contained 1000 discrete sets of size 50×50, and each of them had 10 components of size 5×5. The reconstruction heuristic was implemented in C++ and the test run on an Intel Pentium IV 3.2GHz with 1GB RAM under Debian GNU/Linux 3.1. Table 2 shows the average running times, and the number of correct and incorrect solutions for the 5 test data sets. From the entries of Table 2 we can deduct that the number of incorrect solutions increases as the number of components increases. But we should mention here that in the first three tests an inaccurate reconstruction differed from the original set just in one component, and just in 8 positions. More precisely, the original and the reconstructed components always formed a pair like

$$M = \begin{pmatrix} 0 & 0 & X & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ X & 1 & 1 & 1 & X \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & X & 0 & 0 \end{pmatrix} \quad \text{and} \quad M' = \begin{pmatrix} 0 & 1 & X & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ X & 1 & 1 & 1 & X \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & X & 1 & 0 \end{pmatrix}, \tag{7}$$

where the positions marked by X can take arbitrary values and are the same in M and M'. Moreover, according to Table 1 if the set has components of size 10×10 (or bigger) then the algorithm can reconstruct the set in almost all cases, accurately. In fact, in Tests 4 and 5 we did not find inaccurate reconstructions (although it has a small probability that an inaccurate solution will be reconstructed). Evidently, the larger the set is the more time is needed to reconstruct it, but even for the biggest sets of Test 5 the average running time of the algorithm is very fast. We should also add that the implementation was not optimised on time. Summarizing this we can say that Algorithm 1 has really good performance in both quality and running time.

4 Conclusions and Further Research

We have studied the problem of ambiguity for reconstructing *hv*-convex discrete sets which have decomposable configurations. We have shown that if only the horizontal, vertical, and one of the diagonal projections of the set to be reconstructed are known then the number of solutions with the same projections can be extremely

large. It is an open question whether in this case a reconstruction algorithm can be given to find a solution in polynomial time. If we assume that all four projections of the discrete set are given then the reconstruction of *hv*-convex decomposables can be achieved in polynomial time. We extended this reconstruction algorithm to the more general case when the components of the *hv*-convex set are not necessarily uniquely determined by their two projections but still form a decomposable configuration. Although the extended algorithm in some cases uses exponential time to reconstruct a solution, experimental results show that the average running time of the enhanced algorithm is very fast. The algorithm in some cases does not find a solution (or not the original one). However, our investigation also shows that this happens very rarely (especially if the set has components of size larger than 5×5).

In [15] an algorithm is presented to reconstruct *hv*-convex discrete sets from horizontal and vertical projections. The worst case time complexity of this algorithm is exponential, too [17]. This algorithm is suitable to reconstruct *hv*-convex sets from four projections as well. In this case one should simply reconstruct every *hv*-convex discrete set which have the same given horizontal and vertical projections and then keep only the solutions that also have the proper diagonal and antidiagonal projections. It would be interesting to compare the average performance of this algorithm to our newly developed one. However, the generation of general *hv*-convex sets using uniform random distributions is an unsolved problem, therefore no method is known by which the comparison on the whole set of *hv*-convexes could be done. In the future we want to search for subclasses of the class of *hv*-convexes whose elements can be generated using uniform random distributions, and which are general enough for doing significant comparisons.

Acknowledgments

The author of this paper would like to thank the anonymous reviewer for his suggestions which considerably improved the quality of the paper.

References

[1] A. Alpers, P. Gritzmann, L. Thorens, Stability and instability in discrete tomography, *Proceedings of Digital and Image Geometry, Lecture Notes in Comput. Sci.*, **2243** (2001) 175–186.

[2] P. Balázs, A decomposition technique for reconstructing discrete sets from four projections, *Image and Vision Computing*, accepted.

[3] P. Balázs, Reconstruction of discrete sets from four projections: strong decomposability, *Elec. Notes in Discrete Math.*, **20** (2005) 329–345.

[4] P. Balázs, The number of line-convex directed polyominoes having the same orthogonal projections, *Proceedings of the 13th International Conference on*

Discrete Geometry for Computer Imagery DGCI 2006, Lecture Notes in Comput. Sci., **4245** (2006) 77–85.

[5] P. Balázs, E. Balogh, A. Kuba, Reconstruction of 8-connected but not 4-connected *hv*-convex discrete sets, *Disc. App. Math.*, **147** (2005) 149–168.

[6] E. Balogh, A. Kuba, Cs. Dévényi, A. Del Lungo, Comparison of algorithms for reconstructing *hv*-convex discrete sets, *Lin. Alg. Appl.* **339** (2001) 23–35.

[7] S. Brunetti, A. Del Lungo, F. Del Ristoro, A. Kuba, M. Nivat, Reconstruction of 4- and 8-connected convex discrete sets from row and column projections, *Lin. Alg. Appl.* **339** (2001) 37–57.

[8] A. Del Lungo, Polyominoes defined by two vectors, *Theoret. Comput. Sci.* **127** (1994) 187–198.

[9] A. Del Lungo, M. Nivat, R. Pinzani, The number of convex polyominoes reconstructible from their orthogonal projections, *Discrete Math.* **157** (1996) 65–78.

[10] R.J. Gardner, P. Gritzmann, Uniqueness and complexity in discrete tomography, In [11] (1999) 85–113.

[11] G.T. Herman, A. Kuba (Eds.), *Discrete Tomography: Foundations, Algorithms and Applications*, Birkhäuser, Boston, 1999.

[12] G.T. Herman, A. Kuba (Eds.), *Advances in Discrete Tomography and Its Applications*, Birkhäuser, Boston, 2007.

[13] W. Hochstättler, M. Loebl, C. Moll, Generating convex polyominoes at random, *Discrete Math.* **153** (1996) 165–176.

[14] T.Y. Kong, G.T. Herman, Tomographic equivalence and switching operations, In [11] (1999) 59–84.

[15] A. Kuba, The reconstruction of two-directionally connected binary patterns from their two orthogonal projections, *Comp. Vision, Graphics, and Image Proc.* **27** (1984) 249–265.

[16] H.J. Ryser, Combinatorial properties of matrices of zeros and ones, *Canad. J. Math.* **9** (1957) 371–377.

[17] G.W. Woeginger, The reconstruction of polyominoes from their orthogonal projections, *Inform. Process. Lett.* **77** (2001) 225–229.

Acta Cybernetica 18 (2008) 379–390.

An On-line Speaker Adaptation Method for HMM-based Speech Recognizers

András Bánhalmi* and András Kocsor*

Abstract

In the past few years numerous techniques have been proposed to improve the efficiency of basic adaptation methods like MLLR and MAP. These adaptation methods have a common aim, which is to increase the likelihood of the phoneme models for a particular speaker. During their operation, these speaker adaptation methods need precise phonetic segmentation information of the actual utterance, but these data samples are often faulty.

To improve the overall performance, only those frames from the spoken sentence which are well segmented should be retained, while the incorrectly segmented data should not be used during adaptation. Several heuristic algorithms have been proposed in the literature for the selection of the reliably segmented data blocks, and here we would like to suggest some new heuristics that discriminate between faulty and well-segmented data. The effect of these methods on the efficiency of speech recognition using speaker adaptation is examined, and conclusions for each will be drawn.

Besided post-filtering the set of the segmented adaptation examples, another way of improving the efficiency of the adaptation method might be to create a more precise segmentation, which should then reduce the chance of faulty data samples being included. We suggest a method like this here as well which is based on a scoring procedure for the N-best lists, taking into account phoneme duration.

Keywords: speech recognition, speaker adaptation, faulty transcripts, confidence measures, a posteriori phoneme probabilities

1 Introduction

The probabilistic models for speech recognition are normally trained on a large amount of data samples that contain utterances recorded from many speakers. While these speaker-independent models usually operate with a quite similar and acceptable performance for most speakers, speaker-dependent models which are

*Research Group on Artificial Intelligence of the Hungarian Academy of Sciences and University of Szeged H-6720 Szeged, Aradi vértanúk tere 1., Hungary, E-mail: {banhalmi, kocsor}@inf.u-szeged.hu

trained just on the sample utterances of a particular speaker are much more efficient at the recognition task for this specific speaker. The problem with developing speaker-dependent systems for each speaker separately, however, is that large amounts of speech training data for each speaker may be unavailable and even difficult to acquire. In order to fine-tune the speaker independent model to achieve the efficiency of a speaker dependent model, the following important techniques have been already proposed.

The usual approaches for improving the performance of the speaker-independent models are the transformation of the incoming feature vectors (e.g. by VTLN, CMN or CVN) and the fine-tuning of the parameters in statistical acoustic models (speaker adaptation techniques). The aim of feature vector transformation-based methods is to transform (normalize) both the training and the test data in such a way that the classes are easier to separate. In most cases these methods try to normalize the input data with respect to a given parameter. The VTLN (vocal tract length normalization) method normalizes the spectrum of the input speech data by converting it as if all the samples had been pronounced by speakers with the same vocal tract length [10],while the basic CMN (cepstral mean normalization) method converts the cepstral coefficients of the input data so that the samples for each speaker have the same mean value [5].

The other approach for adjusting a speaker independent model in order to better approximate the performance of a speaker dependent model is speaker adaptation. In classical HMM-based systems various speaker-adaptation techniques have been used with success. These techniques fine-tune the parameters of the speaker-independent system to more 'suitable' ones corresponding to the adaptation data examples of a particular speaker.

In most cases, adaptation can be applied using three strategies: batch adaptation, self-adaptation and on-line adaptation [4, 15]. Batch adaptation performs the reestimation of the model parameters only after all the adaptation data samples have been collected, so it is an off-line method. Self-adaptation is performed on the testing data at runtime, without collecting adaptation data, so this is normally an unsupervised method. The on-line (or incremental) adaptation technique alters the parameters of the statistical model only after a block of adaptation data samples have been enrolled, and this block of data is thrown away after the adaptation method has been applied. Recognition errors and faulty transcripts pose an important problem when the above-mentioned algorithms are used. The main advantage of the on-line adaptation technique over the self-adaptation one is that it has access to much more information to separate the non-faulty adaptation data samples, which could be used in the adaptation phase with more success. The other important advantage of on-line adaptation is that it will adapt the previous model to those data samples which have the maximal probability after enrolling a block of data, hence this method should be much more stable than the self-adaptation one.

Computationally, two main approaches have been proposed in the literature for the adaptation of HMM parameters. The first is the maximum likelihood (ML) - based framework which contains the maximum likelihood linear regression (MLLR

[11]) approach, the maximum likelihood stochastic matching (SM) approach [18] and the constrained transformation approach [3].Some other adaptation techniques are based on the maximum a posteriori formulation (MAP [6]), where only the parameters of the states corresponding to the Viterbi path (the path with maximal probability values) are reestimated. Because the speech recognition algorithm in our speech recognition system works in a similar way (namely it approximates the forward probability with the maximal value along the Viterbi path), this latter technique might be more feasible in our framework than MLLR-like techniques. The second main approach contains discriminative adaptation techniques like MCE (Minimum Classification Error) [9] and MMI (Maximum Mutual Information) [16, 17], all of which try to maximize the recognition accuracy explicitly for a given vocabulary.

Our goal with the experiments presented in this paper was to improve the performance of a continuous speech recognizer by applying some modifications to the supervised adaptation process. As the first step of HMM phoneme model adaptation, the recognition system has to collect the phonetic data from the utterances of the user. To achieve this an automatic segmentation is carried out by the speech recognizer. In Section 2 we will provide a short description about of the key aspects of our framework. The automatic segmentation phase however can be faulty for a variety of reasons; e.g. the initial model is of poor quality, noise has been introduced by the microphone or by the user, or simply the user stutters, or misreads words. Our aim was to exclude these faulty segmented data items from the adaptation or at least to reduce their number. In the literature several methods have been proposed to tackle this problem. Confidence measures have been investigated to help the adaptation process deal with faulty transcripts [1, 12, 7].In the latter articles confidence measures were used to mark possible recognition errors and to exclude the erroneously segmented words from the adaptation process. In Section 3 we also propose two confidence-measure-like methods for improving the efficiency of speaker adaptation.

Actually, the frequency of the occurrence of faulty adaptation data samples can also be reduced directly at the on-line speech recognition level when the automatic segmentation for the speech signal is being performed. When the speech recognizer fills up the N-best hypothesis stacks, the scores of the hypotheses can be set to eliminate certain (possibly faulty) hypotheses from the stack. Here we give a method like this as well in Section 3.

2 The Speech Recognizer and the Adaptation Module

The speech recognizer employed here is a continuous density HMM-based Viterbi N-best decoder extended with some speed-up and pruning techniques for the purpose of being real-time [2]. The speed-up techniques include, for example, some constraints for the hypothesis extension procedure, thresholds for the stack size and for the maximum number of new hypotheses, and fast Gaussian computation tech-

niques. The adaptation module is based on the same program code as that used in the decoder, with the same adjustable constraints and parameters to guarantee the equivalence between the adaptation process and the continuous speech recognition process. Hence we can say that the program has a recognition mode, and also an adaptation mode. The difference between the recognition mode and the adaptation mode lies in the following points:

- When the adaptation mode is running, a huge amount of auxiliary data has to be stored for each active hypothesis in the N-best list whose data will be used to trace back the Viterbi path and to find the mixture and state indices of the HMMs belonging to the Viterbi path during the adaptation process.
- In order to make the search process efficient, some hypotheses are unified after each hypothesis extension. The unifying technique of the adaptation mode differs from that of the speech recognition mode in the sense that in the adaptation mode not all the hypotheses with the same phonetic history are unified, but just those whose trace-back data can be unified without information loss.
- Because of the large amount of stored data in the adaptation mode, the stack size should be reduced to keep the process real-time.

When the recognizer is in the adaptation mode, it will store the following data that will be used to trace back the Viterbi path and compute the new adapted HMMs. These data items are:

- the probability matrix for each state of each HMM,
- the mixture index matrix for each state of each HMM,
- the state matrix (with the series of phonetic labels, respectively) containing the state from which the given state was attained with maximal probability.

From these data items the Viterbi path, the phonetic segments and all the other data necessary for the computation of the adapted parameters can be easily obtained. In order to efficiently store all these data items for all the hypotheses, these data items are kept in a tree structure. If all hypotheses share a common root, then, up to the end of this common root, only one matrix from each type is stored, so the algorithm has linear storage requirements.

The adaptation module can be used both for supervised adaptation and for unsupervised adaptation. Supervised adaptation means that the uttered word series are known. The possible phonetic transcript variants of the word series are defined by a grammar containing rules which only permit the type of phoneme series that could be the phonetic transcription of the given word series. This simple grammatic model can also take into account assimilation rules and the possibility that there are silent gaps between the words. However, when the adaptation process is unsupervised, not a simple grammar, but a rather complex language model is used by the continuous speech recognizer to build up N-best lists of possible hypotheses.

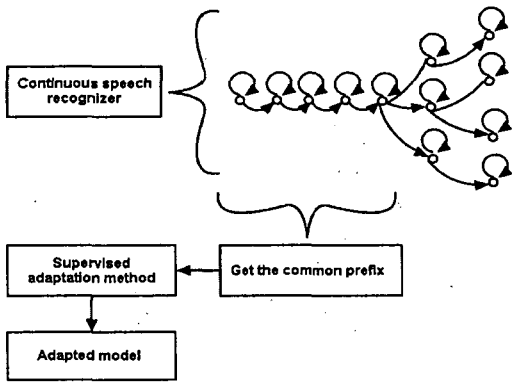


Figure 1: The process of unsupervised adaptation

After processing a few frames, the hypotheses in the N-best stack will have a common phonetic transcription prefix (they will have this, because N-best pruning is used, and the number of hypotheses grows exponentially); afterwards this common segment of the spoken data can be processed by applying the supervised method (see Figure 1).

3 Reducing the Amount of Faulty Adaptation Data

3.1 Extracting the Adaptation Data

As we mentioned earlier, the adaptation process could be more efficient if faulty adaptation data samples were removed from the well-segmented ones before the adaptation process was carried out. We will now discuss a new and simple Viterbi N-best cutting constraint that can be used to efficiently remove faulty adaptation data samples. Methods like this can be computed only with a highly limited stack size because of their high computational cost. So it cannot be normally used in the speech recognition process, but it can be used when adaptation data extraction is being performed, because the data samples necessary for this method are stored and are accessible.

It is a widely accepted property of HMMs that their transitional probability values do not model the duration of phoneme utterances very well. Moreover, the Δ and $\Delta\Delta$ feature components are strongly influenced by the phoneme duration (because they measure how fast the features change). When samples are taken from many different speakers, these features can be very different, so the resulting accuracy of the phonetic segmentation could be quite low. When the duration of the phoneme is modeled incorrectly, long phoneme durations can occur many times. In reality this is very unlikely, except in the case when the phoneme model has to model silent phases between two words. Our aim here is to reduce the number

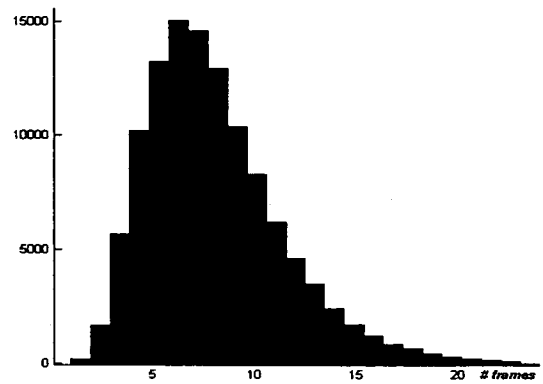


Figure 2: The histogram of the duration of all the phonemes except silence

of hypotheses containing an abnormally long phoneme utterance. Because the Viterbi path is stored for each hypothesis and for each state of the corresponding HMM models, the duration of the last phoneme can be computed using the Viterbi path corresponding to the HMM state having the maximal probability (for each particular state of a HMM there are different Viterbi paths). If the hypotheses with an unlikely duration are punished, then hypotheses with likely phoneme durations will be kept in the N-best list with a higher probability. With this in mind we define an a posteriori phoneme-weighting function by which the probability score of all hypotheses get multiplied:

$$\varphi_L = \begin{cases} 1 & ,\text{if } \mathrm{L} < \theta \\ e^{-\alpha(L-\theta)} & ,\text{if } \mathrm{L} \geq \theta \end{cases}$$

Here L means the duration of the phoneme computed from the Viterbi path. For the phoneme duration threshold (θ) we used a value of 15, and for the punishment constant (α) we used a value of -5. The duration threshold value was obtained from the histogram above (see Figure 2), this histogram was computed on the training database described later in the Section 4.

3.2 Confidence Measures for Adaptation

Many heuristic methods have been proposed in the literature for separating faulty adaptation data from correctly segmented ones. Another family of methods performs a weighting of the learning data samples when the adapted model is computed, here the weights are based on some particular confidence values. In this section we propose some new confidence-measure-like heuristics to reduce the number of faulty adaptation data samples.

Our first confidence measure is based on the observation that many of the incorrect segment boundaries are wrongly positioned by just a small amount. This

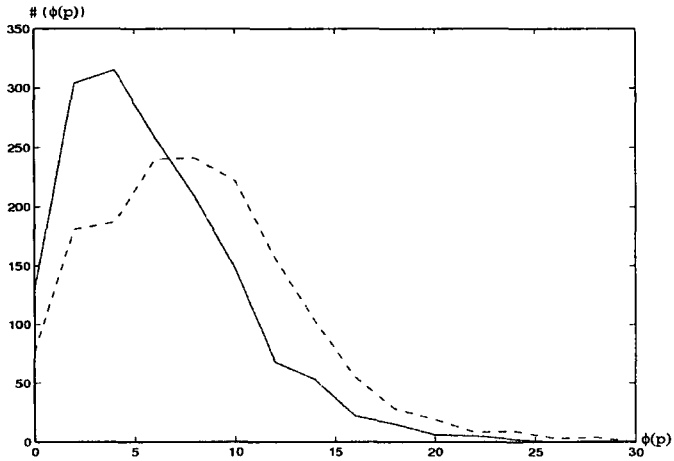


Figure 3: The histogram of the first confidence-measure scores on well-segmented samples (solid line), and on incorrectly segmented samples (dashed line)

means that the faulty segment parts at the boundaries will generally give a lower probability for the first and last states of the HMM. The difference between the probability values at the segments boundaries and the probability of the intermediate part will be higher for the faulty data than for the correctly segmented data. Based on this idea, we devised a simple formulation to measure this difference:

$$\varphi(p) = \left| \frac{\log P(f_1|\Theta = p) + \log P(f_N|\Theta = p)}{2} - \frac{\sum_{i=2\ldots N-1} \log P(f_i|\Theta = p)}{N-2} \right|,$$

where p is the recognized phoneme, Θ represents the HMM model, and f_i is a feature vector for the recognizer. Here φ is the function ranking the test phonetic data, and a classification between the correct and faulty samples can be done using an acceptance threshold.

In order to determine the proper threshold value, we had run an algorithm on the training set which computed the distribution of the above-mentioned scores on correctly segmented data and on incorrectly segmented data. The two histograms are shown in Figure 3 above.

The second scoring method proposed by us is based on the Fisher score [13], [8]. The Fisher score of a probabilistic model (which fits a probabilistic distribution to the data, and uses the Bayes rule for classification) is the gradient of the log-likelihood of a feature series with respect to the model parameters. Put formally,

$$U_f = \nabla_\Theta \log(P(f|\Theta))$$

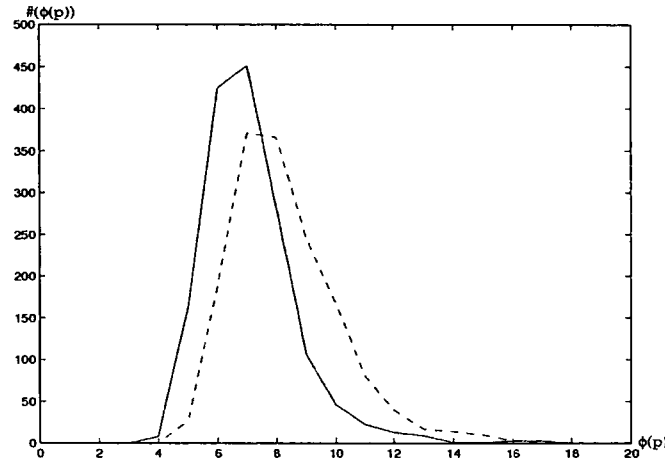


Figure 4: The histogram of the second confidence-measure scores on well-segmented samples (solid line), and on incorrectly segmented samples (dashed line)

Here the parameter Θ represents the model, while the parameter f stands for the feature vector series to be modeled. The gradient vector here measures how much the log-likelihood of the feature data series changes when the model parameters are varied. When using a Gaussian distribution, the Fisher score components with respect to the means of the Gaussian mixtures can be computed via the following formula:

$$\nabla_{\mu_k} \log(P(f|\mu_k)) = \sum_{i=1}^{m} P(k|f_i)\Sigma_k^{-1}(f_i - \mu_k)$$

Here m stands for the number of feature vectors modeled by the k-th Gaussian distribution. The feature vector series are denoted by f, and their elements are denoted by f_i.

We used the above-defined Fisher score to construct a confidence measure which has the following form:

$$\varphi(p) = \frac{1}{N} \sum_{i=1}^{N} ||\nabla_{\mu_k} \log(P(f_i|\mu_k))||,$$

where N is the number of feature vectors, and k is the index of the mixture that corresponds to the Viterbi path. Figure 4 above shows the distribution of the scores for the correctly segmented data samples and for the incorrectly segmented data samples.

4 Experiments and Results

The adaptation technique used here was the MAP (maximum a posteriori) method, which can be used for incremental adaptation by applying the following recursive formula [19]:

$$\mu_{d,N+1} = \frac{x_{N+1} + (N+\alpha) \cdot \mu_{d,N}}{N+1+\alpha}.$$

Here the parameter N represents the number of adapting examples for the given mixture component, while the parameter α controls how fast the mean (μ_d) is altered by this linear regression procedure.

For testing purposes we used the following settings (only the main parameters are given here):

- The continuous speech recognizer had a stack size value of 1000. The maximum number of new hypotheses for the phonetic hypothesis extension was set to 250, and the log-likelihood cut-off parameter was set to 260.
- The phoneme HMM models were monophone HMMs with 3 states, each state having a mixtures of 3 Gaussian distributions.
- The stack size of the adaptation method was 30, and the maximum number of new hypotheses in adaptation mode was restricted to 20. The log-likelihood cut-off parameter was set to 260.
- The recognition system used the conventional mel-frequency cepstral coefficient (MFCC) features. More precisely, 13 coefficients were extracted from each 25 msec frame, along with their Δ and $\Delta\Delta$ values, at a frame rate of 100 frames/sec.
- The α value of the MAP adaptation formula was set to 0.3. Other values were also tested in the interval $(0, 1)$, but no significant difference was observed in the results.
- The threshold for the first confidence-measure method had a value of 5, and the threshold value used by the second method was 8. These settings were selected to reduce a relatively high amount of faulty adaptation data (see Figures 3 and 4).

In the experiments our own training, test and speaker adaptation databases were employed. These databases and the continuous speech recognition system were created by the Research Group on Artificial Intelligence, the University of Szeged and the Laboratory of Speech Acoustics of the Budapest University of Technology and Economics within the framework of the Hungarian Medical Dictation Project financially supported by the national fund IKTA-056/2003. The adaptation database contained spoken sentences of 2 male(denoted by L and T) and 2 female (denoted by A and B) speakers, each speaker uttered 17 paragraphs from the same text.

Database	A	B	L	T	Average
Normal Adaptation	93.32%	92.28%	97.52%	93.29%	94.10%
Using PAPL	94.48%	93.48%	98.10%	94.10%	95.04%
WER reduction	17.36%	15.54%	23.38%	12.07%	17.08%

Table 1: Relative word error rate reductions achieved when using the phoneme a posteriori likelihood (PAPL) method on the adaptation data sets of four speakers (A, B, L, T).

Database	A	B	L	T	Average
Base Adaptation	94.48%	93.48%	98.10%	94.10%	94.97%
1st method	94.92%	93.16%	98.10%	94.46%	95.16%
2nd method	95.77%	93.16%	98.33%	94.32%	95.41%
WER reduction 1.	7.9%	-1.32%	0%	6.10%	3.9%
WER reduction 2.	23.36%	-1.32%	13.68%	3.7%	8.8%

Table 2: Relative word error rate reductions achieved when using the proposed two confidence measures on the adaptation data sets of four speakers (A, B, L, T).

The average duration of this speech data was 6 minutes per speaker, and the total number of phoneme examples for the 44 monophone classes was 3500 on average. The test database contained the utterances of the same four speakers, the recordings of 20 medical reports being taken from each speaker. The HMMs were trained using the MRBA database that was created by the Research Group on Artificial Intelligence, the University of Szeged and the Laboratory of Speech Acoustics of the Budapest University of Technology and Economics within the framework of the Hungarian Medical Dictation Project. This database contains 85365 phoneme examples, 26 female speakers, and 74 male speakers. The grammar model built for testing purposes was a simple word 3-gram, with a dictionary containing some 500 words. This grammar was trained on a text corpus built from 2500 thyroid gland medical reports. The grammar model for the supervised adaptation contained 162 assimilation rules.

Table 1 shows the results of our experiments when using the a posteriori probability multiplier for punishing the hypotheses with extremely long phoneme durations in the N-best stack. The results show that there is a definite improvement in the efficiency of the adaptation when this kind of scoring technique is applied. The experimental results using our confidence-measure-based methods to select the adaptation data are listed in Table 2. As the reader can see, using theses methods, a relative word error rate (WER) reduction of 4-8% was achieved on average, but the WER reduction was not always positive. The reason for the instability of these methods might be due to a significant reduction in the amount of adaptation data.

5 Conclusions and Further Work

Our results show that the efficiency of an adaptation method can be improved in two ways suggested here: by increasing the robustness of the method which extracts the adaptation data so that the automatically segmented data of the uttered sentence should contain fewer incorrect segments, and by selecting and dropping probably faulty adaptation data samples after the segmentation process. Building upon these promising results, more advanced adaptation data filtering methods will be tried in the near future which apply Data Description (One-Class Classification) methods that have a better scoring mechanism for phonetic segments, and are also better able to separate faulty data samples from the good data samples.

References

[1] T. Anastasakos, S.V. Balakrishnan, *The Use of Confidence Measures in Unsupervised Adaptation of Speech Recognizers,* Proc. Int. Conf. on Spoken Language Processing, Vol. 6, pp. 2303–2306., Sydney, NSW, Australia, Dec. 1998.

[2] András Bánhalmi, Dnes Paczolay, Lszl Tth *An Empirical Study on the Performanc of a CSR System Respect to Various Hypothesis-Space Pruning Techniques,* V. MSZNY, pp. 56–68., Szeged, Hungary, 2007.

[3] Digalakis, V. Rtischev, D. and Neumeyer, L., *Speaker Adaptation Using Constrained Reestimation of Gaussian Mixtures,* IEEE Trans. on Speech Audio Processing, 3, pp. 357–366., 1995.

[4] V. Digalakis, *On-line Adaptation of Hidden Markov Models Using Incremental Estimation Algorithms,* Proc. Eurospeech '97, pp. 1859–1862., Rhodes, Greece, 1997.

[5] S. Furui, *Cepstral Analysis Technique for Automatic Speaker Verification,* J. Acoust. Soc. Amer., Vol. 55, pp. 1204–1312., June, 1974.

[6] J.-L. Gauvain and C.-H. Lee, *Maximum a Posteriori Estimation for Multivariate Gaussian Mixture Observations of Markov Chains,* IEEE Transactions on Speech and Audio Processing, 2(2):pp. 291–298., April 1994.

[7] S. Homma, K. Aikawa and S. Sagayama, *Improved Estimation of Supervision in Unsupervised Speaker Adaptation,* Proc. of ICASSP-97, Vol. II, pp. 1023–1026., 1997.

[8] Jaakkola, T., Diekhans, M. and Haussler, D., *Using the Fisher Kernel Method to Detect Remote Protein Homologies,* Proceedings of the Internation Conference on Intelligent Systems for Molecular Biology, pp. 149–158., Aug. 1999.

[9] B.-H. Juang, S.Katagiri, *Discriminative Learning for Minimum Error Classification,* IEEE Trans. on Signal Processing, Vol. 40, No. 12, pp. 3043–3054., 1992.

[10] L. Lee and R.C.Rose, *Speaker Normalisation Using Efficient Frequency Warping Procedures,* Proc. ICASSP96, pp. 353–356., Atlanta, GA, 1996.

[11] C. Leggetter, P. Woodland, *Maximum Likelihood Linear Regression for Speaker Adaptation of Continuous Density HMMs,* Computer Speech and Language, Vol. 9, pp. 171–185., 1995.

[12] T. Matsui and S. Furui, *N-Best-Based Instantaneous Speaker Adaptation Method for Speech Recognition,* Proc. of ICSLP-96, Vol. III, pp. 973–976., 1996.

[13] P.J.Moreno P.Ho and N.Vasconceles, *A Kullback-Leibler Divergence Based Kernel for SVM Classification in Multimedia Applications,* Advances in Neural Information Processing Systems 16. MIT Press, Cambridge, 2004.

[14] P. Nguyen, P. Gelin, J.C. Junqua, J.T. Chien, *N-Best Based Supervised and Unsupervised Adaptation for Native and Non-Native Speakers in Cars,* Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Vol. 1, pp. 173–176., Phoenix, AZ, USA, March 1999.

[15] P. Nguyen, L. Rigazio, R. Kuhn, JC. Junqua, C. Wellekens, *Self-adaptation using eigenvoices for large-vocabulary continuous speech recognition,* In Adaptation-2001, pp. 37–40, 2001.

[16] Y. Normandin, R. Cardin, R. De Mori, *High-Performance Connected Digit Recognition Using Maximum Mutual Information Estimation,* IEEE Trans. on Speech and Audio Processing, Vol. 2, pp. 299–311., April 1994.

[17] W. Reichl, G. Ruske, *Discriminative Training for Continuous Speech Recognition,* Proc. EUROSPEECH, Madrid, Spain, pp. 537–540., 1995.

[18] Sankar, A. and Lee, C.H., *A Maximum-Likelihood Approach to Stochastic Matching for Robust Speech Recognition,* IEEE Trans. on Speech and Audio Processing, 4 (3), pp. 190–202., 1996.

[19] E. Thelen, *Long Term On-Line Speaker Adaptation for Large Vocabulary Dictation,* IEEE ICSPL, 4: pp. 2139–2142., October 1996.

[20] F. Wallhoff, D. Willet, G. Rigoll, *Frame-Discriminative and Confidence-Driven Adaption for LVCSR,* Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Vol. 3, pp. 1835–1838., Istanbul, Turkey, June 2000.

[21] Vicsi, K., Kocsor, A., Teleki, Cs., Tth, L., *Hungarian Speech Database for Computer-using Environment in Offices* II. MSZNY, pp. 315–318., Szeged, Hungary, 2004.

[22] T. Zeppenfeld, M. Finke, K. Ries, M. Westphal, A. Waibel, *Recognition of Conversational Telephone Speech Using the Janus Speech Engine,* Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Vol. 3, pp. 1815–1818., Munich, April 1997.

Acta Cybernetica 18 (2008) 391–402.

Extracting Human Protein Information from MEDLINE Using a Full-Sentence Parser

Róbert Busa-Fekete* and András Kocsor*†

Abstract

Today, a fair number of systems are available for the task of processing biological data. The development of effective systems is of great importance since they can support both the research and the everyday work of biologists. It is well known that biological databases are large both in size and number, hence data processing technologies are required for the fast and effective management of the contents stored in databases like MEDLINE. A possible solution for content management is the application of natural language processing methods to help make this task easier.

With our approach we would like to learn more about the interactions of human genes using full-sentence parsing. Given a sentence, the syntactic parser assigns to it a syntactic structure, which consists of a set of labelled links connecting pairs of words. The parser also produces a constituent representation of a sentence (showing noun phrases, verb phrases, and so on). Here we show experimentally that using the syntactic information of each abstract, the biological interactions of genes can be predicted. Hence, it is worth developing the kind of information extraction (IE) system that can retrieve information about gene interactions just by using syntactic information contained in these text. Our IE system can handle certain types of gene interactions with the help of machine learning (ML) methodologies (Hidden Markov Models, Artificial Neural Networks, Decision Trees, Support Vector Machines). The experiments and practical usage show clearly that our system can provide a useful intuitive guide for biological researchers in their investigations and in the design of their experiments.

Keywords: feature extraction, human gene interaction, data mining, machine learning, MEDLINE

*Research Group on Artificial Intelligence of the Hungarian Academy of Sciences and University of Szeged H-6720 Szeged, Aradi vértanúk tere 1., Hungary, E-mail: {busarobi,kocsor}@inf.u-szeged.hu

†The author was supported by the János Bolyai fellowship of the Hungarian Academy of Sciences.

1 Introduction

The MEDLINE [1] database is today becoming the most comprehensive biomedical abstract repository among the life sciences literature. Due to its easy access and availability, it is one of the most widely-used sources of scientific data employing several information retrieval systems. The NLM (U.S. National Library of Medicine) maintained MEDLINE database contains over 13 million references from about 4900 journals dating from 1965 to the present, and it is updated weekly. Obviously a crucial task in bioinformatics text mining is to develop an automatic system that extracts information about genes and their interactions. That is why we decided to build an information extraction (IE) system which makes use of natural language processing (NLP) techniques.

In the human life sciences, researchers are mostly interested in the interactions of genes, so in this area of science it would be good if biologists had an IE system that could search for relationships among genes [2, 3, 4, 5]. An interaction means, for instance, the binding of genes, or the existence of a gene that can influence the function of another gene. This kind of IE system can be quite useful in the design of biological experiments and drugs. Hence here we will introduce a system that can extract information about gene interactions which occur in living human cells from the MEDLINE. Because we wanted to avoid the building of huge and costly databases, we used and processed only freely available datasets. The system introduced here relies on the part of speech tagged (POS) and full sentence parsed (FSP) parts of MEDLINE. The main aim of our system is twofold: (a) to explore the MEDLINE abstracts for a set of genes given by the user and (b) to gather information about the interactions of genes that are described in the text of an abstract provided by the user.

One of the cornerstones of our information extraction system is the recognition of gene names. We used a thesaurus containing more than 40,000 gene names and their 120,000 synonyms to annotate the gene names in the abstracts. The thesaurus we obtained was built up using two sources: UMLS SPECIALIST Lexicon[7] and the Agilent Technologies [8] database. With this lexicon the identification of gene names can be reliably carried out. Later we will show some results of the efficiency of gene name recognition.

To predict new gene interactions we first needed a part of MEDLINE that had been annotated manually. In particular we needed an annotation based on the interaction of gene pairs when they occurred in the text of the same abstract. The National Center for Biotechnology Information (NCBI) [6] has many databases about gene interactions that have been arranged taxonomically. Using these data sets we were able to get a subset of MEDLINE containing information about human gene interactions. In this way we could derive a training set suitable for machine learning methods. Actually, many features of a syntactic tree can be represented as a multidimensional vector (i.e. depth and frequencies of different labels), hence each pair of gene names can be represented as a vector. In addition, the database about the interactions allows us to find out whether a sample is positive or negative (i.e. whether it is about the interaction of genes occurring in the text.) Here we

tested many machine learning algorithms on the training set. The results shows that the Decision Tree model is the most suitable one for this purpose.

When designing our system we had to take into account the fact that MEDLINE is a rapidly growing system and that the data is stored in compressed XML file format. So we created a framework which could handle the abstracts and their updates in their raw form, and could incorporate them into our IE system.

Not so long ago there was a workshop devoted to genic interaction extraction using MEDLINE records. The task was quite similar to ours here, and many results were produced by the participants of the workshop. These results are freely available [21], and comparing our results we can say that our results are competitive with the state-of-the-art systems. Moreover, we deal with a much bigger part of the gene set.

The paper is organised as follows. In the next section we will discuss the Feature Extraction task and how we can combine the databases that are available. Then in Section 3 we provide a brief overview of the Machine Learning models we employed in experiments. Section 4 following gives a comprehensive study of the performance of our IE system. Lastly, in Section 5, we summarise our results and offer suggestions for future work.

2 Extracting Information from MEDLINE using Distinct Data Sets

2.1 Relationships of the Applied Databases

With the advent of microbiology the experiments produce such a huge amount of information that it has become necessary to organize them into databases. In the middle of the last century some biologists began to collect and organize the papers on the results of biological and chemical experiments. Later this collection, called MEDLINE, became the main information resource for the experts dealing with biology, pharmacology and the human life sciences. Nowadays with the advent of extensive genome projects MEDLINE is getting bigger and bigger, and it contains an indigestible amount of information about proteins, genes and so on. For fast searching among the 13 billion records each abstract has a unique ID. It is called its PMID, and it makes the identification of entries much easier.

In order to construct a working system in the first step we needed to isolate the part of MEDLINE that is connected with human genes. Hence we generated a comprehensive lexicon containing the names and synonyms of the genes. Because biological work and experiments have gone on in parallel without being synchronized many genes were discovered in different labs at the same time, and they were named in a different way. These things makes the recognition of gene names harder. To resolve this problem we used the most comprehensive lexicons of the big biological research centres and chemical labs. The two lexicons that we then merged are the following:

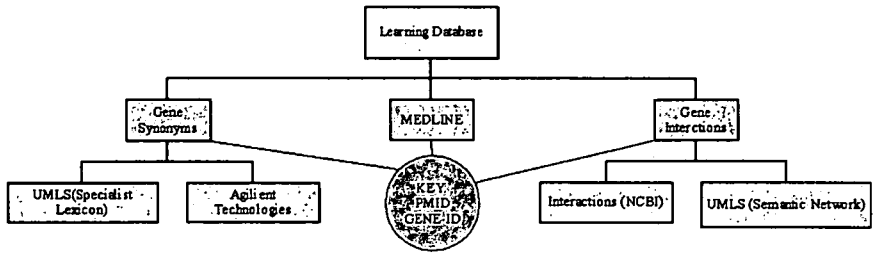


Figure 1: The relationships of the databases

- UMLS Specialist Lexicon
- Lexicon of Aglient Techonolgies

To identify a gene in two distinct lexicons we used a unique gene identification number. This ID is based on a consensus, and it is universal. In this way we obtained a synonym containing more than 40,000 gene IDs and about 120,000 synonyms. It allowed us to have a reliable gene name annotation.

Before we began extracting features we needed to have a part of MEDLINE that was not just about human genes, but about their interactions too. For this we used the interaction database of NCBI. It contains triplets, that is the gene ID of the interactants, and the PMID of the abstracts whose content is about their interactions. Using this we generated a subset of MEDLINE with 12,638 texts of abstracts, and then we annotated the resultant text by applying the synonyms lexicon mentioned above. We made use of this part of MEDLINE for the preparation of the training database.

Summarizing the above-mentioned points, in our studies we generated tree databases for the preparation of the training set: an extended synonym lexicon, a collection of interactions and a part of MEDLINE. The connections between them can be seen in Figure 2.1 later on. The unique PMID and gene ID were of course used as the primary key.

2.2 Relevant nouns and verbs

During our study we noticed that only a few nouns and verbs rarely occur between the pairs of gene names in the abstracts. One reason for this is that the texts were mostly written by researchers who were non-native speakers of English. Another was that a technical text always has a poor vocabulary. For instance in the following sentence the verb *interact* characterizes the given situation very well: *Here we show that BRCA1 interacts and colocalizes with topoisomerase IIalpha in S phase cells.*

In our studies we collected the nouns and verbs that frequently occur between the interactant genes and, based on their distributions we chose the relevant verbs and nouns. In our case a verb was deemed relevant if it occurred more than 1% in any case. These results eventually gave us 35 nouns and 15 verbs. We used this information as a discrete feature in the machine learning task as well.

2.3 Part of Speech Tagging and Syntactic Parsing

A morpho-syntactically analyzed text contains many possible parts of speech tags based on the word stem. The aim of the Part Of Speech (POS) Tagging is the selection of the appropriate POS Tags for each word according to its grammatical role in the sentence. The widespread approaches are based on machine learning techniques available today. But here we used the POS Tagger developed by the Human Language Technology Group of the University of Szeged. This tagger utilizes the internationally acknowledged MSD (Morpho-Syntactic Description) scheme [20] that is also used for encoding words. Due to the fact that the MSD encoding scheme is extremely detailed (one label can store morphological information on up to 17 positions), we did not exploit the granularity of this sort of annotation scheme. We only employed the following groups:

Adjective	Particle
Conjunction	Adverb
Determiner	Postposition
Interjection, sentence word	Article
Numeral	Verb
Noun	Other, unknown word
Special tokens	Abbreviation
Pronoun	

Syntactic parsing is the process of finding the immediate constituents of a sentence, that is a sequence of words. Syntactic parsing is an important part of the field of natural language processing and it is useful for supporting a number of large-scale applications including information extraction. Here we carried out the syntactic parsing of the texts of abstracts using the Link Grammar Parser [16]. The syntax trees of annotated sentences contain various types of phrases, as shown in the following list:

Noun phrase (NP)	Verb prefix (PREVERB)
Adjective phrase (ADJP)	Conjunction (C)
Adverb phrase (ADVP)	Pronoun phrase (PP)
Verb phrase (VP)	Clause (CP)
Infinitive(INF)	Sentence (S)
Negative (NEG)	

To build a learning dataset we collected different types of numerically encodable information describing each tag (part of speech and syntactic). These constituted the vector of attributes for the classification. After we made use of the number of

distinct POS tags that can be found between two given gene names. Here we also utilised the distinct number of syntactic tags that can be found on the only path between the pairs of the genes in the syntactic tree as features. Thus this gave us 35 features based on the sum of the number of POS tags and syntactic tags we used.
In summary the features we employed were the following:

- the number of words between two protein names
- part-of-speech code syntactic labels (for the two protein names themselves and for the words between them)
- the relevant nouns and verbs that occur in a sentence

3 The learning models

To solve classification problems effectively it is worth applying various types of classification methods. The features we used are discrete. Therefore we applied the C4.5 decision tree model, which usually works well on discrete feature set. The SVM classifier was also applied using binarized kernel function because these functions can be more suitable for discrete features than the traditional ones such as Gaussian RBF kernel and polynomial kernel function. We compared these two models to the Hidden Markov Model and Artificial Neural Network which are widely used models. Now we will provide a brief overview of the learning models we applied to the problems.

3.1 C4.5

C4.5 [19] is based on the well-known ID3 tree learning algorithm. It is able to learn pre-defined discrete classes from labeled examples. The result of the learning process is an axis-parallel decision tree. This means that during the training, the sample space is divided into subspaces by hyperplanes that are parallel to every axis but one. In this way, we get many n-dimensional rectangular regions that are labeled with class labels and organized in a hierarchical way, which can then be encoded into the tree. Since C4.5 considers attribute vectors as points in an n-dimensional space, using continuous sample attributes naturally makes sense. For knowledge representation, C4.5 uses the "divide and conquer" technique, meaning that regions are split during learning whenever they are insufficiently homogeneous. Splitting is done by axis-parallel hyperplanes, and hence learning is very fast. One great advantage of the method is time complexity; in the worst case it is $O(dn^2)$, where d is the number of features and n is the number of samples. Based on this we ran the C4.5 algorithm numerous times to perform preliminary tests to decide whether the inclusion of additional features was beneficial to the model or not.

3.2 Hidden Markov Model (HMM) approach

The Hidden Markov Modelling (HMM) technology it is assumed that the observation vectors belonging to a given state are independent, which in turn implies that the corresponding likelihood values can be combined by multiplication. With this poind in mind we adopted this probabilistic approach to predict new interactions. Each class was represented by a HMM, and the decision rule was based on the maximum posteriori probability derived from the HMMs. When we just used morpho-syntactic information the observation was the POS tags between the pairs of gene names. We also tried out the HMM approach on syntactic information (i.e. on the syntactic tags between two gene names), where the input data was the set of syntactic tags on the only path between the interactants. Here we varied the number of states between 2 and 5, because this was found to be the best empirically.

3.3 Artificial Neural Networks (ANN)

Since it was realized that, under the right conditions, ANNs can model class posteriors [13], neural nets have become evermore popular in the Natural Language Processing field. ANNs are based on the parallel architecture of the brain. We can view it as a simple multiprocessor system with a large number of interconnections and interactions between the processing units that use scalar messages. However, describing the mathematical background of ANNs is beyond the scope of this article. Besides, we believe that they are already well known to those who are acquainted with pattern recognition. In the ANN experiments we utilised the most common feed-forward multilayer perceptron network with the backpropagation learning rule.

3.4 Support Vector Machines and Binarized Kernel Functions

Theoretical discoveries generally have their own very different, unique histories before they find any practical application. One such example is the "kernel-idea", which had appeared in several fields of mathematics and mathematical physics before it became a key notion in machine learning. The kernel idea can be applied in any case where the input of some algorithm consists of the pairwise dot (scalar) products of the elements of an n-dimensional dot product space. In this case, simply by a proper redefinition of the two-operand operation of the dot product, we can have an algorithm that will now be executed in a different dot product space, and is probably more suitable for solving the original problem. Of course, when replacing the operand, we have to satisfy certain criteria, as not every function is suitable for implicitly generating a dot product space. The family of Mercer Kernels is, however, a good choice, and is based on Mercer's theorem [18]. Here we turn to the well-known and widely used Support Vector Machines (SVMs) [14, 15], which is a kernel method that separates data points of different classes with the help of a hyperplane. This separating hyperplane produced has a margin of maximal size with a verified optimal generalisation capability. Another nice feature of margin

maximization is that the calculated result is independent of the distribution of the sample points. Perhaps the success and popularity of the method can be attributed to this property.

There are many kernel functions for us to use, and there are also many ways of deriving functions from the existing ones. From the functions available, the two most popular are:

$$\text{Polynomial kernel:} \quad k_1(\mathbf{x},\mathbf{y}) = \left(\mathbf{x}^\top \mathbf{y}\right)^d, \qquad d \in \mathbb{N}, \tag{1}$$

$$\text{Gaussian RBF kernel:} \quad k_2(\mathbf{x},\mathbf{y}) = \exp\left(-||\mathbf{x}-\mathbf{y}||^2/r\right), \qquad r \in \mathbb{R}_+ \tag{2}$$

$$\text{Cosine polynomial kernel} \quad k_3(\mathbf{x},\mathbf{y}) = \left(\frac{\mathbf{x}^T\mathbf{y}}{\|\mathbf{x}\|\,\|\mathbf{y}\|} + \sigma\right)^q, \qquad q \in \mathbb{N}, \sigma \in \mathbb{R}_+. \tag{3}$$

Our feature set consists of discrete features. Hence we will investigate a derivation technique for kernel functions. This technique will make the kernel functions more suitable for discrete features. The main idea behind it is to use the binarized form of the input vectors. In detail if we have a feature having a domain set $|D| = k$, then we can map D into a binary vector space. This bijection represents the elements of D as binary vectors with a fixed length of k. Each binary vector has precisely one non-zero element. Thus each coordinate in the binary representation corresponds to a value in the domain set. Let us denote the mapping by $\eta(\mathbf{x})$ that carries out this bijective mapping componentwise for the input space. Using this η mapping we can extend the well-known kernel functions:

$$k^b(\mathbf{x},\mathbf{y}) = k(\eta(\mathbf{x}),\eta(\mathbf{y})) \tag{4}$$

All of the well-known kernel functions have a binarized form, and the experiments clearly show that, using this kind of kernel on a discrete feature set, the SVM can achieve a higher classification performance.

4 Experiments

4.1 The performance of the learning method

We carried out our experiments using the dataset described in Section 2. We then obtained a learning database containing 22195 positive and 90656 negative samples. The evaluation method we used here was a 10-fold cross validation.

We tried out various learning methods that were outlined in Section 3. In tests we found that the C4.5 decision tree learning method slightly outperformed the other machine learning algorithms. Here the confidence factor was set to 0.33. The ANN method had one hidden layer of one and half times more the number of hidden units than input neurons, used sigmoid activation functions, and 50 training session had a 0.3 learning rate. The SVM method gave a better performance using the binarized form of the well-known kernel function (linear, polynomial and cosine kernels). The best results using the SVM approach was achieved with a binarized

NEGATIVE POSITIVE	PRECISION(%)	RECALL(%)	F-MEASURE(%)
SVM (linear)	61.02 59.56	60.54 60.05	60.78 59.8
SVM (cosine pol.)	63.29 60.29	58.67 64.84	60.89 62.49
SVM (binarized lin.)	63.0 60.56	59.86 63.67	61.39 62.08
SVM (binarized cos.)	67.3 64.11	62.92 68.42	65.04 66.19
ANN	72.07 68.58	70.33 70.39	71.19 69.47
C4.5	71.64 71.89	73.53 69.93	72.58 70.9
HMM(POS)	59.52 56.39	53.03 62.74	56.09 59.39

Table 1: The classification performance of the various algorithms. In each cell the upper and lower values correspond to the performance of the different machine learning models on the negative class and positive class, respectively.

cosine polynomial kernel of 3rd degree. The performance of the HMM was better when we used POS tags as observations. The number of states was always tested in the range $1 - 5$. The experiments revealed that the 3-state HMM best fit the problem, although ironically it gave the worst performance. As the reader will notice in Table 4.1 below, each cell contains the percentage accuracy for the positive and negative classes separately. The columns on the other hand list the precision, recall and F-measure for each method we tested.

4.2 Description of the system

The system can be accessed through a web-based user interface that has two types of queries. First, the user can request a query concerning gene names. In response, the system can provide information about the MEDLINE records that describes the given genes. In addition, the system can visualise the results using Multi Dimensional Scaling. Hence the users will be better able to understand the relationships of the genes in question. Second, the user can also provide a text of an abstract as an input for the system. The system will then collect the gene names that crop up in the text, and it will may discover a possible interaction pattern among the genes that are listed in the abstract. A schematic overview of this is given in Figure 2.

The usefulness of the system cannot be underestimated as it considerably facilitates biological and biomedical activities in two ways. First, it supports the comparability of research findings achieved in different countries. Experiences can

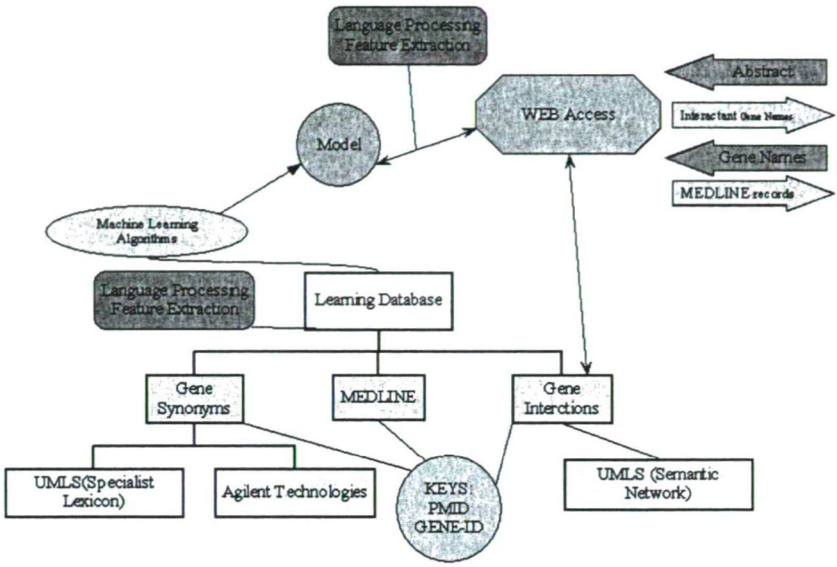


Figure 2: A schematic overview of the system

be accumulated in this way, and conclusions, deductions and extrapolations can be found more easily and in a shorter time. Second, it helps reduce research costs, since results can be obtained in a automated way and this also eliminates the need for many people to work on this time-consuming and laborious task. This way, researchers can focus on a variety of areas using the results produced by the system, and perhaps make new discoveries using our software tool. Below in Figure 3 is a screenshot of the system in operation.

5 Conclusions and Further Work

The information extraction technology presented here differs from existing methods in that it applies semantic-based natural language processing methods to biological content processing problems. As a novel aspect, it can visualise in a graph-like form the information about genes and their interactions that was retrieved, which can then be interactively browsed. This technology facilitates the work of researchers by providing structured, customisable and easily browsable information relating to their daily work. For this reason we think that it is certainly worthwhile developing our system further.

Next, we intend to improve our gene interaction recogniser using syntactic frame matching. This approach is a very commonly used technique in NLP, and we plan to define semantic frames. We hope that with these frames we will able to determine

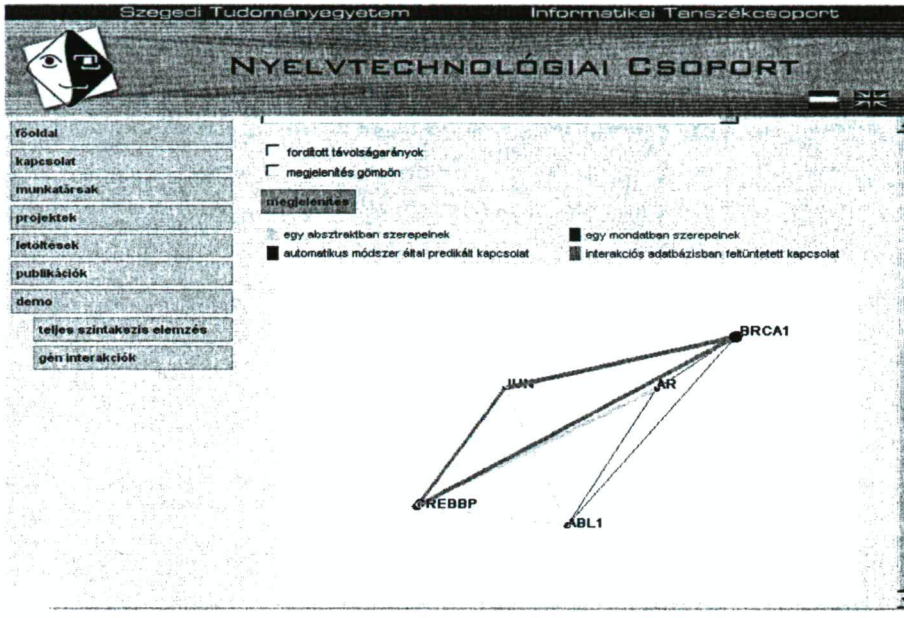


Figure 3: A screenshot of the system

the level of gene interactions as well. This would be a promising start in building a useful informatics tool for bio research and development.

References

[1] http://www.pubmedcentral.nih.gov

[2] Takeshi Sekimizu, Hyun S. Park, Jun'ichi *Identifying the Interaction between Genes and Gene Products Based on Frequently Seen Verbs in Medline Abstracts.* Genome Informatics 9:62-71, 1998.

[3] Marcotte EM, Xenarios I, Eisenberg D. *Mining literature for protein-protein interactions.* Bioinformatics. 2001 Apr;17(4):359-63.

[4] Ono T, Hishigaki H, Tanigami A, Takagi T. *Automated extraction of information on protein-protein interactions from the biological literature.* Bioinformatics. 2001 Feb;17(2):155-61.

[5] Donaldson I, Martin J, de Bruijn B, Wolting C, Lay V, Tuekam B, Zhang S, Baskin B, Bader GD, Michalickova K, Pawson T, Hogue CW. *PreBIND and Textomy–mining the biomedical literature for protein-protein interactions using a support vector machine.* BMC Bioinformatics. 2003 Mar 27;4:11. Epub 2003 Mar 27.

[6] http://www.ncbi.nlm.nih.gov

[7] http://www.nlm.nih.gov/research/umls

[8] http://www.home.agilent.com

[9] Daniel Sleator and Davy Temperley. *Parsing English with a Link Grammar.* Carnegie Mellon University Computer Science technical report CMU-CS-91-196, October 1991.

[10] John Lafferty, Daniel Sleator, and Davy Temperley. *Grammatical Trigrams: A Probabilistic Model of Link Grammar.* Proceedings of the AAAI Conference on Probabilistic Approaches to Natural Language, October, 1992.

[11] Dennis Grinberg, John Lafferty and Daniel Sleator. *A robust parsing algorithm for link grammars.* Carnegie Mellon University Computer Science technical report CMU-CS-95-125, and Proceedings of the Fourth International Workshop on Parsing Technologies, Prague, September, 1995.

[12] V. N. Vapnik. *Statistical Learning Theory.* John Wiley and Son, 1998.

[13] C.M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[14] N. Cristianini and J. Shawe-Taylor. *Support Vector Machines and other kernel-based learning methods.* Cambridge University Press, ISBN 0-521-78019-5, 2000.

[15] B. Schölkopf, C.J.C. Burges, and A.J. Smola. *Advances in Kernel Methods: Support Vector Learning.* MIT Press, Cambridge, MA, 1999.

[16] http://www.link.cs.cmu.edu/link/

[17] http://www.hprd.org

[18] J. Mercer. *Functions of positive and negative type and their connection with the theory of integral equations.* Philos. Trans. Roy. Soc. London, 415–446,1909.

[19] J. R. Quinlan. *C4.5: Programs for machine learning.* Morgan Kaufmann, 1993.

[20] Erjavec, T. and Monachini, M., ed. (1997) Specification and Notation for Lexicon Encoding, *Copernicus project 106 "MULTEXT-EAST"*, Work Package WP1 - Task 1.1 Deliverable D1.1F.

[21] http://genome.jouy.inra.fr/texte/LLLchallenge/

Acta Cybernetica 18 (2008) 403–425.

Programming Language Elements for Correctness Proofs*

Gergely Dévai†

Abstract

Formal methods are not used widely in industrial software development, because the overhead of formally proving program properties is generally not acceptable. In this paper we present an ongoing research project to make the construction of such proofs easier by embedding the proof system into a compiler.

Using the introduced new programming language, the programmer writes formal specification first. The specification is to be refined using stepwise refinement which results in a proof. The compiler checks this proof and generates the corresponding program in a traditional programming language. The resulting code automatically fulfills the requirements of the specification.

In this paper we present language elements to build specification statements and proofs. We give a short overview on the metaprogramming techniques of the language that support the programmer's work. Using a formal model we give the semantics of specification statements and refinements. We also prove the soundness of the basic algorithms of the compiler.

1 Introduction

1.1 Motivation

The study of formal methods to reason about program properties is getting a more and more important research area, as a considerable part of a software product's lifecycle is testing and bug-fixing. The theoretical basis — such as formal programming models and reasoning rules [16, 15, 20, 7, 17] — has been developed so far, but these are rarely used in industry [6]. The main reason for this fact is that formally proving a program property usually takes much more time than writing the program itself.

The goal of this research is to use programming language elements to make the construction of these proofs easier. The basic idea is to develop a new programming language where the source code contains the formal specification and

*This work is supported by "Stiftung Aktion Österreich-Ungarn (OMAA-ÖAU 66öu2)" and "ELTE IKKK (GVOP-3.2.2-2004-07-0005/3.0)".

†Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Budapest, E-mail: deva@elte.hu

the correctness proof of the implementation. The proofs are built up using *stepwise refinement* [23, 22], as this technique provides *correctness by construction*, and also helps the programmers to make the right decisions during software development. The compiler of the language has to check the soundness of the proof steps and to generate the program code in a target language using the information of the proof.

Similarly to programs, proofs also contain schematic fragments. These can be managed efficiently using *proof templates* that have the same role in proof construction as procedures have in traditional program development. This leads to a special kind of *metaprogramming* [9]: by the instantiation of the templates a proof is constructed (and checked) in compile time and from the proof a target language program is generated which automatically fulfills all the requirements stated in the specification.

1.2 Related work

We summarize existing solutions for formally verified software development and point out how the system presented in this paper differs from these approaches.

The most obvious solution is to embed the programming model in the framework of a theorem prover. Once the program is written, one can (automatically) generate its representation in the prover, formulate the desired properties (specification) and discharge them using the tools of the prover. Theorem provers like *Coq* [5] or *Isabelle/HOL* [24] can be used for this purpose. The problem is, that the program often does not fulfill the specification. In that case one has to start the whole procedure from the beginning by fixing the error in the program code and reconstructing parts of the proof. In contrast, our approach uses the *correctness by construction* principle: the programmer writes the specification first and refines it towards the implementation. Using this method one can discover design errors in an early stage of the development.

Another approach is to extend a programming language by annotations (*JML and ESC/Java2* [8], *SPARK* [4], *FPP* [25]) which express the specification and possibly the key elements of the proof. If the source code is extended by the specification statements only, we need additional tools to discharge the proof-obligations: the previously mentioned problems arise again. If the elements of the proof appearing in the code are detailed enough to enable automatic check of soundness, the source code becomes redundant: a complete proof of correctness usually contains all the necessary information needed to reconstruct the algorithm. This is exactly what is done in the system presented in this paper: the programmer writes specification and proof only, the instructions of the algorithm are "extracted" from the proof by the compiler automatically.

Functional and logic programming have a tight relation to formal methods as programs in these languages can be considered as *executable specifications*. For example in a functional language a program that sums the elements of a list reflects the "natural" definition of the problem very well. In contrast, it is not the case if we consider for example the problem of sorting a list. The "natural" way of specifying it, is to state that we seek a sorted permutation of the original sequence, while

(effective) implementations[1] in functional and logic programming languages are closer to the different sorting algorithms used also in imperative programming. This problem motivates several projects developing theorem provers especially for functional programming languages, like the *Lisp*-based *ACL2* [19], and *Sparkle* [10] for the *Clean* language. Theorems proved in these systems express relations between the functions implemented in the functional languages. This means that the construction of the proof takes place after the implementation of the functions. This has the same drawback mentioned before. Furthermore there are essentially "imperative" programming problems (consider for example the IO processes) that are hard to deal with in a purely functional environment. The proof of the soundness of these program fragments require further sophisticated methods [18]. In the language presented in this paper it is possible to specify the problem on an abstract level without any constraints to be "executable", while it also gives the possibility to fully control the effectiveness of the implementation. If we consider the example of sorting, the specification just states that a sorted permutation is needed, and this can be refined towards any of the effective sorting algorithms. It is even possible to choose assembly as the target language and to apply robust optimizations still keeping the program proved correct.

The most similar projects to the one presented in this paper are the *B-method* [3] and *Specware* [21], as both of them uses refinement. The *B-method* uses abstract machines specified by pre- and postconditions of its operations and invariants. The abstract machines are refined towards the implementation, and proof-obligations are generated to each refinement. Additional tools help to construct the proofs. *Specware* uses essentially the same schema, but it uses category theory as its basis. The goal of the project presented here is to keep the essence of these successful approaches while keeping the specification language and the refinement rules as simple and straightforward as possible, eliminating proof assistants and unifying the different levels of refinement.

The main characteristics and contribution of our approach can be summarized as follows:

- It is a refinement-based method to ensure correctness by construction.
- The resulting target language program is generated automatically, the programmer has to develop the specification and its refinements only.
- The compiler and the language is independent of the target language, because code generation is done by a separate module. The compiler's output is a set of state transitions. Any language which is able to implement these transitions can be a target language.
- The proofs are completed using the features of the new language only, no external tools are needed.

[1] It may be also possible to implement the sorting algorithm directly as a search for a sorted permutation (for example in Prolog), but the resulting algorithm is extremely ineffective.

- Reasoning in temporal and classical logic is unified.
- Proof strategies are not hard-wired in the compiler, but can be developed using the metaprogramming techniques provided by the new language.

1.3 Current state of the presented project

The programming language and the underlying system described in this paper is implemented in C++. The compiler currently consists of more than 6000 lines of source code. There are also hundreds of source files written in the new language to test the compiler, and several example programs implementing simple but useful algorithms [1, 13] are constructed for demonstration reasons. A small utility library is developed that contains templates to ease reasoning about loops, conditionals and can automatically construct proofs for expression evaluation. The currently supported target language is C++, but in a previous version the NASM assembly was supported.

As a relatively young project, the system's automated reasoning capabilities are not comparable yet with the power of the leading interactive theorem provers. However the advanced metaprogramming capabilities of the language are quite promising: it makes possible to reuse often used proof parts and to develop own proof strategies.

2 Examples

In this section we informally present the main features of the language using several small examples. The specification language and the refinement possibilities will be presented in more detail (2.1, 2.2), to help the reader to understand their formalization described in sections 3 and 4. On the metaprogramming features of this language and code generation issues we give a brief overview (2.3, 2.4).

2.1 Temporal properties

Expressions of the language are formulas of typed first order logic used to describe states of the program. The instruction pointer (*ip*) is considered as a normal variable and it may also appear in the formulas. For example the expression

```
ip = B & s = "Hello!"
```

states that the program execution is at the label B and the string "*Hello*!" is stored in the variable s. Using the $\gg$ symbol one can connect two such statements to build a *temporal progress property*:

```
ip = A >> ip = B & s = "Hello!";
```

This expresses that whenever the program execution is at label A, it has to reach after some (finite but not certainly bounded) steps label B and then $s =$ "*Hello*!" must hold.

We can express classical pre- and postconditions of Hoare logic [16] using the reserved labels *Start* and *Stop* instead of A and B in the example. Moreover, the explicit usage of the *ip* variable makes it possible to specify non-terminating programs. Like in the following example

```
ip = A >> ip = A & s = "Hello!";
```

where the "postcondition" implies the "precondition", stating that the program repeatedly returns to the same state or does not leave it.

This temporal property is close to the *leads to* property of *Unity* [7] and its relational version [17], but without supposing any kind of *fairness* of the scheduling. (As this language is currently designed for sequential languages, fairness is not a point.)

In many formal specification systems connections between pre- and postconditions are expressed using *auxiliary variables* (also called *parameter variables*). We also use this technique. For example the parameter variable j is used in the following property

```
ip = Start & i = j >> ip = Stop & i = j+1;
```

to state that this program increments the value of the variable i. Program variables and parameter variables are not distinguished syntactically but their declarations are different. As usually, parameter variables are not allowed to appear in the program, only in specification and proof.

It is also possible to express *safety properties* of the program: these are formulas enclosed between the '[' and ']' symbols. A safety property concerns one or more progress properties (the ones that are in the scope of it, for exact definition see section 3.4).

```
[ i > 0 ];
ip = A >> ip = B;
```

This example means that while the program proceeds from the label A to B, if $i > 0$ becomes true, it remains true at least until $ip = B$ is reached.

If a formula holds throughout a program fragment, in Hoare logic style proofs one has to repeat it in all intermediate steps. In our system one can use a safety property instead.

The *always* operator of temporal logic [20] and *invariant* notion of [17] are too strong, stating that a property must hold during the overall program. Our safety property is closer to the *(weak) unless* operator and the *unless* property of *Unity* and [17]. The difference is that our property must hold "between" a pre- and a postcondition.

2.2 Refining the specification

Progress properties of specifications can be refined by two constructs: *sequence* and *case analysis*. Safety properties are not to be refined, they are checked by the compiler automatically.

A *sequence* breaks a progress property into consecutive steps. In the following example the first line is the property to be refined and the two refining statements are enclosed by the braces.

```
ip = Start >> ip = Stop & s = "Hello!"
{
  ip = Start >> ip = A & s = "Hello!" { ... }
  ip = A & s = "Hello!" >> ip = Stop & s = "Hello!" { ... }
}
```

This refinement states that the original progress property is fulfilled by the program such that it first sets the desired value of the variable s while it steps to the label A, and then it terminates.

Note, that this example contains lots of unnecessary details. The algorithm actually used by the the compiler to check its soundness (see section 4.1) enables us to omit most of this redundancy. Furthermore such a simple refinement can be automatically constructed by proof tactics implemented in the language, so that the only line of the specification would be enough.

A *case analysis* can be used to implement conditionals. In the following example we want to compute the factorial of i. The program first computes the condition $i = 0$ and commits a conditional jump. The precondition of the following example describes the state of the program after this jump: it is at label A if $i = 0$, while it is at the label B if $i \neq 0$. The result is computed differently in each of the two cases, that is why we use case analysis. This is denoted by the *select* keyword.

```
(ip = A & i = 0) | (ip = B & !(i = 0))
>> ip = C & f = fact(i)
select
{
  ip = A & i = 0 >> ip = C & f = fact(i) { ... }
  ip = B & !(i = 0) >> ip = C & f = fact(i) { ... }
}
```

The soundness of such a refinement is checked by the algorithm presented in section 4.2.

These two refinement constructs are very close to the *Hoare logic* rules [16] for program sequences and *if* statements. Similar rules are the *transitivity* and *disjunctivity* of the *leads to* operator of *Unity* [7, 17].

2.3 Template features

The basic idea of this language is that the programmer builds specification and proof using the previously presented properties and refinement constructs, then

the compiler checks their soundness and generates the corresponding program in a target language. The programmer's work is supported by a metaprogramming layer of the language consisting of *templates* and *compile-time conditions.*

Templates are often used or valuable proof parts which are parametrized. These templates can be called by the programmer with arguments to obtain a concrete proof fragment.

Template arguments can be examined by *compile-time conditions.* Depending on these conditions a template call may result in different proof fragments. For example we have constructed a template to generate proof for expression evaluation. The expression to be computed by the program is an argument of the template. Compile time conditions examine whether this expression is a constant, a variable or a function application etc. Depending on the kind of the expression, a proof of an assignment instruction or a proof of a function call is produced by the template.

Formally defining the semantics of templates is not in the scope of this paper. We give a brief overview of template substitutions. If a template is called, the arguments are type checked first, then every occurrence of the formal arguments in the template definition is replaced by the corresponding actual ones. Compile-time conditions are evaluated next. Proof parts with false conditions are left out, and the template call is replaced by the remaining parts.

Templates and compile-time conditions are similar to the *macro features* of *macro assemblers* like MASM [2]. However, our templates are type checked. Similarly to macro assemblers, our metaprogramming constructs can also be used to simulate higher level programming constructs like control structures, procedures, exceptions etc. While macros of a macro assembler generate assembly instructions implementing the constructs, our templates generate their proofs. In order to achieve this goal we developed several kinds of templates. In the following we give a brief overview of them.

2.3.1 Temporal and classical axioms

Templates marked with the *axiom* or *atom* keywords contain classical or temporal axioms respectively. The programmer is able to declare functions and predicates to use in specifications and proofs and can state their mathematical properties in axiom templates. Atom templates contain temporal properties of instructions of the target language, like an assignment or procedure call.

2.3.2 Tactics

The *tactic* keyword introduces a template that can be called by the compiler automatically. These templates have to have exactly two *boolean* arguments. If the compiler finds a non-refined progress property (which is not in an axiom or atom) it calls the available tactics with the pre- and postcondition of the progress property as arguments. If none of the tactics provide a valid refinement for the property, an error-message is generated.

2.3.3 Static templates

If the programmer marks a template by the *static* keyword, the compiler checks the soundness of its refinements regardless of its arguments. The soundness of these refinements and the set of program instructions generated from them are not allowed to depend on the actual arguments of the template. If this is violated, the compiler generates an error message.

As a result, when a static template is called, there is no need to check its contents again. This makes it possible to implement induction with static templates. Proofs of loops and procedures are usually placed inside static templates, as induction is often needed to prove their soundness.

2.3.4 Passing proof fragments as arguments

Templates usually get expressions as arguments, but it turned out to be quite useful to pass complete blocks of refinements too. Using this possibility we were able to define templates that generate proofs for *if-statements* and for different kinds of *loops*. The following example is a sketch of computing the absolute value of i. We call the *if* template, which gets two "simple" arguments: the condition of the branch, and the postcondition that is to be established. It also has two "special" arguments: the proofs of the *if-* and *else-branch*.

```
if( i < 0, j = abs(i) )
{
  // proof of if-branch
}
{
  // proof of else-branch
}
```

2.3.5 Templates declared in templates

It is possible to declare templates inside other templates. For example we were able to write a template that can be used to declare procedures: when it is called, it generates two other templates, one static template with the proof of the procedure itself, and another template with the proof of the procedure call.

2.4 Code generation

When the compiler checks the refinements and finds a temporal progress property axiom, it saves the corresponding atom template call to a set. This set of template calls is the compiler's output. A separate code generation module converts it to the syntax of the target language.

Most template calls in the set contain the label of the corresponding instruction and the label of the following instruction. Template calls coresponding to instructions like *goto* and *return* contain their own label only, because these instructions do not pass the control flow to the instruction after them.

That is, these labels define a *partial order* on the set of template calls. The code generator sorts the instructions according to this order and generates the target language code.

3 Semantics of the specification language

In this section we present the model that is the semantic domain of the specification statements of the language. We use this model to prove certain properties of the temporal statements. In section 4 these properties will be used to prove the correctness of the algorithms used by the compiler to check refinements of the language.

3.1 Expressions and logic

Expressions in this language are typed first order logic formulas. The free variables of the formulas are program variables and parameter variables. These variables define a state space that the formulas are interpreted on. The programming model introduced in section 3.2 uses this state space to describe the behavior of programs.

The detailed presentation of the syntax and semantics of the expressions of this language can be found in [11].

3.2 Underlying programming model

The semantics of the safety and progress properties is given using a relational programming model, that we present in this section. The rules that the stepwise refinement is based on are also proved in this model.

3.2.1 State space, program

Let A be an arbitrary set, the *state space*. A *program* over A is a set of *state transitions*:

$$S \subseteq A \times A$$

In case of $(a, b) \in S$, the program S can change its state form a to b.

In this model the *instruction pointer* of a program is a component of the state space, just as all other variables. For example the program

```
K:  b = true;
L:  b = false;
M:
```

operates on a two-component state space, $A = \{K, L, M\} \times \{true, false\}$ and has two variables, *ip* and *b* respectively. It has four state transitions:

$$S = \{((K, false), (L, true)), ((K, true), (L, true)),$$
$$((L, false), (M, false)), ((L, true), (M, false))\}.$$

3.2.2 Operation of programs

The operation of a program can be described by the state sequences that the program follows during its execution. We use the notation A^{**} for the set of all (finite or infinite) nonempty sequences over the set A. The *operation* of program S over state space A is the following subset of A^{**}[2]:

$$r_S = \{\alpha \in A^{**} \mid \forall i \in [1..|\alpha| - 1] : (\alpha_i, \alpha_{i+1}) \in S \wedge (|\alpha| < +\infty \rightarrow \alpha_{|\alpha|} \notin \mathcal{D}_S)\}$$

This definition states that the program changes its state according to its transitions and it stops iff there is no applicable state-transition. Note that each α' postfix of a sequence $\alpha \in r_S$ is in r_S too.

For example, the sequences

$$< (K, false), (L, true), (M, false) >$$

and

$$< (L, true), (M, false) >$$

are valid for the example program presented in 3.2.1, but the sequences

$$< (L, false), (M, true) >$$

and

$$< (K, false), (L, true) >$$

are not.

The notation $F(Q, \alpha)$ is used for the *first occurrence* of an element in the sequence $\alpha \in A^{**}$ for which the statement Q holds.

$$F(Q, \alpha) = \begin{cases} i \in \mathcal{D}_\alpha & \text{if } Q(\alpha_i) \wedge \forall j \in [1..i-1] : \neg Q(\alpha_j) \\ +\infty & \text{if } \forall j \in \mathcal{D}_\alpha : \neg Q(\alpha_j) \end{cases}$$

This notation will be used to define the temporal properties of programs.

3.2.3 Temporal properties of programs

Let S be a program and P, Q and K be statements over the state space A. S *leads to* Q from P ($P \gg_S Q$), iff

$$\forall \alpha \in r_S : (P(\alpha_1) \rightarrow F(Q, \alpha) < +\infty).$$

That is, if the program is in a state for which the statement P holds it will reach some state where Q holds after a finite (but not certainly bounded) number of state transitions. The statement K is a *safety property* of S between P and Q ($[K]_S^{P,Q}$) iff

$$\forall \alpha \in r_S : (P(\alpha_1) \rightarrow \forall j \in [F(K, \alpha)..F(Q, \alpha)] \cap \mathcal{D}_\alpha : K(\alpha_j)).$$

[2] $\mathcal{D}_S = \{a \in A \mid \exists b \in A : (a, b) \in S\}$ is the domain of the relation S.

That is, if the program reaches some state where K holds while it proceeds form P to Q, then K remains true at least until Q is reached.

For example the properties

$$ip = K \gg_S ip = M$$

and

$$[b = true]_S^{(ip=K),(ip=L)}$$

hold for the example program in 3.2.1.

3.2.4 Temporal properties with parameters

Recall the example of section 2, where we used a parameter variable to express a progress property for each integer j:

```
ip = Start & i = j >> ip = Stop & i = j+1;
```

In general, let S be a program over state space A, and B be an arbitrary set, the *parameter space*, $C = A \times B$, and P, Q and K be statements over C. If $b \in B$, we use the notation P^b for the statement over A for which

$$\lceil P^b \rceil = \{a \in A \mid (a, b) \in \lceil P \rceil\}$$

holds. We say, that

$$P \gg_S Q \text{ and } [K]_S^{P,Q}$$

is true iff for every $b \in B$

$$P^b \gg_S Q^b \text{ and } [K^b]_S^{P^b,Q^b}$$

hold respectively.

3.2.5 Refinement rules

Using the relational model we introduce rules of the temporal properties. These rules are the basis for the algorithm that the compiler uses to check the refinement steps in the source code.

The proofs of these rules are not really difficult but rather technical. You can find them in the technical report [12]. Here we give short informal proofs and explanations.

In the following we suppose that S is an arbitrary program over the state space A, the parameter space is B, and $C = A \times B$. Furthermore we suppose that P, Q, R and K are arbitrary statements over C.

Rule of consequence

If $P \Rightarrow Q$ then $P \gg_S Q$ and $[K]_S^{P,Q}$.

To show this rule, we must take the sequences from r_S starting with an element satisfying P. But these elements also satisfy Q and using the definitions of the temporal properties we get what the rule states.

The condition of the rule states that each time the precondition holds, the postcondition also holds immediately. That is why any program can be used to reach the postcondition from the precondition. The same rule is present in the *Unity* based models [7, 17] for the *leads to* operator. In [16] Hoare had two such rules: one for the precondition and one for the postcondition. Both of those rules can be derived from our one and the *rule of sequence.*

Rule of sequence

If $P \gg_S Q$ and $Q \gg_S R$ then $P \gg_S R$.
If $[K]_S^{P,Q}$ and $[K]_S^{Q,R}$ then $[K]_S^{P,R}$.

To deal with the claim about the progress properties is quite simple: in each sequence starting with an element satisfying P, we can find an element for that Q holds, because of the first hypothesis. And then, by the second hypothesis we know that there is an element in the sequence for which R is true.

To prove the second statement we must explore cases depending on the order of the first occurrence of Q, R and K. In each case by using one or two of the hypothesis we can prove the statement.

This rule is essentially the rule of *Hoare logic* for program sequences and the transitivity of *leads to* in *Unity.*

Rule of case analysis

If $P \gg_S R$ and $Q \gg_S R$ then $P \vee Q \gg_S R$.
If $[K]_S^{P,R}$ and $[K]_S^{Q,R}$ then $[K]_S^{P \vee Q,R}$.

To prove this rule it is enough to consider, that if the first element of a sequence satisfies $P \vee Q$, then it satisfies either P or Q. In each case we can use the corresponding hypothesis to prove the claim. This reasoning can be applied for both statements.

This rule can be used to build proof for conditionals in a program. It splits the precondition into parts and allows the programmer to reach the postcondition in different ways from these parts. The corresponding rules are the disjunctivity of *leads to* and the *Hoare rule* for *if statements.*

Rule of safety property application

If $P \gg_S Q$ and $[K]_S^{P,Q}$ then $(P \wedge K) \gg_S (Q \wedge K)$.
If $[I]_S^{P,Q}$ and $[K]_S^{P,Q}$ then $[I]_S^{P \wedge K, Q \wedge K}$.

The core of both statements is that if K is a safety property between P and Q, then starting from $P \wedge K$, if we reach Q, then $K \wedge Q$ will hold. In the first statement we additionally suppose that Q is surely reached, which means $Q \wedge K$ is reached. Similar reasoning applies for the second statement.

In Hoare logic proofs all the unchanged parts of the assertions are present in every step of the proof. Using the rule described here we can "save" these unnecessary parts to safety properties and "put them back" into the progress properties when necessary. In *Unity* a similar rule describes the connection between the *leads to* operator and *invariants* of the program.

Rule of composition

Suppose that $S = S_1 \cup S_2$ and $\mathcal{D}_{S_1} \cap \mathcal{D}_{S_2} = \emptyset$.
If $P \gg_{S_1} Q$ then $P \gg_S Q$.
If $[K]_{S_1}^{P,Q}$ and $P \gg_{S_1} Q$ then $[K]_S^{P,Q}$.

Because the program S_1 reaches Q from P, every state on this way must be in the domain of S_1. Thus, by the crucial condition that the domains of the two composed programs are disjoint, these states can not be in the domain of S_2. From this we get that the compound program does exactly the same from P to Q as S_1 does. From this follows both claims of this rule.

Note that the disjointness can easily be fulfilled in case of sequential programs, but it is much harder for parallel/concurrent ones. Similar rules are established in *Unity*. In *Hoare logic*, this rule is implicitly present in each of its rules, as they are all compositional. The Hoare-style sequencing rule can be emulated in this model by first applying our *composition rule* for both programs and then applying our *sequencing rule*.

3.3 Syntax of proofs

In this paper we do not deal with the formal description of the operation of templates. After processing the meta programming elements in the code, the resulted proof consists of specification statements and their refinements. In this section we present the syntax of these elements.

In the grammar the following notations are used: non-terminal symbols are enclosed between the < and > symbols, alternatives are divided by the | symbol, the [and] symbols enclose optional parts, while [and]* denotes iteration (0, 1 or more times), terminals appear between single quotes.

```
<proof> ::=
  [ <safety property> | <temporal axiom> | <classical axiom>
```

```
  | <sequence> | <case analysis>
  | <conclusion refinement> ]*

<safety property> ::=
  '[' <expression> ']' ';'

<temporal axiom> ::=
  [ [ <condition> ':' ] <safety property> ';' ]*
  <expression> '>>' <expression> ';'

<classical axiom> ::=
  <expression> '=>' <expression> ';'

<sequence> ::=
  <expression> '>>' <expression> '{' <proof> '}'

<case analysis> ::=
  <expression> '>>' <expression> 'select'
  '{' [ <sequence> | <case analysis>
      | <conclusion refinement> ]* '}'

<conclusion refinement> ::=
  <expression> '=>' <expression> [ 'select' ]
  '{' [ <conclusion axiom> | <conclusion refinement> ]* '}'
```

That is, the proof is a sequence of safety and progress properties and conclusions. Each progress property and conclusion has to be refined, unless it is a progress property axiom or a conclusion axiom. These axioms are always produced by a template containing temporal or first order logic axioms. Conditions in safety property axioms are special expressions that can be computed in compile time.

3.4 Semantics of statements

Now we connect the statements of the language with the model presented in section 3.2. First, we define the state space, that the formulas are interpreted on. Let the $\mathcal{A} = \{v_1, ..., v_n\}$ be the set of program variables and $\mathcal{B} = \{p_1, ..., p_m\}$ be the set of parameter variables in the proof, and let V_i and P_j denote the sets of values corresponding to the types of v_i and p_j respectively. Then the formulas are interpreted on the space $(V_1 \times ... \times V_n) \times (P_1 \times ... \times P_m)$.

Let S denote the model of the specified program on state space $V_1 \times ... \times V_n$. A progress property $Q \gg R$ of the proof specifies that $Q \gg_S R$ has to be fulfilled by S.

In the grammar of section 3.3 the sequence of statements directly derived from the $< proof >$ symbol is called a block. The scope of a safety property consists of the statements from the location of the safety property to the end of the innermost

block that contains it. If the progress properties $Q_1 \gg R_1$, ..., $Q_n \gg R_n$ are in the scope of the safety property $[K]$, it specifies, that S fulfills $[K]_S^{Q_1,R_1}$, ..., $[K]_S^{Q_n,R_n}$.

A safety property axiom $c : [K]$; may contain expression variables. We say that $[K']$ is stated by the axiom if it is possible to assign expressions to the expression variables such that replacing them in K results in K', and the condition c is true for this assignment. If the temporal axiom consists of $c_1 : [K_1]$; ... $c_n : [K_n]$; $P \gg Q$; then the axiom specifies $P \gg_S Q$, and for each $[L]$ that is stated by one of the safety property axioms, $[L]_S^{P,Q}$ is specified too.

We say that a refinement is sound, if each program that fulfills the refining properties, also fulfills the refined properties. In the next section we present algorithms to check refinements, and prove that each refinement accepted by these algorithms is sound in the sense of the previous definition.

4 Algorithms to check refinements

In this section we present the algorithms of the compiler used to check the soundness of refinement steps. In the pseudo codes we use the following conventions. Parameters are always passed by value, that is, the procedures do not have side-effects, results are given by return values only. We use set variables with the usual set operations, and stacks with *push*, *pop* and *top* operations. In section 4.5 the function *sizeof* is also used to give the number of elements in a stack. If T is a progress property, we use the notations $pre(T)$ and $post(T)$ to denote the pre- and postconditions of T respectively.

In the algorithms the procedure $infer(P, Q)$ is called. This can be any algorithm that tries to infer the formula Q from P. The only requirement is, that it has to be sound, that is, if it returns true then $P \Rightarrow Q$ has to hold. Of course, this procedure can not be complete, because first order logic is not decidable.

An other algorithm, $GCNF(P)$ is also used in the algorithms. It transforms the formula P to a *generalized conjunctive normal form*. The exact form of this GCNF and the $infer$ algorithm currently used in the compiler are described in [11].

4.1 Processing sequential refinements

Algorithm: $process - sequent(Stm, \mathcal{K}, V)$
Parameters: Stm: the statement to process, $\mathcal{K}$: set of formulas, V: stack of formulas
Local variables: T: statement, P: formula
Return value: stack of formulas

1. $V := push(V, GCNF(pre(Stm)))$; $T :=$the first refining statement;

2. if T is the first statement of an axiom then call
   $V, T := process - axiom(T, \mathcal{K}, V)$;
   go to step 8;

3. if T is a safety property $[K]$ then $\mathcal{K} := \mathcal{K} \cup \{K\}$; go to step 8;
4. if $infer(top(V), pre(T))$ returns $false$ then return ERROR;
5. if T is a sequential refinement then call
   $V := process - sequent(T, \mathcal{K}, V)$;
   go to step 7;
6. if T is a refinement by case analysis then call
   $V := process - select(T, \mathcal{K}, V)$;
   go to step 7;
7. $P := top(V)$; $V := pop(V)$; $V := push(V, GCNF(P\&post(T)))$;
8. if T is the last refining statement in Stm,
   a) then go to step 9;
   b) else T :=the next statement of the refinement; go to step 2;
9. if $infer(top(V), post(Stm))$ returns $false$ then return ERROR;
10. return $pop(V)$;

4.2 Processing refinements by case analysis

Algorithm: $process - select(Stm, \mathcal{K}, V)$
Parameters: Stm: the statement to process, $\mathcal{K}$: set of formulas, V: stack of formulas
Local variables: T: statement, P: formula
Return value: stack of formulas

1. D :=empty disjunction; T :=the first refining statement;
2. $V := push(V, GCNF(pre(Stm)))$; $D := D \mid pre(T)$;
3. if T is a sequential refinement then call
   $V := process - sequent(T, \mathcal{K}, V)$;
   go to step 5;
4. if T is a refinement by case analysis then call
   $V := process - select(T, \mathcal{K}, V)$;
   go to step 5;
5. $P := top(V)$; $V := pop(V)$; $V := push(V, GCNF(P\&post(T)))$;
6. if $infer(top(V), post(Stm))$ returns $false$ then return ERROR;
7. $V := pop(V)$;
8. if T is the last refining statement in Stm,

a) then go to step 9;

b) else T :=the next statement of the refinement; go to step 2;

9. if $infer(pre(Stm), D)$ returns $false$ then return ERROR;

10. return V;

4.3 Processing axioms

Algorithm: $process - axiom(Stm, \mathcal{K}, V)$
Parameters: Stm: the first statement to process, $\mathcal{K}$: set of formulas, V: stack of formulas
Local variables: $\mathcal{M}$: set of statements, U: formula, W: stack of formulas
Return value: stack of formulas, statement

1. $\mathcal{M} := \emptyset$; W :=empty stack;

2. if Stm is a safety property axiom $[M]$

   a) then $\mathcal{M} := \mathcal{M} \cup \{M\}$; Stm :=the next statement; go to step 2;

   b) else go to step 3;

3. if $infer(top(V), pre(Stm))$ returns $false$ then return ERROR;

4. for each element $K \in \mathcal{K}$: if $\exists M \in \mathcal{M}$: $check - safety - property(K, M)$ returns $false$ then return ERROR;

5. for each element $F = F_1 \& ... \& F_n$ of V (from the bottom to the top):

   a) for each F_i $(i \in [1..n])$:
   if $\exists M \in \mathcal{M}$: $check - safety - property(F_i, M)$ returns $false$ then F :=remove F_i from F;

   b) $W := push(W, F)$;

6. $U := top(W)$; $W := pop(W)$; $W := push(W, GCNF(U \& post(Stm)))$;

7. return W, Stm;

4.4 Using safety property axioms

Algorithm: $check - safety - property(K, L)$
Parameters: K: formula, L: safety property axiom statement (of the form $c : [I]$)
Return value: boolean

1. Try to assign an expression to the expression variables in I such than K and I match. If it is not possible then return $false$;

2. Evaluate the condition c with the assigned expressions. If it is $true$ return $true$, else return $false$.

4.5 Soundness of the algorithms

In this section we present a theorem that states the soundness of the presented algorithms, and three lemmas that are used in the proof of the theorem. The proofs can be found in appendix **A.**

Theorem. If the refinements in a (finite) proof are accepted by the algorithms presented in sections 4.1–4.4, and a program fulfills all the axioms used in the proof, then the program fulfills all the temporal properties appearing in the proof.

Lemma 1. If a program S fulfills an axiom with properties $c : [I]$; and $P \gg Q$ and $check - invariant(K, c : [I])$ returns *true*, then $[K]_S^{P,Q}$ also holds.

Lemma 2. If the call $W, Stm' := process - axiom(Stm, \mathcal{K}, V)$ processes the axiom consisting of statements $c_1 : [I_1](= Stm)$, $c_2 : [I_2]$, ..., $c_n : [I_n]$, $P \gg Q$ without returning an error and the program S fulfills these properties, then

- $top(V) \gg_S top(W)$,
- $\forall K \in \mathcal{K} : [K]_S^{top(V),top(W)}$,
- $\forall i \in [1..sizeof(pop(V)) - 1]$ for the i^{th} elements F_i of $pop(V)$ and G_i of $pop(W)$: $F_i \Rightarrow G_i$ and $[G_i]_S^{top(V),top(W)}$ is true.

Lemma 3. If the call $W := process - sequent(Stm, \mathcal{K}, V)$ or $W := process - select(Stm, \mathcal{K}, V)$ accepts a refinement without error, and the program S fulfills all the properties inside the refinement, then the following hold:

- $pre(Stm) \gg_S post(Stm)$,
- $\forall K \in \mathcal{K} : [K]_S^{pre(Stm),post(Stm)}$,
- $\forall i \in [1..sizeof(V)]$ for the i^{th} elements F_i of V and G_i of W: $F_i \Rightarrow G_i$ and $[G_i]_S^{pre(Stm),post(Stm)}$.

5 Summary

The project presented in this paper experiments with two aspects of formal methods:

- embedding of a refinement based calculus into a compiler to produce code correct by construction,
- and using metaprogramming techniques to make proof construction easier.

In the current paper we discussed the first aspect. Semantics of specification statements were presented as well as the basic refinement-checking algorithms of the compiler together with their proofs of correctness.

Further research efforts have been issued to test the flexibility of our programming model and specification language. We embedded a model to reason of dynamic memory management and pointers [13], and also some datatypes of the C++ Standard Template Library and their basic operations with iterators were specified in this language [14]. These embeddings were possible without modifying the compiler and language design. Therefore we concluded that it is flexible and expressive enough.

In this paper we gave only a brief overview of the metaprogramming toolset of this language. Our current research concentrates on supporting the programmers' work by these tools. We investigate how to emulate higher level proof rules by templates.

There are also interesting research areas for later development of this work. It would be useful to extend the expressive power of the specification statements, for example to specify randomized algorithms, parallel programs, resource consumption of the program etc.

In its current state this system in already applicable to specify programming problems and to derive not-too-complicated algorithms as verified solutions. The limitation is clearly the non-sufficient automatic reasoning capabilities of the system. We experiment with the metaprogramming features of the language to overcome this limitation. While most formal methods use their built-in provers as black boxes, in our case most of the proof strategies are implemented not in the compiler but using the language itself. They are accessible and extendable.

A Proof of theorems of section 4.5

A.1 Proof of the theorem

We prove the theorem by structural induction on the structure of the proof tree.
In the base case we observe a refinement where all the refining statements are axioms. By the assumption of the theorem, the program S fulfills all these axioms. From this, by lemma 3 we get that S fulfills the refined properties too.
In the inductive case, by the induction assumption the program fulfills all the properties inside a refinement. From this, by lemma 3 we get that S fulfills the refined properties too.

A.2 Proof of lemma 1

If the call returned *true*, it means that it was possible to assign expressions to the expression variables such that K and I matched and the condition was also *true*. Using the semantics described in section 3.4 it means that K is stated by $c : [I]$, and $[K]_S^{P,Q}$.

A.3 Proof of lemma 2

Step 2 of the algorithm collects all the safety property axioms into the set $\mathcal{M}$. Because the algorithm processed the axioms without error, by step 4 we know that $\forall K \in \mathcal{K} : [K]_S^{P,Q}$.
Step 5 copies the elements of V to W in such a way that it removes certain parts of the conjunctive chains, thus $\forall i \in [1..sizeof(V)] : F_i \Rightarrow G_i$, where F_i and G_i are the i^{th} elements of V and W respectively. By lemma 1 we get that the removed parts are those that are not safety properties of the axiom. It means that $\forall G \in W : [G]_S^{P,Q}$. In the following steps the algorithm modifies only the top element of W, so at the end we have $\forall i \in [1..sizeof(pop(V))] : F_i \Rightarrow G_i$ and $[G_i]^{P,Q}$, where F_i and G_i are the i^{th} elements of $pop(V)$ and $pop(W)$ respectively.

Let us denote the value of $top(W)$ by T at the end of step 5. Thus, we also have $top(V) \Rightarrow T$ and $[T]_S^{P,Q}$. Because the call did not return an error, by step 3 we know that $top(V) \Rightarrow P$, and because of $top(V) \Rightarrow T$ also $top(V) \Rightarrow P\&T$. By the assumption of the lemma we know that $P \gg_S Q$ holds. Using the rule of safety property application we get that $P\&T \gg_S Q\&T$. In step 6 the algorithm changes $top(W)$ from T to $Q\&T$, that is we have $P\&T \gg_S top(W)$. From this and from $top(V) \Rightarrow P\&T$ by the rules of conclusion and sequence we get $top(V) \gg_S top(W)$. Using the safety property parts of the same rules give:
$\forall G \in pop(W) : [G]_S^{top(V),top(W)}$ and $\forall K \in \mathcal{K} : [K]_S^{top(V),top(W)}$.

A.4 Proof of lemma 3

We prove the lemma by structural induction on the structure of the proof tree.
The base case is a refinement where all the refining statements are axioms. From the syntax presented in section 3.3 follows that such a refinement is a sequential one. First, we prove, that each time when the algorithm $process-sequent(Stm', \mathcal{K}', V')$ is at step 2, then the following loop invariants hold: $pre(Stm') \gg_S top(V)$, $\forall K \in \mathcal{K}' : [K]_S^{pre(Stm'),top(V)}$ and $\forall i \in [1..sizeof(V')]$ for the i^{th} element $F_i \in V'$ and $G_i \in V$: $F_i \Rightarrow G_i$ and $[G_i]_S^{pre(Stm'),top(V)}$. When the algorithm is at step 2 for the first time $top(V) = pre(Stm')$, because of the initialization in step 1, so the loop invariants hold because of the rule of conclusion. By lemma 2 we get that step 2 preserves these invariants. As, according to our assumption, there are axioms in this refinement only, step 2 and 8 are repeated until we reach the end of the refinement. Then by step 9 we get that $top(V) \Rightarrow post(Stm')$. From this and the loop invariants, using the rules of conclusion and sequence we get the properties that we wanted to prove.
In the inductive case of the proof we have two cases: the cases of refinements by sequence and case analysis.
If the refinement is sequential, then the proof is similar to the base case including the loop invariant, but we have to deal with steps 3-7 too. In step 3 $\mathcal{K}$ is changed but in the loop invariant $\mathcal{K}'$ is present, so that is preserved. If the execution is at step 4 then we know that T is a progress property, and that $top(V) \Rightarrow pre(T)$. Then, depending on the type of the refinement of T step 5 or 6 is executed. We use

the induction assumption and a proof similar to the end of the proof A.3 to show that the loop invariant is preserved.
If the refinement is a case analysis, let V_i^1 and V_i^2 denote the value of the stack V at the end of step 2 and step 6 respectively at the i^{th} refining statement T_i. By the induction assumption at steps 3 and 4 we get that $pre(T_i) \gg_S post(T_i)$, $\forall K \in \mathcal{K} : [K]_S^{pre(T_i),post(T_i)}$ and $\forall i \in [1..sizeof(V_i^1)]$ for the j^{th}element $F_j \in V_i^1$ and $G_j \in V_i^2$: $F_j \Rightarrow G_j$ and $[G_j]_S^{pre(T_i),post(T_i)}$. Thus we have $pre(Stm) \Rightarrow top(V_i^2)$ and $[top(V_i^2)]_S^{pre(T_i),post(T_i)}$. From the latter one by the rule of safety property application we get $pre(T_i)\&top(V_i^2) \gg_S post(T_i)\&top(V_i^2)$, $\forall K \in \mathcal{K} : [K]_S^{pre(T_i)\&top(V_i^2),post(T_i)\&top(V_i^2)}$ and $\forall G \in V_i^2 : [G]_S^{pre(T_i)\&top(V_i^2),post(T_i)\&top(V_i^2)}$.
As step 5 changes $top(V)$ from $top(V_i^2)$ to $post(T_i)\&top(V_i^2)$ by step 6 we get that $post(T_i)\&top(V_i^2) \Rightarrow post(Stm)$. From this by the rules of consequence and sequence we have $pre(T_i)\&top(V_i^2) \gg_S post(Stm)$, $\forall K \in \mathcal{K} : [K]_S^{pre(T_i)\&top(V_i^2),post(Stm)}$ and $\forall G \in V_i^2 : [G]_S^{pre(T_i)\&top(V_i^2),post(Stm)}$, which is true for each statement T_i in the refinement. Using the rule of disjunction $n-1$ times, we get $(pre(T_1)\&top(V_1^2)) \mid \ldots \mid (pre(T_n)\&top(V_n^2)) \gg_S post(Stm)$ and the similar safety properties.
Because of step 2 we know that at step 9 $D = pre(T_1) \mid \ldots \mid pre(T_n)$, and step 9 checks that $pre(Stm) \Rightarrow D$. From this, by $pre(Stm) \Rightarrow top(V_i^2)$ we get that $pre(Stm) \Rightarrow (pre(T_1)\&top(V_1^2)) \mid \ldots \mid (pre(T_n)\&top(V_n^2))$ is also true. Using the rule of consequence and the rule of sequence for this and for the previous result we get the properties of the lemma.

References

[1] Home of LaCert: http://deva.web.elte.hu/LaCert.

[2] Home of MASM: http://masm32.com/.

[3] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[4] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison Wesley, 2003.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[6] J. P. Bowen and M. G. Hinchey. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 8–16, New York, NY, USA, 2005. ACM Press.

[7] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, 1988.

[8] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362/2005, pages 108–128. Springer, 2005.

[9] M. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[10] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers, Sparkle: A functional theorem prover. *LNCS*, page 55, 2001.

[11] G. Dévai. Programming language elements for proof construction. In *Volume of abstracts of the 6th Joint Conference on Mathematics and Computer Science*, 2006.

[12] G. Dévai. Refinement rules of LaCert. Technical report, Dept. of Programming Languages and Compilers, Fac. of Informatics, ELTE University, 2007.

[13] G. Dévai and Z. Csörnyei. Separation logic style reasoning in a refinement based language. In *Proceedings of the 7th International Conference on Applied Informatics (to appeare)*, 2007.

[14] G. Dévai and N. Pataki. Towards verified usage of the C++ Standard Template Library. In *Proceedings of the 10th Symposium on Programming Languages and Software Tools (to appeare)*, 2007.

[15] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[17] Z. Horváth. *A Relational Model of Parallel Programs (in Hungarian).* PhD thesis, Phd School in Informatics, Eötvös Loránd University, Budapest, Hungary, 1996.

[18] Z. Horváth, T. Kozsik, and M. Tejfel. Extending the Sparkle core language with object abstraction. *Acta Cybernetica*, 17:419–445, 2005.

[19] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[20] F. Kröger. *Temporal Logic of Programs.* Springer, Berlin, Heidelberg, 1987.

[21] J. McDonald and J. Anton. Specware - producing software correct by construction, 2001.

[22] C. Morgan. *Programming from specifications.* Prentice Hall International (UK) Ltd., second edition, 1994.

[23] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9(3):287–306, 1987.

[24] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[25] J. Winkler. The frege program prover FPP. *Internationales Wissenschaftliches Kolloquium*, 42:116–121, 1997.

Acta Cybernetica 18 (2008) 427–450.

Pebble Alternating Tree-Walking Automata and Their Recognizing Power*

Loránd Muzamel†

Abstract

Pebble tree-walking automata with alternation were first investigated by Milo, Suciu and Vianu (2003), who showed that tree languages recognized by these devices are exactly the regular tree languages. We strengthen this by proving the same result for pebble automata with "strong pebble handling" which means that pebbles can be lifted independently of the position of the reading head and without moving the reading head. Then we make a comparison among some restricted versions of these automata. We will show that the deterministic and non-looping pebble alternating tree-walking automata are strictly less powerful than their nondeterministic counterparts, i.e., they do not recognize all the regular tree languages. Moreover, there is a proper hierarchy of recognizing capacity of deterministic and non-looping n-pebble alternating tree-walking automata with respect to the number of pebbles, i.e., for each $n \geq 0$, deterministic and non-looping $(n+1)$-pebble alternating tree-walking automata are more powerful than their n-pebble counterparts.

1 Introduction

The concept of a *tree-walking automaton (twa)* was introduced in [1] for modeling the syntax-directed translation from strings to strings. A twa A, obeying its state-behaviour, walks on the edges of the input tree s and accepts s if the (only) accepting state q_{yes} is accessed. Every tree language recognized by a twa is regular. It was an open problem for more than 30 years whether twa can be determinized or whether twa can recognize all regular tree languages. The answer for these two questions were provided in [4] and [3] saying that (1) twa cannot be determinized and (2) twa do not recognize all regular tree languages. Hence $dTWA \subset TWA \subset REG$, where $dTWA$ and TWA denote the tree language classes recognized by deterministic twa and twa, respectively, and REG is the class of regular tree languages.

*Research of the author was partially supported by German Research Foundation (DFG) under GrantGK 334/3 during his stay in the period February-April 2005 at TU Dresden, and also was supported by the Hungarian Scientific Foundation (OTKA) under Grant T 030084.

†Department of Computer Science, University of Szeged, Árpád tér 2., H-6720 Szeged, Hungary, E-mail: muzamel@inf.u-szeged.hu

The generalization of twa with nested pebbles came recently, by two independent motivations: On the one hand, with the advancement of XML theory, finite state recognizers (with name *n-pebble tree automata*) were used in [21] to show that the XML typechecking problem is decidable. On the other hand, the concept of *n-pebble tree-walking automata (n-ptwa)* were defined in [9] to recognize first-order logic on trees. Later, in [10] n-ptwa were extended with a more general pebble handling. In the present paper we will consider tree recognizers along the line of [10].

An n-ptwa A is equipped with a pointer (or reading head), and n different pebbles, which are denoted by $1, \ldots, n$. The pointer of A walks on the edges of an input tree s, while the pebbles can be dropped at and lifted from a node of s in a *stack-like fashion* which means the following:

Dropping of pebbles: If there are $l < n$ pebbles on s, then pebble $l + 1$ can be dropped at the node pointed by the pointer.

Lifting of pebbles: There are two different approaches.

> *weak pebble handling:* If there are $l > 0$ pebbles on s, then pebble l can be lifted iff it is placed at the node pointed by the pointer.
>
> *strong pebble handling:* If there are $l > 0$ pebbles on s, then pebble l can be lifted independently of the position of the pointer.

The automaton A computes on s as follows. Initially, A is in the initial state q_0, its pointer points to the root of s, and no pebbles are placed on s. Then – applying its rules – A moves along the edges of the input tree, drops, and lifts pebbles in a stack-like fashion (with strong or weak pebble handling, depending on the definition). Each step depends on (1) the current state, (2) the presence of the pebbles on the input tree, and (3) the position of the pointer. *A accepts s*, if the (only) accepting state q_{yes} is accessed. Otherwise, *A rejects s*. We say that L is the tree language *recognized by A*, if L contains exactly the trees accepted by A.

Originally, the n-ptwa was defined in [9] with weak pebble handling. In the the present paper we are interested in the more general strong pebble handling, which was used in [10, 22, 5].

In [10] it was proved that tree languages recognized by ptwa are regular. In [5] it was shown that there is a proper hierarchy of the recognizing power of n-ptwa with respect to n, moreover, there is a regular tree language which cannot be recognized by any ptwa. Formally,

$$TWA \subset 1\text{-}PTWA \subset 2\text{-}PTWA \subset \ldots \subset PTWA \subset REG,$$

where n-$PTWA$ denotes the class of tree languages recognized by n-ptwa, for $n \geq 0$, and $PTWA = \bigcup_{n \geq 0} n\text{-}PTWA$.

It was also an interesting and surprising result of [5] that ptwa with strong pebble handling have the same recognizing power as those with weak pebble handling.

Alternation was introduced in [7] as a natural generalization of nondeterminism for Turing machines, finite automata, and pushdown automata. Due to the generality of the concept, it is obvious how to define alternation for other types of sequential automata. For various kinds of (sequential) tree automata, alternation

was first investigated in [23]. In [10] it was left as an open problem, whether the tree languages recognized by n-ptwa with alternation and strong pebble handling are regular or not.

In the remainder of this paper we will consider pebble tree-walking devices only with strong pebble handling. Moreover, the definition of n-patwa in the present paper will follow the line of the definition of n-pebble tree transducers of [11].

A computation of an *n-pebble alternating tree-walking automaton (n-patwa)* A on an input tree s starts in the initial state with the pointer at the root node, and there are no pebbles on s. Depending on the applicable rules it generates new parallel computations (such that each has its own copy of s with the current position of the pointer, and the pebbles). The automaton A accepts s if all the computations spawned from the initial configuration terminate in the (only) accepting state q_{yes}. We say that L is the tree language *recognized by* A if L contains exactly the trees accepted by A. In case $n = 0$, we write *alternating tree-walking automaton (atwa)* for 0-patwa. We denote the tree language class recognized by n-patwa, deterministic n-patwa, atwa, and deterministic atwa by n-$PATWA$, n-$dPATWA$, $ATWA$, and $dATWA$, respectively. The unions $\bigcup_{n\geq 0} n\text{-}PATWA$ and $\bigcup_{n\geq 0} n\text{-}dPATWA$ are denoted by $PATWA$ and $dPATWA$, respectively.

As main result of this paper, we answer the open problem raised at page 18 of [10] and prove that for all $n \geq 0$, n-patwa recognize exactly the regular tree languages, i.e., n-$PATWA = REG$.

Roughly speaking, an n-patwa A is *looping* if there is an input tree s such that one of the computations of A on s gets into an infinite loop. Otherwise A is *non-looping*. We denote the non-looping version of the above tree language classes by subscripting an 'nl' to their names e.g. $dTWA_{nl}$, $dATWA_{nl}$, n-$dPATWA_{nl}$, etc.

In the second part of this paper we investigate the recognizing power of deterministic non-looping subclasses of the above tree language classes and show that these subclasses do not recognize all the regular tree languages, moreover the following proper inclusion hierarchy holds:

$dTWA \subset dATWA_{nl} \subset 1\text{-}dPATWA_{nl} \subset 2\text{-}dPATWA_{nl} \ldots \subset dPATWA_{nl} \subset REG.$ (*)

The paper is organized as follows. In Section 2 we define the necessary concepts. In Section 3 we give the formal definition of an n-patwa and define the looping property for them. In Section 4 we present our main result and prove that n-patwa recognize the regular tree languages. In Section 5 we prove the proper hierarchy (*). Finally, in Section 6 we conclude our results and give some future research topics.

2 Preliminaries

2.1 Sets, strings, and relations

We denote the set of nonnegative integers by $\mathbb{N}$. For every $n \in \mathbb{N}$, we let $[n] = \{1, \ldots, n\}$.

For a set A, $\mathcal{P}(A)$ denotes the power set of A. The empty set is denoted by $\emptyset$. If it does not lead to confusion, we write a for a singleton set $\{a\}$.

For a set A, A^* denotes the set of *strings* (or: *words*) over A; the *empty string* is denoted by ε. For a string $w \in A^*$, $|w|$ denotes its *length*. For every $n \geq 0$, we define $A^{\leq n} = \{u \in A^* \mid |u| \leq n\}$. For every $u \in A^*$, and $1 \leq l \leq |u|$, $u(l)$ denotes the l-th element of A in u.

An *alphabet* is a finite nonempty set. Let A be an alphabet and $L \subseteq A^*$ a finite, nonempty set. We write the strings of L^* in the form $[u_1; \ldots; u_l]$, where $l \geq 0$ and $u_1, \ldots, u_l \in L$. The empty string over L is denoted by $[]$.

Let $\rho \subseteq H \times H$ be a binary relation. The fact that $(a, b) \in \rho$ for some $a, b \in H$ is also denoted by $a \rho b$. For every $l \geq 0$, the l-th power of ρ is denoted by ρ^l, the transitive closure, and the reflexive, transitive closure of ρ are denoted by ρ^+ and ρ^*, respectively.

2.2 Trees and tree languages

A *ranked set* is an ordered pair $(\Sigma, rank_\Sigma)$, where Σ is a set and $rank_\Sigma$ is a mapping of type $\Sigma \to \mathbb{N}$. If Σ is an alphabet, then $(\Sigma, rank_\Sigma)$ is a *ranked alphabet*. If $rank_\Sigma(\sigma) = k$ for $\sigma \in \Sigma$ and $k \geq 0$, then the rank of σ is k and we indicate this fact also by writing $\sigma^{(k)}$. For every $k \geq 0$, we define $\Sigma^{(k)} = \{\sigma \in \Sigma \mid rank_\Sigma(\sigma) = k\}$. If Σ is clear from the context, we write $rank$ instead of $rank_\Sigma$, moreover, we drop $rank_\Sigma$ and write a ranked set as Σ.

We denote by $maxrank(\Sigma)$ the maximum of ranks of symbols of Σ, i.e., $maxrank(\Sigma) = \max\{rank(\sigma) \mid \sigma \in \Sigma\}$.

Let Σ be a ranked set. The set of *trees over* Σ, denoted by T_Σ, is the smallest set of strings $T \subseteq (\Sigma \cup \{(,)\} \cup \{,\})^*$ such that $\Sigma^{(0)} \subseteq T$ and whenever $k \geq 1$, $\sigma \in \Sigma^{(k)}$, and $t_1, \ldots, t_k \in T$, then $\sigma(t_1, \ldots, t_k) \in T$. Certainly, $T_\Sigma \neq \emptyset$ if and only if $\Sigma^{(0)} \neq \emptyset$.

For every tree $s \in T_\Sigma$, we define the set $pos(s) \subseteq [maxrank(\Sigma)]^*$ of *the nodes of* s as follows. We let $pos(s) = \{\varepsilon\}$ if $s \in \Sigma^{(0)}$, and $pos(s) = \{iu \mid 1 \leq i \leq k, u \in pos(s_i)\}$ if $s = \sigma(s_1, \ldots, s_k)$ for some $k \geq 1$, $\sigma \in \Sigma^{(k)}$ and $s_1, \ldots, s_k \in T_\Sigma$.

Now, for a tree $s \in T_\Sigma$ and a node $u \in pos(s)$, we define $lab(s, u) \in \Sigma$, i.e., *the label of* s *at node* u, by induction:

(i) if $s \in \Sigma^{(0)}$ (which implies $u = \varepsilon$), then $lab(s, u) = s$;

(ii) if $s = \sigma(s_1, \ldots, s_k)$ for some $k \geq 1$, $\sigma \in \Sigma^{(k)}$ and trees $s_1, \ldots, s_k \in T_\Sigma$, then

- if $u = \varepsilon$, then $lab(s, u) = \sigma$,
- if $u = iu'$, where $1 \leq i \leq k$, and $u' \in pos(s_i)$, then $lab(s, u) = lab(s_i, u')$.

For every $s \in T_\Sigma$ and $u \in pos(s)$ we define the parent of u, denoted by $parent(u)$ and the child number of u, denoted by $childno(u)$ as follows:

(i) if $u = \varepsilon$, then $childno(u) = 0$ and $parent(u)$ is undefined,

(ii) if $u = u'j$ for some $u' \in pos(s)$ and $j \in \mathbb{N}$, then $childno(u) = j$ and $parent(u) = u'$.

If Σ is a ranked alphabet, then any subset $L \subseteq T_\Sigma$ is a *tree language*. The *complement of* L is the tree language $\overline{L} = T_\Sigma - L$. If $\mathcal{L}$ is a class of tree languages, then $co\text{-}\mathcal{L} = \{\overline{L} \mid L \in \mathcal{L}\}$.

We will need a tree recognizer concept called *top-down tree automaton*. The unfamiliar reader can consult with [17, 18] for this concept, although they called it *root-to-frontier automaton*. A tree language is *regular*, if it can be recognized by a top-down tree automaton. We denote the class of regular tree languages by *REG*. The following classical result saying that regular tree languages are closed under complementation will be needed later.

Proposition 2.1 *REG = co-REG.*

2.3 MSO logic for trees

Monadic second order (MSO) logic was originally proposed to describe properties of strings in [6]. MSO logic can be extended for trees, see [24, 8, 2]. We will recall the syntax and the semantics of this logic over a ranked alphabet Σ.

Syntax:

We define the language $MSOL(\Sigma)$ of *MSO formulas (over* Σ*)*. This language is built up from the following symbols.

node variables: $x, y, x_1, x_2, \ldots$. We denote the set of node variables by VAR_1.

node-set variables: $X, X_1, X_2, \ldots$. We denote the set of node-set variables by VAR_2.

other symbols: $\neg$, $\wedge$, $\exists$, (,)

Atomic formulas are strings of one of the following types:

- $lab_\sigma(x)$, where $\sigma \in \Sigma$, and $x \in VAR_1$,
- $child_i(x_1, x_2)$, where $1 \leq i \leq maxrank(\Sigma)$, and $x_1, x_2 \in VAR_1$,
- $x \in X$, where $x \in VAR_1$, and $X \in VAR_2$.

The language of *MSO formulas over* Σ is the smallest set $MSOL(\Sigma)$ satisfying the following conditions.

(i) Each atomic formula is a formula of $MSOL(\Sigma)$.

(ii) Let $\phi_1, \phi_2 \in MSOL(\Sigma)$, $x \in VAR_1$, and $X \in VAR_2$. Then $(\neg\phi_1)$, $(\phi_1 \wedge \phi_2)$, $\exists x(\phi_1)$, $\exists X(\phi_1) \in MSOL(\Sigma)$.

Let $\phi \in MSOL(\Sigma)$ be an MSO formula and x (X) a node (node-set) variable in ϕ. Then an occurrence of x (X) in ϕ is said to be *free* in ϕ, if x (X) is not in the scope of $\exists x$ ($\exists X$), otherwise that occurrence is *bound* in ϕ. The formulas without free occurrences of node and node-set variables are the *closed formulas*.

Semantics:

The truth value of a formula is considered through structures. A *structure (over* Σ*)* is a triple (s, Π_1, Π_2), where

- $s \in T_\Sigma$,
- $\Pi_1 : VAR_1 \to pos(s)$, and
- $\Pi_2 : VAR_2 \to \mathcal{P}(pos(s))$.

Now, let (s, Π_1, Π_2) be a structure and $\phi \in MSOL(\Sigma)$ a formula. We define that the structure (s, Π_1, Π_2) *models* $\phi \in MSOL(\Sigma)$, or ϕ *is true in* (s, Π_1, Π_2) (denoted by $(s, \Pi_1, \Pi_2) \models \phi$) by formula induction on ϕ as follows.

(i)/a If $\phi = lab_\sigma(x)$, then $(s, \Pi_1, \Pi_2) \models \phi$ iff the label of the node $\Pi_1(x)$ is σ.

(i)/b If $\phi = child_i(x_1, x_2)$, then $(s, \Pi_1, \Pi_2) \models \phi$ iff node $\Pi_1(x_2)$ is the parent of node of $\Pi_1(x_1)$ and $childno(\Pi_1(x_1)) = i$.

(i)/c If $\phi = x \in X$, then $(s, \Pi_1, \Pi_2) \models \phi$ iff $\Pi_1(x) \in \Pi_2(X)$.

(ii)/a If $\phi = (\neg\phi_1)$, then $(s, \Pi_1, \Pi_2) \models \phi$ iff $(s, \Pi_1, \Pi_2) \not\models \phi_1$.

(ii)/b If $\phi = (\phi_1 \wedge \phi_2)$, then $(s, \Pi_1, \Pi_2) \models \phi$ iff $(s, \Pi_1, \Pi_2) \models \phi_1$, and $(s, \Pi_1, \Pi_2) \models \phi_2$.

(ii)/c If $\phi = \exists x(\phi_1)$, then $(s, \Pi_1, \Pi_2) \models \phi$ iff there is a node $u \in pos(s)$ and a structure (s, Π_1', Π_2), such that for every $y \in VAR_1$, we have

$$\Pi_1'(y) = \begin{cases} u & \text{if } y = x, \\ \Pi_1(y) & \text{if } y \neq x, \end{cases}$$

and $(s, \Pi_1', \Pi_2) \models \phi_1$.

(ii)/d If $\phi = \exists X(\phi_1)$, then $(s, \Pi_1, \Pi_2) \models \phi$ iff there is a node set $U \subseteq pos(s)$ and a structure (s, Π_1, Π_2'), such that for every $Y \in VAR_2$, we have

$$\Pi_2'(Y) = \begin{cases} U & \text{if } Y = X, \\ \Pi_2(Y) & \text{if } Y \neq X, \end{cases}$$

and $(s, \Pi_1, \Pi_2') \models \phi_1$.

To improve the readability of a formula, we omit the outer brackets. Moreover, we will use the standard shorthand $\phi_1 \vee \phi_2$ for $\neg\phi_1 \wedge \neg\phi_2$, $\phi_1 \to \phi_2$ for $\neg\phi_1 \vee \phi_2$, $\forall x\phi$ for $\neg\exists x\neg\phi$, and $\forall X\phi$ for $\neg\exists X\neg\phi$.

It is straightforward that for a closed formula $\phi \in MSOL(\Sigma)$, and structure (s, Π_1, Π_2), the mappings Π_1 and Π_2 do not influence the fact that $(s, \Pi_1, \Pi_2) \models \phi$ or not. Hence for a closed formula ϕ, we will write $s \models \phi$ for $(s, \Pi_1, \Pi_2) \models \phi$.

Let $\phi \in MSOL(\Sigma)$ be a closed formula. The tree language *defined by* ϕ is the tree language $L(\phi) = \{s \in T_\Sigma \mid s \models \phi\}$. A tree language $L \subseteq T_\Sigma$ is *MSO-definable*, if there is a closed formula $\phi \in MSOL(\Sigma)$, where $L = L(\phi)$. The following classical result from [8, 24] states that the MSO-definable tree languages are exactly the regular tree languages.

Proposition 2.2 A tree language is MSO-definable if and only if it is regular. ◇

3 Pebble alternating tree-walking automata

3.1 Syntax and semantics

In this section we introduce the concept of an *n-pebble alternating tree-walking automaton.* For this, we define the set of *instructions.*

Definition 3.1 For every integer $d \geq 0$, let

$$I_d = \{stay, up, drop, lift, down_1, down_2, \ldots, down_d\}.$$

The elements of I_d are called *instructions.*

For a ranked alphabet Σ, symbol $\sigma \in \Sigma$, $n \geq 0$, bit vector $b \in \{0,1\}^{\leq n}$, and $j \in \{0, 1, \ldots, maxrank(\Sigma)\}$, let $I_{\sigma,b,j,n} \subseteq I_{rank(\sigma)}$ be the smallest set satisfying the following conditions:

(i) $stay \in I_{\sigma,b,j,n}$,

(ii) if $j \neq 0$, then $up \in I_{\sigma,b,j,n}$,

(iii) for every $1 \leq i \leq rank(\sigma)$ we have $down_i \in I_{\sigma,b,j,n}$,

(iv) if $|b| < n$, then $drop \in I_{\sigma,b,j,n}$,

(v) if $b \neq \varepsilon$, then $lift \in I_{\sigma,b,j,n}$.

If n is clear from the context, then we write $I_{\sigma,b,j}$ for $I_{\sigma,b,j,n}$. ◇

Definition 3.2 For $n \geq 0$, an *n-pebble alternating tree-walking automaton* (shortly *n-patwa*) is a system $A = (Q, \Sigma, q_0, q_{yes}, R)$, where

- Q is a finite nonempty set, the *set of states*, which is partitioned into pairwise disjoint subsets $Q_0, Q_1, \ldots, Q_n$,
- Σ is a ranked alphabet, the *input alphabet*,
- $q_0 \in Q_0$ is a distinguished state, the *initial state*,
- $q_{yes} \notin Q$ is a new state, the *accepting state*,
- R is a finite *set of rules*, which is partitioned into pairwise disjoint subsets $R_0, R_1, \ldots, R_n$, such that for each $0 \leq i \leq n$, the set R_i consists of
  - *accepting rules* of the form $\langle q, \sigma, b, j\rangle \to \langle q_{yes}, stay\rangle$,
  - *pebble tree-walking rules* of the form $\langle q, \sigma, b, j\rangle \to \langle p, \varphi\rangle$, and
  - *alternating rules* of the form $\langle q, \sigma, b, j\rangle \to \{\langle p_1, stay\rangle, \langle p_2, stay\rangle\}$,

where $q \in Q_i$, $\sigma \in \Sigma$, $b \in \{0,1\}^i$, $0 \leq j \leq maxrank(\Sigma)$, $p_1, p_2 \in Q_i$, $\varphi \in I_{\sigma,b,j}$, moreover

$$p \in \begin{cases} Q_i & \text{if } \varphi \in \{stay, up, down_1, down_2, \ldots\}, \\ Q_{i+1} & \text{if } \varphi = drop, \text{ and} \\ Q_{i-1} & \text{if } \varphi = lift. \end{cases}$$

◇

By a *pebble alternating tree-walking automaton (patwa)* we mean an n-patwa for some n.

A tree $s \in T_\Sigma$ is called an *input tree to* A or just an input tree. In the remainder of this section A stands for the n-patwa $A = (Q, \Sigma, q_0, q_{yes}, R)$.

We say that A is *deterministic*, if, for every $q \in Q$, $\sigma \in \Sigma$, $b \in \{0,1\}^{\leq n}$, and $j \in \{0, 1, \ldots, maxrank(\Sigma)\}$, there is at most one rule of R with left-hand side $\langle q, \sigma, b, j \rangle$. Next we introduce further syntactic restrictions for patwa.

Definition 3.3 A is

- an *alternating tree-walking automaton* (shortly *atwa*), if A is a 0-patwa.
- an *n-pebble tree-walking automaton* (shortly *n-ptwa*)[10], if there are no alternating rules in R.
- a *tree-walking automaton* (shortly *twa*)[1], if A is a 0-ptwa. ◇

By a *pebble tree-walking automaton (ptwa)* we mean an n-ptwa for some n. Next we make some preparation for defining the semantics of a patwa. First we define the concept of an n-pebble configuration.

Definition 3.4 For an input tree $s \in T_\Sigma$, an *n-pebble configuration* (or: *pebble configuration*) over s (and A) is a pair $h = (u, \pi)$, where $u \in pos(s)$ is a node of s and $\pi \in (pos(s))^{\leq n}$, i.e., π is a string over $pos(s)$ of length at most n. The set of pebble configurations over s and A is denoted by $PC_{A,s}$. ◇

A pebble configuration $h = (u, \pi) \in PC_{A,s}$, with the string of strings $\pi = [u_1; \ldots; u_l]$ contains the information that the node being scanned by A (the current node) of the input tree s is u and A put $l = |\pi|$ pebbles on the nodes $u_1, \ldots, u_l$ of s. Note that more than one pebble can be put on the same node.

We define a mapping that tests a pebble configuration and returns a triple, which will influence the computation relation.

Definition 3.5 Let $s \in T_\Sigma$ be an input tree and $h = (u, \pi) \in PC_{A,s}$ a pebble configuration. Then $test_s(h) = (\sigma, b, j)$, where

- $\sigma = lab(s, u)$,
- $b \in \{0,1\}^*$ is a string (bit vector) of length $l = |\pi|$, where, for every $1 \leq i \leq l$,

$$b(i) = \begin{cases} 1 & \text{if } \pi(i) = u \\ 0 & \text{if } \pi(i) \neq u, \end{cases}$$

(Note, it follows from Definition 3.4 that $l \leq n$.)

- $j = childno(u)$. ◇

If s is clear from the context, then we write $test(h)$ for $test_s(h)$. Next we define how an instruction can be executed on a configuration.

Definition 3.6 Let $s \in T_\Sigma$ be an input tree and $h = (u, \pi) \in PC_{A,s}$ an n-pebble configuration over s with $\pi = [u_1; \ldots; u_l]$. Let $test(h) = (\sigma, b, j)$ and take an instruction $\varphi \in I_{test(h)} = I_{\sigma,b,j}$. The *execution of* φ *on* h is the pebble configuration $\varphi(h)$ defined in the following way.

$$\varphi(h) = \varphi((u,\pi)) = \begin{cases} (u, \pi) & \text{if } \varphi = stay, \\ (parent(u), \pi) & \text{if } \varphi = up, \\ (ui, \pi) & \text{if } \varphi = down_i, \\ (u, [u_1; \ldots; u_l; u]) & \text{if } \varphi = drop, \\ (u, [u_1; \ldots; u_{l-1}]) & \text{if } \varphi = lift. \end{cases}$$

◇

Now we define the concept of a *configuration of* A.

Definition 3.7 Let $s \in T_\Sigma$ be an input tree. A *configuration of* A (over s) is a pair $\langle q, h \rangle$, where $q \in Q \cup \{q_{yes}\}$ and $h \in PC_{A,s}$. ◇

Roughly speaking, a configuration is a snapshot of the computation, storing the current state, the node pointed at by the pointer, and the positions of the dropped pebbles. The set of configurations of A over s is denoted by $C_{A,s}$.

Due to alternation, A is capable to do arbitrary many parallel computations (threads) while processing s and hence, the computation relation works over the subsets of $C_{A,s}$. We turn to introduce this computation relation.

Definition 3.8 Let $s \in T_\Sigma$ be an input tree. Then $\vdash_{A,s} \subseteq \mathcal{P}(C_{A,s}) \times \mathcal{P}(C_{A,s})$ is the *computation relation of* A *on* s, where for all configuration sets $H_1, H_2 \in \mathcal{P}(C_{A,s})$ we have $H_1 \vdash_{A,s} H_2$ if and only if there is a configuration $\langle q, h \rangle \in H_1$, such that one of the following is true.

(1) There is an accepting rule $\langle q, \sigma, b, j \rangle \to \langle q_{yes}, stay \rangle$ in R such that $test(h) = (\sigma, b, j)$ and $H_2 = (H_1 - \{\langle q, h \rangle\}) \cup \{\langle q_{yes}, h \rangle\}$.

(2) There is a pebble tree-walking rule $\langle q, \sigma, b, j \rangle \to \langle p, \varphi \rangle$ in R such that $test(h) = (\sigma, b, j)$ and $H_2 = (H_1 - \{\langle q, h \rangle\}) \cup \{\langle p, \varphi(h) \rangle\}$.

(3) There is an alternating rule $\langle q, \sigma, b, j \rangle \to \{\langle p_1, stay \rangle, \langle p_2, stay \rangle\}$ in R such that $test(h) = (\sigma, b, j)$ and $H_2 = (H_1 - \{\langle q, h \rangle\}) \cup \{\langle p_1, h \rangle, \langle p_2, h \rangle\}$. ◇

We note that the role of the alternating rules is to spawn two parallel computations (threads) from one computation, such that the two new computations start out to work from the current pebble configuration. Moreover, each parallel computation has its own copy of the input tree and an own pebble configuration, which cannot be modified by other computations.

Pebble tree-walking rules are responsible for the sequential steps of a computation (moving on the edges, dropping and lifting of pebbles), and accepting rules are for terminating and accepting a computation.

The n-patwa A works as follows on an input tree s. It starts out in the *initial configuration set* $\{\langle q_0, (\varepsilon, [])\rangle\}$ (i.e., only one thread, initial state, pointer at the root node, and no pebbles dropped on s). Then, applying $\vdash_{A,s}$ step by step, it computes further configuration sets. The goal is that each parallel computation spawned from the initial configuration should be accepting, in other words, to terminate in a special configuration set $H \in \mathcal{P}(C_{A,s})$, such that the state-component of each configuration of H is q_{yes}. In that case H is an *accepting configuration set*. It is easy to see that there is no computation step from an accepting configuration set.

Let $ACC_{A,s} = \{q_{yes}\} \times PC_{A,s}$ be the largest accepting configuration set. Thus the tree language recognized by A is defined as follows.

Definition 3.9 The *tree language recognized by* A is

$$L(A) = \{s \in T_\Sigma \mid \langle q_0, (\varepsilon, [])\rangle \vdash^*_{A,s} H, \text{ for some } H \subseteq ACC_{A,s}\}. \qquad \diamond$$

The classes of tree languages computed by n-patwa, atwa, n-ptwa, and twa are denoted by n-$PATWA$, $ATWA$, n-$PTWA$, and TWA, respectively. The unions $\bigcup_{n\geq 0} n$-$PATWA$, and $\bigcup_{n\geq 0} n$-$PTWA$ are denoted by $PATWA$, and $PTWA$, respectively. The deterministic subclasses of the above tree language classes are denoted by prefixing a letter 'd' in front of their names, e.g., n-$dPATWA$, $dATWA$.

It should be clear that with the growing number of pebbles, the recognizing power of patwa and ptwa do not decrease, i.e., n-$PATWA \subseteq (n+1)$-$PATWA$, and n-$PTWA \subseteq (n+1)$-$PTWA$ for every $n \geq 0$.

3.2 Looping and non-looping patwa

Now we turn to the looping property of patwa. Roughly speaking, A is looping, if it has an infinite computation on an input tree.

We introduce the looping property for patwa similarly as the circularity concept was introduced for attributed grammars [20, 19], attributed tree transducers [12, 16], and pebble (macro) tree transducers [11, 14, 15].

We say that $\langle q, h\rangle \in C_{A,s}$ is a *looping configuration*, if there is a configuration set $H \subseteq C_{A,s}$, such that $\langle q, h\rangle \in H$ and $\langle q, h\rangle \vdash^+_{A,s} H$. Moreover, A is *looping*, if there is an input tree $s \in T_\Sigma$, a configuration set $H \subseteq C_{A,s}$ such that

- H contains a looping configuration and
- $\langle q_0, (\varepsilon, [])\rangle \vdash^*_{A,s} H$.

Otherwise, A is *non-looping*.

The looping property for pebble macro tree transducers appear in [15] by name "strong circularity". Let us denote the non-looping version of the above tree language classes by n-$PATWA_{nl}$, n-$dPATWA_{nl}$, etc.

3.3 Patwa with general alternating rules

When using alternation in a patwa, sometimes it is convenient not to be restricted to the forms of the possible right-hand sides of alternating rules of Definition 3.2. It should be clear that we can allow not only two, but arbitrary many state-instruction pairs for the right-hand sides of the alternating rules. Moreover, in the right-hand side of an alternating rule we allow not only *stay*, but arbitrary instructions. A *general alternating rule* is a rule of the form $\langle q, \sigma, b, j\rangle \to \{\langle q_1, \varphi_1\rangle, \ldots, \langle q_m, \varphi_m\rangle\}$ with $m \geq 1$, $\langle q_1, \varphi_1\rangle, \ldots, \langle q_m, \varphi_m\rangle \in Q \times I_{\sigma,b,j}$. Moreover, we assume that the state set is not partitioned, and $q_1, \ldots, q_m$ can be arbitrary states of $Q \cup \{q_{yes}\}$.

An *n-patwa with general alternating rules* is a tuple $A = (Q, \Sigma, q_0, q_{yes}, R)$, where R is a finite set of general alternating rules (and the rest is as for an n-patwa). For A, the notion 'deterministic', and the concept of 'configuration' are defined in the same way as for an n-patwa.

For defining the computation relation of A we remove point (1) and (2) and modify point (3) in Definition 3.8 in the following way.

(3) There is a general alternating rule $\langle q, \sigma, b, j\rangle \to \{\langle p_1, \varphi_1\rangle, \ldots, \langle p_m, \varphi_m\rangle\}$ in R such that $test(h) = (\sigma, b, j)$ and $H_2 = (H_1 - \{\langle q, h\rangle\}) \cup \{\langle p_1, \varphi_1(h)\rangle, \ldots, \langle p_m, \varphi_m(h)\rangle\}$.

Finally, the tree language $L(A)$ recognized by A is defined in the same way as in Definition 3.9, and the looping property of A can be defined similarly as in section 3.2.

We leave the proof of the following lemma to the reader.

Lemma 3.10 For every $n \geq 0$, and n-patwa A with general alternating rules, we can construct an n-patwa A', such that

- $L(A) = L(A')$,
- A is deterministic iff A' is deterministic, and
- A is non-looping iff A' is non-looping. ◇

3.4 Some notes about alternation and patwa

The idea of extending Turing machines and automata with alternation comes from [7]. The term "alternation" means the mixture of *existential nondeterminism* and *universal nondeterminism.*

Existential nondeterminism is the classical nondeterminism concept, i.e., a configuration will be accepting, if there is at least one accepting computation which starts out from that configuration. On the other hand, universal nondeterminism means, that a configuration is accepting, if all possible computations which start out of that configuration lead to acceptance.

The mixture of existential and universal nondeterminism is solved in the folklore by partitioning the state set Q into Q_{OR} (existential states), and Q_{AND} (universal states), moreover the configurations with states from Q_{OR} (resp. Q_{AND}) are regarded with existential nondeterminism (resp. universal nondeterminism).

Our definition of patwa differs from the usual alternating devices, because the present form of patwa is sometimes more handable in this paper. In our context, each state is existential. The universal nondeterminism for patwa is due to the alternating rules which spawn parallel computations, such that all of those computations should be accepting in order to accept the input.

However, it is easy to show that the definition of patwa with classical alternation (with existential, universal states and without alternating rules) would yield tree recognizers with the same recognizing power as patwa of the present paper have.

4 The recognizing power of patwa

In this section we show that the tree languages recognized by patwa are exactly the regular tree languages. We closely follow the ideas in the proof of Theorem 4.7 of [21].

It is easy to see that each top-down tree automaton can be simulated by a 0-patwa. To prove the converse, we will give a closed MSO formula ϕ for every patwa A, such that $L(\phi) = L(A)$. Then using Proposition 2.2, we will obtain that each tree language recognized by a patwa is regular. To make the proof more understandable, we will need some abbreviations of $MSOL(\Sigma)$ formulas, which are listed bellow. Let $d = maxrank(\Sigma)$ and $0 \leq i \leq d$ an arbitrary number.

- $\underline{x_1 = x_2} \equiv \forall X(x_1 \in X \leftrightarrow x_2 \in X)$ (that is true in (s, Π_1, Π_2), if node $\Pi_1(x_1)$ equals node $\Pi_1(x_2)$),
- $\underline{child(x_1, x_2)} \equiv child_1(x_1, x_2) \vee \ldots \vee child_d(x_1, x_2)$ (that is true in (s, Π_1, Π_2), if $\Pi_1(x_1)$ is a child of $\Pi_1(x_2)$),
- $\underline{root(x)} \equiv \forall x_1(\neg child(x, x_1))$ (that is true in (s, Π_1, Π_2), if $\Pi_1(x)$ is the root node)
- $\underline{root \in X} \equiv \forall x(root(x) \rightarrow x \in X)$ (that is true in (s, Π_1, Π_2), if the root node is in $\Pi_2(X)$),
- $\underline{chno_i(x)} \equiv \begin{cases} root(x) & \text{if } i = 0 \\ \exists x_1(child_i(x, x_1)) & \text{if } i > 0 \end{cases}$

  (that is true in (s, Π_1, Π_2), if the child number of $\Pi_1(x)$ is i), and
- $\underline{true} \equiv \forall x(x = x)$ (which is a valid formula).

In the remainder of this section let $n \geq 0$, $A = (Q, \Sigma, q_0, q_{yes}, R)$ an n-patwa, and $s \in T_\Sigma$ an input tree to A. We enumerate the states in Q such that $Q =$

$\{q_0, \ldots, q_m\}$, and $Q_0 = \{q_{m_0} = q_0, \ldots, q_{m_1-1}\}, Q_1 = \{q_{m_1}, \ldots, q_{m_2-1}\}, \ldots, Q_n = \{q_{m_n}, \ldots, q_{m_{n+1}-1} = q_m\}$. Let us observe that $m_{n+1} = m + 1$.

Next we give an alternative way to define that A accepts or rejects a tree s. In fact, we will define the acceptance through node sets $S_0, \ldots, S_m \subseteq pos(s)$, where for each $0 \leq i \leq m$, node set S_i is associated with state q_i of A. Note that for $0 \leq l \leq n$, the node sets concerned with the states of Q_l are $S_{m_l}, \ldots, S_{m_{l+1}-1}$, since $Q_l = \{q_{m_l}, \ldots, q_{m_{l+1}-1}\}$. Then, we show that the alternative definition of acceptance described below can be expressed by an MSO formula. Hence, by Proposition 2.2, it follows that the tree language accepted by A is regular.

We begin with some preparation. Namely, we define the *closed*, and the *strongly closed* properties for node sets $S_0, \ldots, S_m$.

Definition 4.1 Let $0 \leq l \leq n$, $u_1, \ldots, u_l \in pos(s)$, and $S_0, \ldots, S_{m_l-1}$, $S_{m_l}, \ldots, S_{m_{l+1}-1} \subseteq pos(s)$. We define the node sets $S_{m_l}, \ldots, S_{m_{l+1}-1}$ to be *l-closed with respect to A, s, $u_1, \ldots, u_l$, and* $S_0, \ldots, S_{m_l-1}$ by a downward induction on l as follows. (Note that the base of the induction is $l = n$.)

(i) If $l = n$, then the following statements hold.

(1) For every $m_l \leq \mu \leq m_{l+1} - 1$ and $u \in pos(s)$, if $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \langle q_{yes}, (u, [u_1; \ldots; u_l])\rangle$, then $u \in S_\mu$.

(2) For every $m_l \leq \mu, \nu \leq m_{l+1} - 1$ and $u, u' \in pos(s)$, if $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \langle q_\nu, (u', [u_1; \ldots; u_l])\rangle$, and $u' \in S_\nu$, then $u \in S_\mu$.

(3) For every $m_l \leq \mu, \nu_1, \nu_2 \leq m_{l+1} - 1$ and $u \in pos(s)$, if $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \{\langle q_{\nu_1}, (u, [u_1; \ldots; u_l])\rangle, \langle q_{\nu_2}, (u, [u_1; \ldots; u_l])\rangle\}$, $u \in S_{\nu_1}$, and $u \in S_{\nu_2}$, then $u \in S_\mu$.

(4) For every $m_l \leq \mu \leq m_{l+1} - 1$, $m_{l-1} \leq \nu \leq m_l - 1$ (provided that $l > 0$), and $u \in pos(s)$, if $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \langle q_\nu, (u, [u_1; \ldots; u_{l-1}])\rangle$ and $u \in S_\nu$, then $u \in S_\mu$.

(ii) Let $l < n$. Then (1)-(4) hold, moreover:

(5) For every $m_l \leq \mu \leq m_{l+1} - 1$, $m_{l+1} \leq \nu \leq m_{l+2} - 1$, and $u \in pos(s)$, if $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \langle q_\nu, (u, [u_1; \ldots; u_l; u])\rangle$, and for all node sets $S_{m_{l+1}}, \ldots, S_{m_{l+2}-1}$ that are $(l+1)$-closed with respect to A, s, $u_1, \ldots, u_l, u$, and $S_0, \ldots S_{m_{l+1}-1}$, we have $u \in S_\nu$, then $u \in S_\mu$. ◇

Definition 4.2 Let $0 \leq l \leq n$, $S_0, \ldots, S_{m_{l+1}-1} \subseteq pos(s)$, and $u_1, \ldots, u_l \in pos(s)$. We say that $S_0, \ldots, S_{m_{l+1}-1}$ are *strongly l-closed with respect to A, s,* and $u_1, \ldots, u_l$, if

$S_0, \ldots, S_{m_1-1}$ are 0-closed with respect to A and s,
$S_{m_1}, \ldots, S_{m_2-1}$ are 1-closed with respect to A, s, u_1, and $S_0, \ldots, S_{m_1-1}$,
$\vdots$
$S_{m_l}, \ldots, S_{m_{l+1}-1}$ are l-closed with respect to A, s, $u_1, \ldots, u_l$, and $S_0, \ldots, S_{m_l-1}$. ◇

In case $l = 0$ we make the following observation.

Observation 4.3 $S_0, \ldots, S_{m_1-1}$ are strongly 0-closed if and only if $S_0, \ldots, S_{m_1-1}$ are 0-closed with respect to A and s. ◇

Lemma 4.4 Let $u_1, \ldots, u_n \in pos(s)$ be arbitrary nodes. Define the node sets $T_0, \ldots, T_m \subseteq pos(s)$ such that for each $0 \leq l \leq n$, $m_l \leq \mu \leq m_{l+1} - 1$, and $u \in pos(s)$, we have $u \in T_\mu$ iff there is an accepting configuration set $H \subseteq ACC_{A,s}$ such that $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash^+_{A,s} H$.

Then $T_0, \ldots, T_m$ are strongly n-closed with respect to A, s, and $u_1, \ldots, u_n$.

Proof. Let $0 \leq l \leq n$. It suffices to prove that $T_{m_l}, \ldots, T_{m_{l+1}-1}$ are l-closed with respect to A, s, $u_1, \ldots, u_l$, and $T_0, \ldots, T_{m_l-1}$. We prove by induction on l. (Note that the induction base is $l = n$.)

(i) Let $l = n$. It is easy to see that properties (1) - (4) of Definition 4.1 hold for $T_{m_n}, \ldots, T_{m_{n+1}-1}$ with respect to A, s, $u_1, \ldots, u_n$, and $T_0, \ldots, T_{m_n-1}$.

(ii) Let $l < n$. Then, we can also easily see that properties (1) - (4) hold for $T_{m_l}, \ldots, T_{m_{l+1}-1}$ with respect to A, s, $u_1, \ldots, u_l$, and $T_0, \ldots, T_{m_l-1}$. Now we prove that property (5) of Definition 4.1 holds for $T_{m_l}, \ldots, T_{m_{l+1}-1}$ with respect to A, s, $u_1, \ldots, u_l$, and $T_0, \ldots, T_{m_l-1}$ as follows.

Let $m_l \leq \mu \leq m_{l+1} - 1$, $m_{l+1} \leq \nu \leq m_{l+2} - 1$, and $u \in pos(s)$, assume that

(*) $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \langle q_\nu, (u, [u_1; \ldots; u_l; u])\rangle$, and

(**) $u \in S_\nu$ for all node sets $S_{m_{l+1}}, \ldots, S_{m_{l+2}-1}$, that are $(l+1)$-closed with respect to A, s, $u_1, \ldots, u_l, u$, and $T_0, \ldots, T_{m_{l+1}-1}$.

Moreover, we define the node sets $T'_{m_{l+1}}, \ldots, T'_{m_{l+2}-1} \subseteq pos(s)$, such that for each $m_{l+1} \leq \eta \leq m_{l+2} - 1$, and $v \in pos(s)$, we have $v \in T'_\eta$ iff there is an accepting configuration set $H \subseteq ACC_{A,s}$ such that $\langle q_\eta, (v, [u_1; \ldots; u_l; u])\rangle \vdash^+_{A,s} H$.

We make the following observations.

a) By the induction hypothesis, the node sets $T'_{m_{l+1}}, \ldots, T'_{m_{l+2}-1}$ are $(l+1)$-closed with respect to A, s, $u_1, \ldots, u_l, u$, and $T_0, \ldots, T_{m_{l+1}-1}$.

b) Then, by (**) and a) we obtain that $u \in T'_\nu$.

c) By the definition of $T'_{m_{l+1}}, \ldots, T'_{m_{l+2}-1}$ and b), we obtain that there is an accepting configuration set $H \subseteq ACC_{A,s}$, such that $\langle q_\nu, (u, [u_1; \ldots; u_l; u])\rangle \vdash^+_{A,s} H$.

d) By (*) and c) we obtain that $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash^+_{A,s} H$.

Thus, $u \in T_\mu$, which confirms property (5) of Definition 4.1 for node sets $T_{m_l}, \ldots, T_{m_{l+1}-1}$ with respect to A, s, $u_1, \ldots, u_l$, and $T_0, \ldots, T_{m_l-1}$. With this, we have finished the proof of this lemma. ◇

In the following we show that the acceptance of A can be described in an alternative but equivalent way in terms of closed node sets.

Lemma 4.5 Let $0 \leq l \leq n$, $u, u_1, \ldots, u_l \in pos(s)$, and $m_l \leq \mu \leq m_{l+1} - 1$. The following statements are equivalent.

(a) There is a set of accepting configurations $H \subseteq ACC_{A,s}$, such that $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash^*_{A,s} H$.

(b) For every $S_0, \ldots, S_{m_{l+1}-1} \subseteq pos(s)$, if the node sets $S_0, \ldots, S_{m_{l+1}-1}$ are strongly l-closed with respect to A, s, and $u_1, \ldots, u_l$, then $u \in S_\mu$.

Proof. (direction "(a) $\Rightarrow$ (b)":) Let $k \geq 1$ and suppose that $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash^k_{A,s} H$ and that $S_0, \ldots, S_{m_{l+1}-1} \subseteq pos(s)$ are strongly l-closed with respect to $u_1, \ldots, u_l$. We prove by induction on k.

(i) Let $k = 1$. Then obviously, H is singleton and $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \langle q_{yes}, (u, [u_1; \ldots; u_l])\rangle = H$. By property (1) of Definition 4.1 we obtain that $u \in S_\mu$.

(ii) Let $k > 1$. Then we consider the following cases.

case 1: $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \langle q_\nu, (u', [u_1; \ldots; u_l])\rangle \vdash^{k-1}_{A,s} H$, where $m_l \leq \nu \leq m_{l+1} - 1$. Then, by the induction hypothesis $u' \in S_\nu$. Hence, by property (2) of Definition 4.1 we obtain that $u \in S_\mu$.

case 2: $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \{\langle q_{\nu_1}, (u, [u_1; \ldots; u_l])\rangle, \langle q_{\nu_2}, (u, [u_1; \ldots; u_l])\rangle\} \vdash^{k-1}_{A,s} H$, where $m_l \leq \nu_1, \nu_2 \leq m_{l+1} - 1$. Then, it is obvious that there are accepting configuration sets $H_1, H_2 \subseteq ACC_{A,s}$ and numbers $k_1, k_2 < k$ such that $\langle q_{\nu_1}, (u, [u_1; \ldots; u_l])\rangle \vdash^{k_1}_{A,s} H_1$ and $\langle q_{\nu_2}, (u, [u_1; \ldots; u_l])\rangle \vdash^{k_2}_{A,s} H_2$. By the induction hypothesis $u \in S_{\nu_1}$ and $u \in S_{\nu_2}$. Hence, by property (3) of Definition 4.1 we obtain that $u \in S_\mu$.

case 3: $l \geq 1$ and $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \langle q_\nu, (u, [u_1; \ldots; u_{l-1}])\rangle \vdash^{k-1}_{A,s} H$, where $m_{l-1} \leq \nu \leq m_l - 1$. Then, by the induction hypothesis $u \in S_\nu$. Hence, by property (4) of Definition 4.1, we obtain that $u \in S_\mu$.

case 4: $l < n$ and $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash_{A,s} \langle q_\nu, (u, [u_1; \ldots; u_l; u])\rangle \vdash^{k-1}_{A,s} H$, where $m_{l+1} \leq \nu \leq m_{l+2} - 1$. Let $S_{m_{l+1}}, \ldots, S_{m_{l+2}-1} \subseteq pos(s)$ be arbitrary $l+1$-closed node sets with respect to A, s, $u_1, \ldots, u_l, u$, and $S_0, \ldots, S_{m_{l+1}-1}$. Then, by the induction hypothesis $u \in S_\nu$. Hence, by property (5) of Definition 4.1 we obtain that $u \in S_\mu$.

(direction "(b) $\Rightarrow$ (a)":)

Let $u_{l+1}, \ldots, u_n \in pos(s)$ be arbitrary dummy nodes, and $T_0, \ldots, T_m \subseteq pos(s)$ the node sets defined in the same way as in Lemma 4.4. By that lemma, $T_0, \ldots, T_m$ are strongly n-closed with respect to A, s, and $u_1, \ldots, u_n$. From this fact and Definition 4.2 we obtain the following statement.

Statement: The node sets $T_0, \ldots, T_{m_{l+1}-1}$ are strongly l-closed with respect to A, s, and $u_1, \ldots, u_l$.

Now, assume that (b) holds. By (b) and our Statement we get that $u \in T_\mu$. It follows that there is an accepting configuration set $H \subseteq ACC_{A,s}$, such that $\langle q_\mu, (u, [u_1; \ldots; u_l])\rangle \vdash^+_{A,s} H$, and with this, we have finished the proof. $\diamond$

Corollary 4.6 $s \in L(A)$ if and only if for all 0-closed node sets $S_0, \ldots, S_{m_1-1}$, we have that $\varepsilon \in S_0$.

Proof.

$$
\begin{array}{ll}
 & s \in L(A) \\
\text{(Definition 3.9)} \iff & \exists H \subseteq ACC_{A,s} \text{ such that } \langle q_0, (\varepsilon, [\,])\rangle \vdash^*_{A,s} H \\
\text{(Lemma 4.5)} \iff & \text{for all strongly 0-closed node sets } S_0, \ldots, S_{m_1-1}, \\
 & \text{we have } \varepsilon \in S_0 \\
\text{(Observation 4.3)} \iff & \text{for all 0-closed node sets } S_0, \ldots, S_{m_1-1}, \\
 & \text{we have } \varepsilon \in S_0. \qquad \diamond
\end{array}
$$

Now we are ready to prove the main result of this section.

Theorem 4.7 For every $n \geq 0$, n-patwa recognize exactly the regular tree languages. Formally, $REG = n\text{-}PATWA$.

Proof. Clearly, already 0-patwa are capable to simulate classical top-down tree automata, hence each regular tree language is recognizable by an n-patwa for $n \geq 0$, i.e., $REG \subseteq n\text{-}PATWA$ for $n \geq 0$. For the converse, it suffices to prove that $L(A)$ is a regular tree language (since A is picked as an arbitrary n-patwa).

Now we construct an MSO-formula defining $L(A)$. The thorough reader will find this formula almost literally the same as the one in the proof of Theorem 4.7 of [21].

Let $b \in \{0,1\}^{\leq n}$ be a bitvector of length l. We define a predicate $pebbles_b(x, x_1, \ldots, x_l)$ with free variables $x, x_1, \ldots, x_l$ which is true in a structure (s, Π_1, Π_2), if the presence of pebbles at node $\Pi_1(x)$ agrees with b, assuming that l pebbles are on the input tree and the positions of pebbles $1, \ldots, l$ are $\Pi_1(x_1), \ldots, \Pi_1(x_l)$, respectively. The predicate $pebbles_b(x)$ is defined by induction on l.

(i) If $l = 0$, then $pebbles_b(x) = pebbles_\varepsilon(x) = true$.

(ii) If $l > 0$, then

$$
pebbles_b(x, x_1, \ldots, x_l) = \begin{cases} pebbles_{b'}(x, x_1, \ldots, x_{l-1}) \wedge (x_l = x) & \text{if } b = b'1 \\ pebbles_{b'}(x, x_1, \ldots, x_{l-1}) \wedge \neg(x_l = x) & \text{if } b = b'0 \end{cases}
$$

For every $\sigma \in \Sigma$, $b \in \{0,1\}^{\leq n}$, where $|b| = l$, and $0 \leq j \leq maxrank(\Sigma)$ let

$$\theta_{\sigma,b,j}(x) = lab_\sigma(x) \wedge pebbles_b(x, x_1, \ldots, x_l) \wedge chno_j(x)$$

be the formula with free first-order variables $x, x_1, \ldots, x_l$, which is true in a structure (s, Π_1, Π_2) iff the test result of the pebble configuration $(\Pi_1(x), [\Pi_1(x_1); \ldots; \Pi_1(x_l)])$ is (σ, b, j), see Definition 3.5.

For each $0 \leq l \leq n$ and $m_l \leq \mu \leq m_{l+1} - 1$, we give the formula $\phi^{(l)}_\mu$ as follows.

$$
\phi^{(l)}_\mu = \begin{cases} \forall X_0 \ldots \forall X_{m_1-1} \Big(0\text{-}closed \rightarrow root \in X_\mu\Big) & \text{if } l = 0 \\ \forall X_{m_l} \ldots \forall X_{m_{l+1}-1} \Big(l\text{-}closed \rightarrow x_l \in X_\mu\Big) & \text{if } l > 0, \end{cases}
$$

where

$$l\text{-}closed = \bigwedge_{r \in R_l} \psi_r,$$

and ψ_r's are defined as follows.

(1) If r is an accepting rule of the form $\langle q_\mu, \sigma, b, j\rangle \to \langle q_{yes}, stay\rangle$, where $q_\mu \in Q_l$, $0 \le l \le n$, and $\sigma \in \Sigma$, $b \in \{0,1\}^l$, $0 \le j \le maxrank(\Sigma)$, then

$$\psi_r = \forall x_{l+1}\Big(\theta_{\sigma,b,j}(x_{l+1}) \to x_{l+1} \in X_\mu\Big).$$

(2) If r is of the form $\langle q_\mu, \sigma, b, j\rangle \to \langle q_\nu, stay\rangle$, where $q_\mu, q_\nu \in Q_l$, $0 \le l \le n$, $\sigma \in \Sigma$, $b \in \{0,1\}^l$, and $0 \le j \le maxrank(\Sigma)$, then

$$\psi_r = \forall x_{l+1}\Big((\theta_{\sigma,b,j}(x_{l+1}) \wedge x_{l+1} \in X_\nu) \to x_{l+1} \in X_\mu\Big).$$

(3) If r is of the form $\langle q_\mu, \sigma, b, j\rangle \to \langle q_\nu, up\rangle$, where $q_\mu, q_\nu \in Q_l$, $0 \le l \le n$, $\sigma \in \Sigma$, $b \in \{0,1\}^l$, and $0 \le j \le maxrank(\Sigma)$, then

$$\psi_r = \forall x_{l+1}\forall y\Big((\theta_{\sigma,b,j}(x_{l+1}) \wedge child(x_{l+1}, y) \wedge y \in X_\nu) \to x_{l+1} \in X_\mu\Big).$$

(4) If r is of the form $\langle q_\mu, \sigma, b, j\rangle \to \langle q_\nu, down_i\rangle$, where $q_\mu, q_\nu \in Q_l$, $0 \le l \le n$, $\sigma \in \Sigma$, $b \in \{0,1\}^l$, and $0 \le j \le maxrank(\Sigma)$, then

$$\psi_r = \forall x_{l+1}\forall y\Big((\theta_{\sigma,b,j}(x_{l+1}) \wedge child_i(y, x_{l+1}) \wedge y \in X_\nu) \to x_{l+1} \in X_\mu\Big).$$

(5) If r is an alternating rule of the form $\langle q_\mu, \sigma, b, j\rangle \to \{\langle q_{\nu_1}, stay\rangle, \langle q_{\nu_2}, stay\rangle\}$, where $q_\mu, q_{\nu_1}, q_{\nu_2} \in Q_l$, $0 \le l \le n$, $\sigma \in \Sigma$, $b \in \{0,1\}^l$, and $0 \le j \le maxrank(\Sigma)$, then

$$\psi_r = \forall x_{l+1}\Big((\theta_{\sigma,b,j}(x_{l+1}) \wedge x_{l+1} \in X_{\nu_1} \wedge x_{l+1} \in X_{\nu_2}) \to x_{l+1} \in X_\mu\Big).$$

Moreover:

(6) If r is of the form $\langle q_\mu, \sigma, b, j\rangle \to \langle q_\nu, lift\rangle$, where $q_\mu \in Q_l$, $1 \le l \le n$, $\sigma \in \Sigma$, $b \in \{0,1\}^l$, $0 \le j \le maxrank(\Sigma)$, and $q_\nu \in Q_{l-1}$, then

$$\psi_r = \forall x_{l+1}\Big((\theta_{\sigma,b,j}(x_{l+1}) \wedge x_{l+1} \in X_\nu) \to x_{l+1} \in X_\mu\Big).$$

(7) If r is of the form $\langle q_\mu, \sigma, b, j\rangle \to \langle q_\nu, drop\rangle$, where $q_\mu \in Q_l$, $0 \le l \le n-1$, $\sigma \in \Sigma$, $b \in \{0,1\}^l$, $0 \le j \le maxrank(\Sigma)$, and $q_\nu \in Q_{l+1}$, then

$$\psi_r = \forall x_{l+1}\Big((\theta_{\sigma,b,j}(x_{l+1}) \wedge \phi_\nu^{(l+1)}) \to x_{l+1} \in X_\mu\Big).$$

We make the following observations concerning the formula $\phi_\mu^{(l)}$.

a) $\phi_\mu^{(l)}$ has free node-set variables $X_0, \ldots, X_{m_l-1}$, and free node variables $x_1, \ldots, x_l$.

(In particular, $\phi_\mu^{(0)}$ is a closed formula.)

b) The subformula *l-closed* of $\phi_\mu^{(l)}$ has free node-set variables $X_{m_l}, \ldots, X_{m_{l+1}-1}$, in addition to the free variables above, and *l-closed* is true in a structure (s, Π_1, Π_2) if and only if $\Pi_2(X_{m_l}), \ldots, \Pi_2(X_{m_{l+1}-1})$ are l-closed with respect to A, s, $\Pi_1(x_1), \ldots, \Pi_1(x_l)$ and $\Pi_2(X_0), \ldots, \Pi_2(X_{m_l-1})$.

Note that the conjunction of formulas of type (1)-(7) expresses Definition 4.1 for node sets $\Pi_2(X_{m_l}), \ldots, \Pi_2(X_{m_{l+1}-1})$ (with respect to A, s, $\Pi_1(x_1), \ldots, \Pi_1(x_l)$ and $\Pi_2(X_0), \ldots, \Pi_2(X_{m_l-1})$).

c) Hence, $\phi_\mu^{(l)}$ is true in a structure (s, Π_1, Π_2) if for all node-sets $S_{m_l}, \ldots, S_{m_{l+1}-1} \subseteq pos(s)$, l-closed with respect to A, s, $\Pi_1(x_1), \ldots, \Pi_1(x_l)$, and $\Pi_2(X_0), \ldots, \Pi_2(X_{m_l-1})$, we have that

- $root \in S_\mu$, if $l = 0$, or
- $\Pi_1(x_l) \in S_\mu$, if $l > 0$.

Thus, by Corollary 4.6, we obtain that $s \in L(A)$ if and only if $s \models \phi_0^{(0)}$. Hence, $L(A) = L(\phi_0^{(0)})$ and this concludes that the tree language recognized by A is MSO-definable, and thus it is regular. ◇

5 Inclusion results for patwa

In this section we investigate the recognizing power of deterministic and non-looping patwa with and without pebbles. First we collect the preliminary results, which are necessary for this section.

Theorem 1 of [4] says that deterministic tree-walking automata are less powerful than their nondeterministic counterparts. Formally, $dTWA \subset TWA$. We note that the separating tree language treated by [4] (which cannot be recognized by a deterministic twa) can be recognized already by a nondeterministic and non-looping twa. Thus, $dTWA_{nl} \subset TWA_{nl}$, and moreover, by Proposition 1 of [22] (saying that $dTWA = dTWA_{nl}$), we obtain the following "non-looping version" of the above proper inclusion result.

Proposition 5.1 $dTWA \subset TWA_{nl}$. ◇

Theorem 1. of [22] states that deterministic twa are closed under complementation.

Proposition 5.2 $dTWA = co\text{-}dTWA$. ◇

One of the main results of [5] is Theorem 1.1 saying that ptwa do not recognize all the regular tree languages, formally, $PTWA \subset REG$. Using the obvious fact that $PTWA_{nl} \subseteq PTWA$, we obtain the following proposition.

Proposition 5.3 $PTWA_{nl} \subset REG$. ◇

Moreover Theorem 1.2 of [5] says, that the expressive power of n-patwa is strictly less than the expressive power of $(n+1)$-patwa for each $n \geq 0$, formally, $n\text{-}PTWA \subset (n+1)\text{-}PTWA$. We note that Theorem 1.2 of [5] refines Theorem 2 of [3], which says that $TWA \subset REG$. However, we wish to obtain the "non-looping version" of the proper inclusion $n\text{-}PTWA \subset (n+1)\text{-}PTWA$. For this we make the following observations.

(1) In the preceding paragraph of Theorem 3 of [22] it was shown that for each n-ptwa A with weak pebble handling we can construct a non-looping n-ptwa A' with weak pebble handling, such that $L(A) = L(A')$.

(2) It was shown in Lemma 5.1 of [5] that for each n-ptwa A we can construct an n-ptwa A' with weak pebble handling, such that $L(A) = L(A')$.

(3) By Theorem 1.2 of [5], $n\text{-}PTWA \subset (n+1)\text{-}PTWA$.

We note that the "weak pebble handling" property for ptwa is discussed in the Introduction. By (1)-(3) we conclude the following proposition.

Proposition 5.4 For each $n \geq 0$, $n\text{-}PTWA_{nl} \subset (n+1)\text{-}PTWA_{nl}$. ◇

Now we prove that the complements of the tree languages of $n\text{-}dPATWA_{nl}$ form exactly the tree language class $n\text{-}PTWA_{nl}$.

Lemma 5.5 For each $n \geq 0$, $co\text{-}n\text{-}dPATWA_{nl} = n\text{-}PTWA_{nl}$.

Proof. $co\text{-}n\text{-}dPATWA_{nl} \subseteq n\text{-}PTWA_{nl}$: Let $A = (Q, \Sigma, q_0, q_{yes}, R)$ be a deterministic and non-looping n-patwa. We construct the (nondeterministic, non-looping) n-ptwa $A' = (Q, \Sigma, q_0, q_{yes}, R')$ where R' is the smallest set of rules satisfying the following conditions.

- For each $q \in Q$, $\sigma \in \Sigma$, $b \in \{0,1\}^{\leq n}$, and $j \in \{0, \ldots, maxrank(\Sigma)\}$, if there is no rule in R with left-hand side $\langle q, \sigma, b, j\rangle$, then the accepting rule $\langle q, \sigma, b, j\rangle \to \langle q_{yes}, stay\rangle$ is in R'.
- For each pebble tree-walking rule $\langle q, \sigma, b, j\rangle \to \langle p, \varphi\rangle$ of R it is also in R'.
- For each alternating rule $\langle q, \sigma, b, j\rangle \to \{\langle p_1, stay\rangle, \langle p_2, stay\rangle\}$, the pebble tree-walking rules $\langle q, \sigma, b, j\rangle \to \langle p_1, stay\rangle$ and $\langle q, \sigma, b, j\rangle \to \langle p_2, stay\rangle$ are in R'.

Since M is non-looping, it is obvious that also M' is non-looping. The proof of $L(A') = \overline{L(A)}$ is straightforward, hence we omit it.

$n\text{-}PTWA_{nl} \subseteq co\text{-}n\text{-}dPATWA_{nl}$: Let $A = (Q, \Sigma, q_0, q_{yes}, R)$ be a non-looping (nondeterministic) ptwa. We construct the deterministic patwa with general alternating rules $A' = (Q', \Sigma, q_0, q'_{yes}, R')$ (by Lemma 3.10 we are allowed to use general alternating rules) as follows.

- $Q' = Q \cup \{q_{yes}\}$, and
- R' is the smallest set of rules satisfying the following conditions.
  - For each $q \in Q$, $\sigma \in \Sigma$, $b \in \{0,1\}^{\leq n}$, and $j \in \{0, \ldots, maxrank(\Sigma)\}$, if there is no rule in R with left-hand side $\langle q, \sigma, b, j\rangle$, then the accepting rule $\langle q, \sigma, b, j\rangle \to \langle q'_{yes}, stay\rangle$ is in R'.

– For each $q \in Q$, $\sigma \in \Sigma$, $b \in \{0,1\}^{\leq n}$, and $j \in \{0, \ldots, maxrank(\Sigma)\}$, if $\{\langle q_1, \varphi_1\rangle, \ldots, \langle q_m, \varphi_m\rangle\}$ is the set of state-instruction pairs that are the right-hand sides of rules with left-hand side $\langle q, \sigma, b, j\rangle$, then the rule $\langle q, \sigma, b, j\rangle \to \{\langle q_1, \varphi_1\rangle, \ldots, \langle q_m, \varphi_m\rangle\}$ is in R'.

Again, it is obvious that A' is deterministic, non-looping, and we leave the proof $\overline{L(M)} = L(M')$ to the reader. ◇

Now we prove the following proper inclusion result.

Theorem 5.6 $dTWA \subset dATWA_{nl}$.

Proof. We prove by contradiction. Let us assume that $dTWA = dATWA_{nl}$. Then we make the following observations.

a) Obviously, $co\text{-}dTWA = co\text{-}dATWA_{nl}$.

b) By Proposition 5.2, $dTWA = co\text{-}dATWA_{nl}$.

c) By Lemma 5.5, $dTWA = TWA_{nl}$, which contradicts Proposition 5.1.

With this, we have proved this theorem. ◇

Next we prove that with the deterministic and non-looping n-patwa are strictly weaker than deterministic and non-looping $(n+1)$-patwa are.

Theorem 5.7 For each $n \geq 0$, $n\text{-}dPATWA_{nl} \subset (n+1)\text{-}dPATWA_{nl}$.

Proof. The inclusion $n\text{-}dPATWA_{nl} \subseteq (n+1)\text{-}dPATWA_{nl}$ is obvious. We prove the proper inclusion by contradiction. Let us assume that $n\text{-}dPATWA_{nl} = (n+1)\text{-}dPATWA_{nl}$. Then, applying operation 'co' to both sides of the equation we get that $co\text{-}n\text{-}dPATWA_{nl} = co\text{-}(n+1)\text{-}dPATWA_{nl}$. By Lemma 5.5 we obtain that $n\text{-}PTWA_{nl} = (n+1)\text{-}PTWA_{nl}$, which contradicts Proposition 5.4. ◇

Finally, we prove, that deterministic and non-looping patwa do not recognize all the regular tree languages.

Theorem 5.8 $dPATWA_{nl} \subset REG$.

Proof. $dPATWA_{nl} \subseteq REG$ comes from Theorem 4.7 The proper inclusion is proved by contradiction. Let us assume that $dPATWA_{nl} = REG$. Then we make the following observations.

a) Applying the operation 'co' to both sides, we obtain that $co\text{-}dPATWA_{nl} = co\text{-}REG$.

b) By Proposition 2.1 we obtain that $co\text{-}dPATWA_{nl} = REG$.

c) By Lemma 5.5 we get that $PTWA_{nl} = REG$, which contradicts Proposition 5.3. ◇

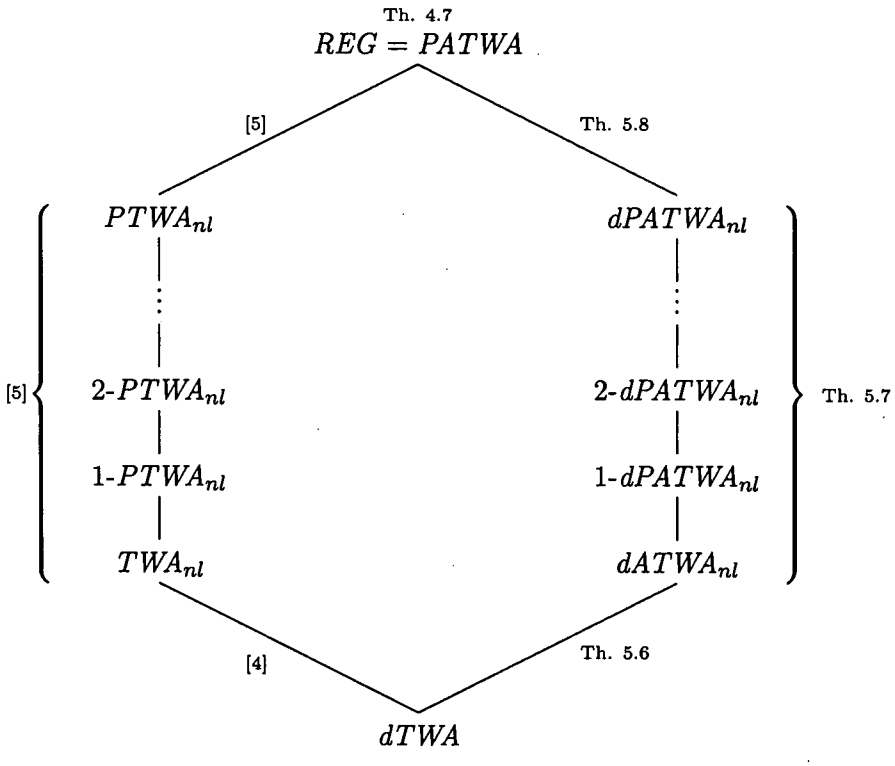


Figure 1: Inclusion diagram of some subclasses of *PATWA*. The continuous lines represent proper inclusions.

6 Conclusions

In this paper we gave a formal definition for pebble tree-walking automata extended with alternation [23] and have answered the open problem raised at page 18 in [10], which asked, whether the patwa recognize the class of regular tree languages. Our answer is yes, i.e., $PATWA = REG$.

In the remainder of this paper we have investigated the recognizing power of some subclasses of *PATWA*. We have come to the conclusion that

$$dTWA \subset dATWA_{nl} \subset 1\text{-}dPATWA_{nl} \subset \ldots \subset dPATWA_{nl} \subset REG.$$

However, it is still an open problem, whether $n\text{-}dPATWA \subset (n+1)\text{-}dPATWA$.

The most important known and new results are summarized in Figure 1.

We can find the relation between patwa and pebble tree transducers. It is trivial that the domain of each n-pebble tree transformation of [11] can be recognized by an n-patwa with *weak pebble handling*, i.e., pebbles can be lifted only from a node pointed at by the pointer. We can extend the pebble tree transducers of [11], such that the pebble in the input tree with the highest number can be lifted even from a node not pointed at by the pointer. This is the *strong pebble handling* (see

[10, 22, 5]), which is also used throughout the present paper. Then we can easily see that the domains of n-pebble tree transformations are exactly the tree languages recognized by n-patwa, i.e., the regular tree languages. Using the results of this paper, we obtain that

- $dom(n\text{-}PTT) = REG$, where $n \geq 0$, and
- $dTWA \subset dom(0\text{-}dPTT_{nl}) \subset dom(1\text{-}dPTT_{nl}) \subset \ldots \subset dom(dPTT_{nl}) \subset REG$,

assuming that $n\text{-}PTT$ is the class of n-pebble tree transformations, $PTT = \bigcup_{n\geq 0} n\text{-}PTT$ (the prefix 'd', and the subscription 'nl' denote the deterministic and non-looping subclasses), and for a tree transformation τ, the notation $dom(\tau)$ means the domain tree language of τ.

Another application of the inclusion result $dATWA_{nl} \subset 1\text{-}dPATWA_{nl} \subset REG$ is, that deterministic and non-looping atwa (0-patwa) are exactly the *tree-walking automata in universal acceptance mode* of [13], where it was proved that these automata recognize exactly the domains of partial attributed tree transformations [12]. It is straightforward that the non-looping version for this result also holds, i.e., non-looping tree-walking automata in universal acceptance mode recognize exactly the tree languages recognized by non-looping deterministic *atwa*, that are the domains of non-looping partial attributed tree transformations. Hence, we obtain that $dTWA \subset dom(ATT_{nl}) \subset 1\text{-}dPATWA_{nl} \subset REG$, where ATT_{nl} is the class of non-looping partial attributed tree transformations.

Acknowledgments

I thank Zoltán Fülöp for giving me a lot of useful advice. Moreover, I am grateful to the anonymous referee for his/her valuable ideas and suggestions which have rised the quality of this paper substantially.

References

[1] A. V. Aho and J. D. Ullman. Translations on a context–free grammar. *Inform. Control*, 19:439–475, 1971.

[2] R. Bloem and J. Engelfriet. Characterization of properties and relations defined in monadic second order logic on the nodes of trees. Technical Report Technical Report 97-03, Leiden University, August 1997.

[3] M. Bojańczyk and T. Colcombet. Tree-walking automata do not recognize all regular languages. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 234–243, New York, NY, USA, 2005. ACM Press.

[4] M. Bojańczyk and T. Colcombet. Tree-walking automata cannot be determinized. *Theoretical Computer Science*, 350:164–173, 2006.

[5] M Bojańczyk, M. Samuelides, T. Schwentick, and L. Segoufin. Expressive power of pebble automata. In *ICALP'06: Proceedings of 33rd International Colloquium on Automata, Languages and Programming*, pages 157–168. Springer Berlin / Heidelberg, 2006.

[6] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[7] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[8] J. Doner. Tree acceptors and some of their applications. *J. Comput. System Sci.*, 4:406–451, 1970.

[9] J. Engelfriet and H. J. Hoogeboom. Tree-walking pebble automata. In *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 72–83, London, UK, 1999. Springer-Verlag.

[10] J. Engelfriet and Hendrik Jan Hoogeboom. Automata with nested pebbles capture first-order logic with transitive closure. Technical Report 05-02, Leiden University, The Netherlands, April 2005.

[11] J. Engelfriet and S. Maneth. A Comparison of Pebble Tree Transducers with Macro Tree Transducers. *Acta Informatica*, 39:613–698, 2003.

[12] Z. Fülöp. On attributed tree transducers. *Acta Cybernet.*, 5:261–279, 1981.

[13] Z. Fülöp and S. Maneth. Domains of partial attributed transducers. *Inform. Process. Letters*, 73:175–180, 2000.

[14] Z. Fülöp and L. Muzamel. Decomposition Results for Pebble Macro Tree Transducers. Technical Report TUD-FI05-13, Technical University of Dresden, 2005.

[15] Z. Fülöp and L. Muzamel. Circularity and Decomposition Results for Pebble Macro Tree Transducers. *submitted*, 2006.

[16] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics — Formal Models Based on Tree Transducers.* Monographs in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 1998.

[17] F. Gécseg and M. Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[18] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer-Verlag, 1997.

[19] D. E. Knuth. Semantics of context-free languages: Correction. *Math. Systems Theory*, 5(1):95–96, 1971. Errata of [20].

[20] D.E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2:127–145, 1968.

[21] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. of Comp. Syst. Sci.*, 66:66–97, 2003.

[22] A. Muscholl, M. Samualides, and L. Segoufin. Complementing deterministic tree-walking automata. *Information Processing Letters*, 99:33–39, 2006.

[23] G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41:305–318, 1985.

[24] J. W. Thatcher and J.B. Wright. Generalized finite automata theory with application to a decision problem of second-order logic. *Math. Systems Theory*, 2(1):57–81, 1968.

Acta Cybernetica **18** (2008) 451–462.

Robust Clustering - Based Realtime Vowel Recognition

Dénes Paczolay*, András Bánhalmi*, and András Kocsor*†

Abstract

In the therapy of the hearing impaired one of the key problems is how to deal with the lack of proper auditive feedback which impedes the development of intelligible speech. The effectiveness of the therapy relies heavily on accurate phoneme recognition. Because of the environmental difficulties, simple recognition algorithms may have a weak classification performance, so various techniques such as normalization and classifier combination are applied to raising the overall recognition accuracy. In earlier work we came to realise that the classification accuracy is higher on a database that is manually clustered according to the gender and age of the speakers. This paper examines what happens when we cluster the database into a few groups automatically and then we train separate classifiers for each cluster. The results shows that this two-step method can increase the recognition performance by several percent.

Keywords: speech recognition, speech therapy, two-step classification method

1 Introduction

In the therapy of the hearing impaired one of the central problems is how to deal with the lack of proper auditive feedback that hinders the development of intelligible speech. Our Phonological Awareness Teaching System, the "SpeechMaster" package, seeks to apply speech recognition technology to speech therapy. It provides a visual phonetic feedback to supplement the insufficient auditive feedback of the hearing impaired. Our computer-aided training software package uses an effective phoneme recognizer and provides a realtime visual feedback in the form of flickering letters positioned over calling pictures.

*Research Group on Artificial Intelligence of the Hungarian Academy of Sciences and University of Szeged H-6720 Szeged, Aradi vértanúk tere 1., Hungary, E-mail: {pdenes, banhalmi, kocsor}@inf.u-szeged.hu

†Machine Intelligence Laboratory NPC., Petőfi S. Sgt. 43., H-6725 Szeged, Hungary, Applied Intelligence Laboratory Ltd., Petőfi S. Sgt. 43., H-6725 Szeged, Hungary, E-mail: kocsor@ail.hu

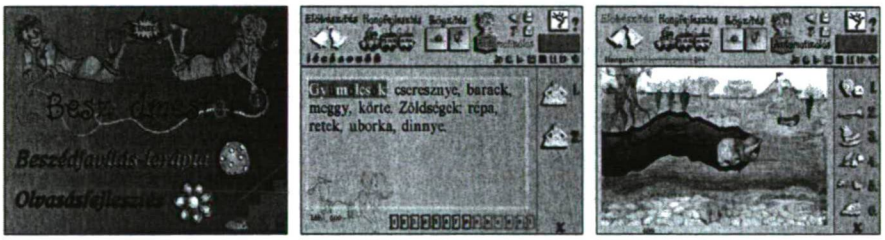
Since the system should work reliably for children of different ages and teachers as well, the recognizer has to be trained with the voices of users of both genders and of practically any age. The task is also special because the system has to recognize isolated phones, so it cannot rely on language models. Consequently, there is a heavy burden on the acoustic classifier, and we need to apply any helpful trick that might improve the overall performance.

During our previous work we found that the classification accuracy is generally higher on a homogeneous database (one whose gender and age are homogeneous) than a mixed database. This is probably because the variance of a homogeneous database is better than a mixed one. To train the latest version of SpeechMaster we applied training databases that were manually clustered according to speaker gender and age because we wished to achieve a higher recognition performance. This paper describes what happens when we cluster the database into several groups automatically and then train a separate classifier for each of these groups.

This paper is organized as follows. In the following section we will present our speech therapy system, then in Section 3 we will describe our previous study and experiences gained from it. In the next two sections we provide a brief description of the clustering and classification algorithms used in our tests. Section 6 then compares the performance of the various recognition methods. Lastly, we give some brief conclusions and ideas for the future.

2 Our therapy system: the SpeechMaster

The SpeechMaster package was developed for speech impediment therapy and teaching reading. The system is based on automatic speech recognition (machine learning [1, 3, 10]) and advanced signal processing methods. The developers co-operated with speech therapeutists and elementary school teachers, and tested the system with children in real environments. In the therapy of the hearing impaired one of the key problems is how to deal with the lack of proper auditive feedback - which, of course, impedes the development of intelligible speech. The idea is really to make the vocal sounds 'visible' for the hearing impaired. This way they are able to check their pronunciation by sight, that is, their hearing is supplemented



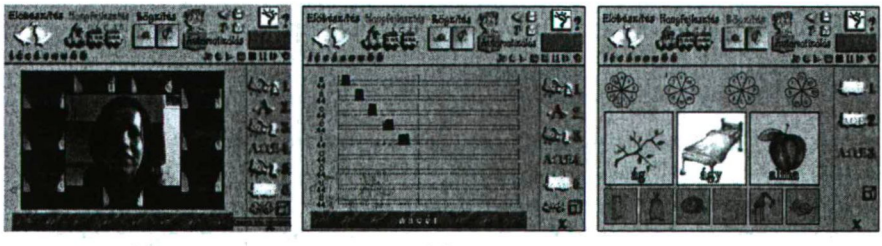
(a) (b) (c)

Figure 1: Screenshots from "SpeechMaster"

by visual input. The speech therapy of the hearing impaired traditionally requires enormous patience and the continuous presence of a teacher since, during the fixation of the correct sound-formation, a large amount of repetition and correction by the teacher are both needed. This automation process is significantly speeded up and simplified with our software, and also allows the students to practice on their own or with the teacher. In speech impediment therapy, at the beginning of the development of oral competence, it is recommended that young children concentrate mainly on their own voicing. This is supported by the creation of the many playful sound formation exercises. For each drill, skill and acceptance levels can be adjusted with a potentiometer. The software package also contains many useful features: customisable profiles, easily extendable word and image lists with sample utterances, half-speed sound replay, a web-camera serving as a "phonetic mirror" and so on. The program is available at our website and may be downloaded free of charge.

2.1 Learning the pronunciation of vowels

It is experimentally known [5] that the training of the utterance of vowels is more difficult than that of consonants because their phonation is not so easy to explain. The key feature of the therapy of the hearing impaired is the refined pronunciation of vowels in order to attain articulate/intelligible speech. It would be a real help in therapy if the computer were able to provide an objective rating of the quality of the uttered vowels. If it were reliable and matched the subjective opinion of the therapist, it would relieve teachers of the burden of the tedious work they have with traditional therapy. In SpeechMaster the role of effective real-time vowel recognition is essential. Real-time visual feedback helps improve the student's articulation because it aids the damaged or missing auditory feedback. The software package provides clear and simple forms of real-time visual feedback. It also has many feedback configurations: it can display the best individual vowel, all the vowels, diagrams and so on. The student can use a web-camera as a "phonetic mirror" to check his/her own articulation or compare his/her utterance with that of the teacher. The student's utterances are stored in separate directories in chronological order for analysis at same later time.



(a) (b) (c)

Figure 2: Screenshots of the learning the vowel pronunciation with SpeechMaster

2.2 Computer-aided therapy in practice

SpeechMaster has several target environments: (nursery) school, therapy and home. In most cases the children and the therapeutist use the recogniser during therapy in the following way: the therapeutist presents a vowel (word) and the child repeats it. The level of acceptance can be varied for each vowel separately and so the therapeutist can maintain the pupil's motivation. At home the child can play back the sample utterance and practice it. The child can work with his parents too, if he or she wishes. These activities require real-time "speaker adaptation, or normalization" techniques or a good user-independent recogniser.

3 The manually clustered vowel database

The variance of the data over the clusters of a database is smaller than that of the full database. Because of this, the classification error over the clusters is usually smaller. This gave us the idea of using a two-step recognition method where the first step identifies the cluster of the data item, and the second step classifies it using the cluster-specific classifier. This way if the error rate of clustering on unseen data is small enough, the final recognition performance can be increased. When we recorded the vowel database for SpeechMaster, we stored utterances separately according to the gender and age of the speakers. This manually clustered database was applied for training vowel recognition in the latest version of SpeechMaster. Table 1 shows the classification accuracies measured on the non-clustered and the clustered databases together with the results for each cluster. It is clear that the performance on the clusters "Men", "Women", "Children" as well as the performance of the two-step method were both better than that of the original one-step method (no pre-clustered). The results below were obtained from a database of 200 speakers [7] (CSCS 2004). We recorded the utterances of healthy hearing children, because wanted to like teach the hearing impaired to speak and not simply to allow them to recognise their vowels. Each speaker uttered 9 clearly formatted and pronounced, sustained, voiced Hungarian vowels. This task is easier than a general phoneme recognition task.

Speakers	Accuracy
Men	90.81 %
Women	91.32 %
Children	96.11 %

Method	Accuracy
1-step (No clusters)	88.32 %
2-step (3 clusters)	91.16 %

Table 1: Recognition accuracy for a manually clustered vowel database

As one can see, the 2-step method outperforms the 1-step classification when the clusters correspond to the manually clustered "Men", "Women", "Children" labels of the database. In the following we shall examine what happens when the clusters are created automatically. The clustering method here uses speaker-space vectors [6]. These vectors contains the mean feature vectors of 9 Hungarian vowels.

4 Clustering methods

Data clustering is a commonly applied technique in statistical data analysis. Clustering is a process where a data set is partitioned into subsets (clusters) so that the data in each subset (ideally) share some common trait - often approximately based on some pre-defined distance measure. Machine learning typically treats data clustering as a form of unsupervised learning. Actually there are two types of data clustering algorithms: hierarchical ones and partitioning ones. Hierarchical algorithms create successive clusters using previously established clusters, while partition algorithms find all clusters simultaneously. Hierarchical algorithms can be agglomerative (bottom-up) or divisive (top-down). Agglomerative algorithms start by considering each element as a separate cluster and successively merge them into larger clusters. Divisive algorithms start with the whole set and proceed to divide it into successively smaller clusters. In our experiments we investigated a partition clustering method and a bottom-up hierarchical clustering method.

4.1 K-Means

K-means clustering is an iterative partitional algorithm [1] the clusters the data points into K disjoint subsets $S_j (j = 1, ..., K)$ by minimizing the sum-of-squares criteria

$$\sum_{j=1}^{K} \sum_{i \in S_j} |x_i - \mu_j|^2,$$

where x_i is a vector representing the i^{th} data point and μ_j is the geometric centroid of the data points in S_j.

The method consists of the following steps:

1. Randomly generate K clusters and determine the cluster centres, or directly generate K seed points as cluster centres.
2. Assign each point to the nearest cluster centre.
3. Recompute the new cluster centres.
4. Repeat until some convergence criterion is met (e.g. the assignment does not change). It is guaranteed to stop, because number of ways the dataset can be partitioned is finite and the algorithm decreases the error criterion in every step.

This algorithm has two significant advantages that allows it to be useable on large datasets, namely its simplicity and speed. Its main disadvantage is that it does not yield the same results with each run, since the resulting clusters depend on the initial random assignments. It maximises inter-cluster (or minimises intra-cluster) variance, but does not guarantee that each result will have a global minimum variance. An improved version of the algorithm is described in [2] which refines

the initial points. We did not try this improved version, because our database was quite small (only 300 speakers). We performed the partition several times with the base algorithm, and then selected the best partitions.

4.2 Unweighted Pair-Group Method with Arithmetic Mean

This is a simple hierarchical agglomerative (bottom-up) algorithm used in bioinformatics to create phylogenetic trees [4]. At each step this iterative algorithm merges the two nearest clusters and recalculates their distances from the remaining ones. The new distances can be calculated using the formula:

$$D_{(ij),k} = (\frac{N_i}{N_i + N_j})D_{i,k} + (\frac{N_j}{N_i + N_j})D_{j,k}$$

where the distance between the i^{th} and j^{th} cluster is $D_{i,j}$ and the i^{th} cluster contains N_i data points.

This method often leads to a degenerate tree (cluster), so we decided to introduce an extra criterion: we fuse the i^{th} and j^{th} clusters if and only if, for a given i and j, N_i or N_j, is less than some given threshold.

5 Classifiers

The classification problem is a supervised learning task. The learner is required to learn (to approximate the behaviour of) a function which chooses, for a sample represented by a feature vector, the right class by looking at several input-output examples of the function.

5.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) is a well-known machine learning method. The basic idea behind ANNs is that many simple functional units (neurons) when combined in parallel produce effective models for learning [1]. A unit receives its input from several other units, or perhaps from an external source. Each input has an associated weight w, which can be modified so as to model synaptic learning. The unit next computes some function f of the weighted sum of its inputs:

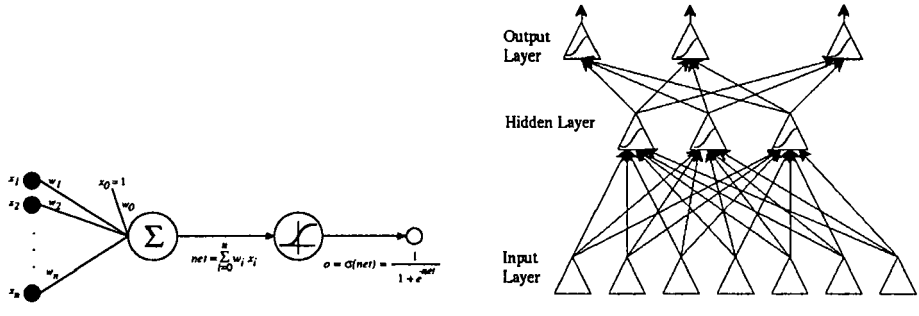
$$net_j = \sum_i w_{ij} y_i$$

$$y_j = f(net_j)$$

The function f is the activation function of the unit. A commonly used activation function is the Sigmoid function:

$$\frac{1}{1 + e^{-net}}$$

The input, output and hidden layer(s) contain many individual units and can model any function. The neurons on each layer are usually fully interconnected with other neurons on an adjacent layer (see Fig. 3). The ANN then learns by modifying the weights in the sigmoid unit. The back-propagation learning rule finds a local, but not necessarily global error minimum [1]. During the classification task the input will be the feature vector. The index belonging to the maximum value of the output vector will be the index returned as the class of the input sample.



(a) A sigmoid unit (b) A three layered network

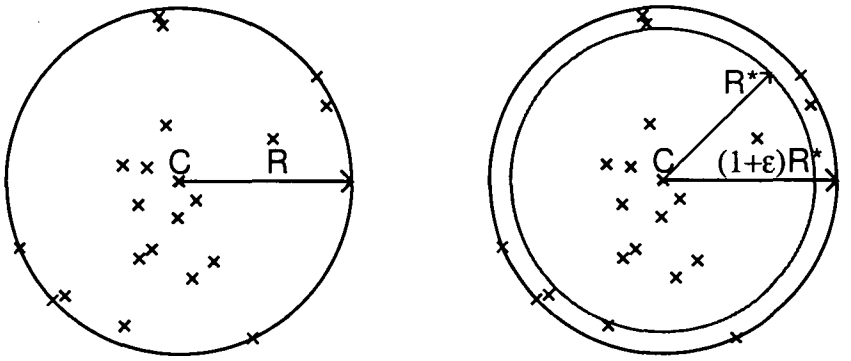
Figure 3: A sigmoid unit and a full ANN

5.2 Core Vector Machine

The Core Vector Machine (CVM) method [9] is a variant of the Support Vector Machine (SVM) [10] approach. The Support Vector Machine performs the following task: it maps the input vectors into a high dimensional feature space through a non-linear mapping. In this space a linear decision surface has high generalization ability. The standard Support Vector Machine training algorithm is of $O(n^3)$ in time complexity and $O(n^2)$ in space complexity, where n is the size of the training database. CVM only approximates the optimal solution via an iterative algorithm, but it has $O(n)$ time complexity and its space complexity is independent of n. The basic aim here is to find, using the notion of core sets, an efficient approximation for the solution of the minimum enclosing ball (MEB) problem (see Fig. 4). This iterative algorithm works by selecting the furthest point outside the current estimated ball until all the points are covered. The CVM technique essentially combines the method of core sets and nonlinear kernels.

6 Experiments and evaluation

Firstly we will describe the corpus and the feature extraction technique, followed by the clustering and classifier algorithms used in the tests. After that we will specify the task of the recognition test, and list the results in tables.



(a) The MEB problem (b) ϵ approximation

Figure 4: Solving the MEB problem by an efficient approximation.

6.1 Conditions

- **Corpus:** For training and testing purposes we recorded samples from 300 speakers, namely 75 women, 75 men, 75 girls and 75 boys. (The ages of the children were between 6 and 9.) The speech signals were recorded and stored at a sampling rate of 22050 Hz in 16-bit quality. Each speaker uttered all the Hungarian vowels, one after the other, separated by a short pause. Since we decided not to discriminate their long and short versions, we only worked with 9 vowels altogether.

- **Feature set:** The signals were processed in 10 ms frames, the log-energies of 24 critical-bands being extracted by using FFT and triangular weighting [8]. The energy of each frame was normalized separately, so only the spectral shape was used for classification.

- **Speaker-space:** The speaker-space database contained the 24 critical-bands of the 9 vowels for each speaker. Hence the dimension of the speaker-space was 9x24.

- **The K-Means clustering method:** We tested it with values of k between 3 to 6. In the evaluation $k = 4$ was chosen because this is the maximum value of k for which the size of the clusters did not become unusably small. The applied distance metric was the Euclidean one.

- **The UPGMA clustering method:** We used the modified UPGMA method with a threshold value of 10. It produced 3 clusters. During experiments we applied the Euclidean distance as a distance metric.

- **The ANN classifier:** We employed the well-known three-layer feed-forward MLP networks trained with the back-propagation learning rule. The number of hidden neurons was 16, which performing some preliminary tests.

- **The CVM classifier:** For the CVM we used the Radial Basis Function

$$k(x,y) = \exp\left(-\frac{\|x-y\|^2}{\gamma}\right)$$

with

$$\gamma = \frac{1}{m^2}\sum_{i,j=1}^{m}\|x_i - x_j\|^2$$

where m is the size of the train set.

6.2 The recognition tests

The experiments were conducted as follows. First we divided the database into train and test sets. The ratio of the data in the train and test sets was 80% to 20%, keeping the ratio of boys, girls, men and women the same in each set.

We clustered the training part of the speaker-space into k blocks using K-Means and UPGMA. Since the speaker-space was not available for the test set (because the features of speaker-space contains all 9 vowel), an ANN (CVM) learner (denoted by M_0) was trained to separate the clusters of the speakers. The training of the the M_0 was then performed on the vowel training database. The speaker clusters defined a clustering of this database at the same time. K separate ANNs (CVMs) learners (denoted $M_{1...k}$) were afterwards trained to classify the vowels within each cluster.

The testing was performed only on a previously unseen vowel database. In the first step the M_0 machine learner chose the proper cluster for each test data (features of a single vowel). The test items were then classified by the corresponding $M_{1<=j<=k}$ learner. In this part classifier combination methods were be used as well.

7 The test results

7.1 Recognition accuracy on the clusters $M_{1...k}$

Table 2 shows that the vowel classification accuracy over the automatically formed clusters "$K_{1..4}$", "$U_{1..3}$" of the data sets turned out to be better than that over the manually clustered "Men", "Women", "Girls", "Boys", "Children" data sets. But the reader should note that this classification accuracy was calculated on the train set, and over-learning may influence the results.

K-Means

	Accuracy	
Cluster	ANN	CVM
K_1	96.76%	100.00%
K_2	98.26%	100.00%
K_3	97.74%	99.83%
K_4	98.64%	100.00%

UPGMA

	Accuracy	
Cluster	ANN	CVM
U_1	97.99%	99.87%
U_2	97.12%	100.00%
U_3	97.35%	99.94%

Manually

	Accuracy	
Cluster	ANN	CVM
Women	90.12%	91.13%
Men	92.84%	92.51%
Girls	99.51%	98.01%
Boys	92.83%	92.54%
Children	96.05%	95.59%

Table 2: Vowel recognition accuracy on the clusters $M_{1..k}$ expressed in percentage terms.

7.2 Clustering accuracy on the train database (M_0)

Table 3 shows the results of the cluster identification test on the train database using the M_0 machine learner. The classification accuracy of the clusters was between 93% and 96%.

		Accuracy	
Method	**#clust.**	**ANN**	**CVM**
Baseline*	1	100.00%	100.00%
Manually	3	94.12%	94.48%
Manually	4	93.32%	94.24%
K-Means	4	94.25%	95.63%
UPGMA	3	93.53%	93.70%

*This corresponds to the original, one-step recognition method

Table 3: Cluster classification accuracy on the train database M_0 (in percent)

7.3 Recognition accuracy on the test database

Table 4 lists the final test results, that is the vowel classification accuracy on the test database. As can be seen, the performance of the two-phase recogniser was better than that of the original one-step method. On the other hand the performance of the ANN and the CVM methods were quite similar. Still, we found that this database was not large enough to show the full advantages of using CVM.

		Accuracy	
Method	**#clust.**	**ANN**	**CVM**
Baseline	1	89.57%	90.58%
Manually	3	92.03%	92.48%
Manually	4	90.97%	91.01%
K-Means	4	92.59%	93.26%
UPGMA	3	91.33%	90.97%

Table 4: Vowel classification accuracy on the test database (in percent)

8 Conclusions and future suggestions

This paper described a computer-aided speech therapy system where, of course, effective real-time vowel recognition is essential. We presented a simple idea for increasing the recognition performance based on our previous experiences that the training part is more efficient when the database is homogeneous in some way. During our study we found that the classification accuracy was higher when we used a database that was separated according to speaker gender and age than when it was not. This suggested the idea of using a two-step recognition method where the data is automatically clustered, and separate classifiers are trained over the clusters. We found experimentally that the classification error over these clusters is actually smaller than that over the full database. In the proposed two-step recognition process the algorithm first identifies the cluster of the data item, and then, in the second step, the item is classified by applying the cluster-specific classifier. We found that with this method the recognition performance improved, so the clustering step can indeed improve the recognition performance. However, the error from the clustering part (namely, that of the learner M_0) during testing seemed to cause a significant loss in performance. Hence, in the future, we plan to do more experiments to find a better method for choosing the right kind of cluster.

9 Acknowledgements

The project described in paper was financially support by the Hungarian Ministry of Education.

References

[1] Bishop, C. M. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[2] Bradley, P. S. and Fayyad, U. M. Refining initial points for K-Means clustering. In *Proc. 15th International Conf. on Machine Learning*, pages 91–99. Morgan Kaufmann, San Francisco, CA, 1998.

[3] Duda, R. O., Hart, P. E., and Stork, D. G. *Pattern Classification.* John Wiley and Son, New York, 2001.

[4] Fitch, W. M. and Margoliash, E. Construction of phylogenetic trees. *Science*, 155:279–284, 1967.

[5] Hégely, G. and Kocsor, A. A vizuális beszédértékelés alkalmazásának magyar vonatkozású történeti áttekintése a hallássérültek beszédoktatásában. *Alkalmazott Nyelvtudomány*, 2005.

[6] Hu, Z., Barnard, E., and Vermeulen, P. Speaker normalization using correlations among classes. In *Image, Speech, Signal Processing and Robotics*, volume II, pages 223–228, The Chinese University of Hong Kong, Hong Kong, 1998.

[7] Paczolay, D., Felföldi, L., and Kocsor, A. Classifier combination schemes in speech impediment therapy systems. Submitted to Periodica Polytechnica Electrical Engineering, 2005.

[8] Rabiner, L. R. and Juang, B. H. *Fundamentals of Speech Recognition.* Englewood Cliffs, NJ, Prentice Hall, 1993.

[9] Tsang, I. W., Kwok, J. T., and Cheung, P. Core vector machines: Fast svm training on very large data sets. *The Journal of Machine Learning Research*, 6:363–392, 2005.

[10] Vapnik, V. N. *Statistical Learning Theory.* John Wiley and Son, 1998.

Acta Cybernetica 18 (2008) 463–478.

Sentence Alignment of Hungarian-English Parallel Corpora Using a Hybrid Algorithm

Krisztina Tóth, Richárd Farkas, and András Kocsor*

Abstract

We present an efficient hybrid method for aligning sentences with their translations in a parallel bilingual corpus. The new algorithm is composed of a length-based and anchor matching method that uses Named Entity recognition. This algorithm combines the speed of length-based models with the accuracy of anchor finding methods. The accuracy of finding cognates for Hungarian-English language pair is extremely low, hence we thought of using a novel approach that includes Named Entity recognition. Due to the well selected anchors it was found to outperform the best two sentence alignment algorithms so far published for the Hungarian-English language pair.

Keywords: sentence segmentation, sentence alignment, length-based alignment, hybrid method, Named Entity recognition, anchor, cognates, dynamic programming

1 Introduction

In the last few years parallel corpora have become evermore important in natural language processing. There are many applications which could benefit from parallel texts like (i) automatic translation programs (as machine learning algorithms) that are used as training databases, (ii) translation support tools that can be obtained from them (translation memories, bilingual dictionaries) and (iii) Cross Language Information Retrieval methods. These applications require a high-quality correspondence of text segments like sentences. Sentence alignment establishes relations between sentences of a bilingual parallel corpus. This relation may not have just a one-to-one correspondence between sentences; there could be a many-to-zero (in the case of insertion or deletion), many-to-one (if there is a contraction or an expansion) or even many-to-many alignments.

Various methods have been proposed to solve the sentence alignment task. These are all derived from two main classes: length-based and lexical methods,

*Research Group on Artificial Intelligence of the Hungarian Academy of Sciences and University of Szeged H-6720 Szeged, Aradi vértanúk tere 1., Hungary, E-mail: {tothkr, rfarkas, kocsor}@inf.u-szeged.hu

but the most successful are combinations of them (hybrid algorithms). *Algorithms using the sentence length* are just based on statistical information given in the parallel text. The common statistical strategies all use the number of characters like Gale & Church's [8] or words like Brown et. al.'s method [1] of sentences which models the relationship between sentences to find the best correspondence. These algorithms are not so accurate if sentences are deleted, inserted or there are many-to-one or many-to-many correspondences between sentences. *Lexical-based methods* [2] [10] utilise the fact that if the words in a sentence pair correspond to each other, then the sentences are also probably translations of each other. Length-based methods align sentences quickly and the alignment is moderately accurate, while the lexical based methods are more accurate but much slower than sentence length-based alignment techniques.

Many applications combine methods which allow the generation of a fast and accurate alignment [4, 13]. These hybrid algorithms utilize various kind of anchors to enhance the quality of the alignment such as numbers, date expressions, various symbols, auxiliary information (like session numbers and the names of speakers in the Hansard corpus[1]) or cognates. Cognates are pairs of tokens of historically related languages with a similar orthography and meaning like parlament/parliament in the case of the English-French language pair. Several methods have also been published to identify cognates. Simard et. al. [20] considered words as cognates, i.e. those that had a correspondence with at least four initial letters, so pairs like government-gouvernement should be excluded. McEnery and Oakes [12] did the calculation of the similarity of two words using Dice's coefficient. These cognate-based methods work well for Indo-European languages, but languages belonging to different families (like Hungarian-English) or with different character sets the number of cognates found is low.

The newest generation of algorithms uses both the length and lexical information but they are based on the Machine Learning paradigm [3, 6]. These approaches requires a great and precise (manually labeled) training corpus which is not present for English-Hungarian at the moment.

Methods have been published for Hungarian-English language pair by Pohl [15] and Varga et. al. [23]. These are also hybrid methods that use a length-based model, but to increase the accuracy Pohl uses an anchor-finding method and the algorithm developed by Varga (called Hunalign) based on a word-translation approach.

In this paper we will introduce an efficient hybrid algorithm for sentence alignment based on sentence length and anchor matching methods that incorporate Named Entity recognition. This algorithm combines the speed of length-based models with the accuracy of the anchor-finding methods. Our algorithm here exploits the fact that Named Entities cannot be ignored from any translation process, so a sentence and its translation equivalent contain the same Named Entities. With Named Entity recognition the problem of cognate low hits for the Hungarian-English language pair can be resolved. To the best of our knowledge this work is

[1] http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC95T20

among approaches for any language pair, the first sentence alignment method that uses Named Entities as anchors.

To handle the problem of sentence alignment an efficient sentence segmentation method and an accurate parallel corpus are needed. We will introduce our expert rule based sentence segmentation and our parallel corpus as well. The recently built corpus contains over 5000 sentences per language and seeks to represent normal everyday language.

In Section 2 the sentence segmentation problem is presented, then Section 3 is devoted to sentence alignment. Section 4 introduces our reference corpus for sentence segmentation for the Hungarian-English language pair along with experiments carried out using our algorithms and several other algorithms. Our results on sentence segmentation and sentence alignment will be discussed here as well. Lastly in Section 5 we provide a short summary and some suggestions for the future.

2 The segmentation problem

The success of sentence alignment depends on the location of sentence boundaries. A common definition of a sentence is: *A sentence is a syntactically autonomous sequence of words, terminated by a sentence-end punctuation.* The term sentence-end punctuation includes full-stops ('.'), exclamation marks ('!') and question marks ('?'), but a sentence ending might be denoted by a colon (':') or semicolon (';'), provided the sentence can stand on its own syntactically (be syntactically autonomous). This definition works well if the text contains sentences in the narrowest sense. But in cases where the input contains structured elements (like tables or enumerations) this definition becomes useless because it requires that a sentence always end with a sentence-end marker. Thus we chose to redefine the meaning of a sentence from our computer linguistic perspective: *A character-stream is regarded as a sentence if it is a sentence in the narrowest sense, a title, an item of an enumeration or a cell in a table.*

Segmenting a text into sentences is a non-trivial task since all end-of-sentence punctuation marks are ambiguous. The most ambiguous sentence-end-punctuation is the full-stop. A full-stop could be a part of a date, denote an ordinal number in Hungarian, an abbreviation, be the end of a sentence, or even an abbreviation at the end of a sentence. The following sentence contains full-stops that have different roles:

```
A Szamos u. 16. alatt található XX. században épült kb. 20 méter magas épületet 2005. 06. 05. és 2006. 06. 05. között az XY. Kft. újította fel.
```

This sentence would probably be segmented into 12 sections by a sentence segmentation application that identifies a sentence boundary after each full-stop. This example and the following statistics demonstrate that the problem of sentence segmentation is worth spending some time on in order to come up with a solution. In the Brown corpus 10% of the full-stops denote abbreviations [7]. According to

[11], 47% of the full-stops in the Wall Street Journal lie inside an abbreviation and in scientific texts it is even more: from 54.7% to 92.8% [14]. Like the full-stop an exclamation mark or a question mark can be inside a sentence e.g. when they occur within quotation marks or parentheses, as in the following sentence:

```
"Látok!" - mondta a vak (aki lehet, hogy nem is vak!?)
```

To handle these problems we used the following rule based system. We collected two lists; *special characters* that are different types of quotations and parentheses, and *potential sentence-end-marks* that are full-stops, exclamation marks, question marks, colons and dots.

The algorithm has three steps:

Step 1 The first step of our segmentation process is the removal of *special characters* from the front and the end of every word.

Step 2 The word ending with a potential-sentence-end-marker (*candidate*) is analyzed: it could be an ordinal number, an abbreviation or a simple word. It has then to decide whether the candidate is an ordinal number. As for whether the word is an abbreviation or a simple word, it checks it against a look-up abbreviation list.

Step 3 The candidate's environment (the word following it) is analyzed:

(a) If there is no subsequent word, the sentence boundary has been identified.

(b) If the candidate is a simple word, and is followed by a word that is a number or begins with a capital letter, we identify a sentence boundary; otherwise there is no boundary.

(c) If the token is an abbreviation we do not segment because the abbreviation might be followed by number (like 'ca. 30'), or an abbreviation (Prof. or Dr.) or a proper name (Dr. Müller).

(d) If the candidate is an ordinal number, and it is the first token in the sentence, we do not identify a sentence boundary (but with this method we can identify rows of tables). If the ordinal number is followed by a number, or a word has a lower case first letter we do not identify a sentence boundary.

(e) A special case of the sentence-end-markers is the colon. In cases after the colon a sentence in the narrowest sense is sought: we identify a sentence boundary after a colon (as in 'Az EU alábbi intézményei a következő feladatokat látják el: Az EU Bíróság bírál.') Otherwise the colon is followed by an enumeration (like 'Halihó Malacka, vegyél nekem: mézet, kenyeret, szalonnát.') then we recognize it as one sentence.

3 The hybrid model

After the sentence boundaries are determined – using the decision process described above – for Hungarian and English we need to perform a sentence alignment in a paragraph.

Figure 1 outlines our model. As input we have two texts, a Hungarian and its translation in English. In the first step the texts will be sentence segmented, and then paragraph aligned. We look for the best possible alignment within each paragraph. For each Hungarian-English sentences we determine the cost of the sentence alignment with the help of dynamic programming. At each step we know

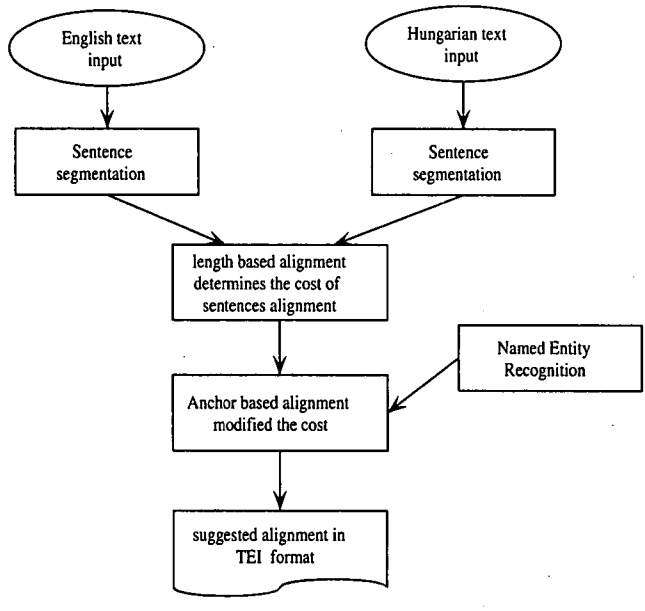


Figure 1: The overview of the alignment system

the cost of the previous alignment path, and the cost of the next step can be calculated via the length-based method and anchors (including Named Entities) – as described later in detail – for each possible alignment originating from the current point (from one-to-one up to three-to-three). The base cost of an alignment is Δ (see Section 3.1), which is increased by punishing for many-to-many alignments. Without this punishment factor the algorithm would easily choose, for example, a two-to-two alignment instead of the correct two consecutive one-to-one alignments. This base cost is then modified by the matched anchors. The normalized form of the numbers, the special characters collected from the current sentences and each matching anchor together reduce the base cost by 10%. The cost is also reduced by 10% if the sentences have the same number of Named Entities.

The problem of finding the path with minimal cost (after the cost of each possible step has been determined) is solved by dynamic programming. The search begins from the first sentences of the two languages and must terminate in the last sentences of each language text. For this we used the well known forward-backward method in dynamic programming.

3.1 Length-based alignment

This module exploits the fact that sentence lengths are correlated. The measure of a sentence length is the number of characters in a sentence, just like that in the Gale and Church [8] algorithm.

We will assume that the ratio, between the length of the source sentence and target sentence, has a normal distribution (independent and identically distributed from sentence to sentence). The mean and standard deviation of each can be calculated from our new Hungarian-English parallel corpus (introduced in Section 4.2.1): $E(l_1/l_2) \approx 1.1$ and $V(l_1/l_2) \approx 7.9$, where l_1 is the number of characters in a Hungarian sentence and l_2 is the number of characters in its translation.

Just like [8] we define δ to be $(l_2 - l_1 E(l_1/l_2)) / \sqrt{l_1 V(l_1/l_2)}$ so that it has a normal distribution with zero mean and a variance of one (at least when the two sentences in question are actually the translations equivalents of each other). The base cost of the alignment (for two sentences with length l_1 and l_2) respectively will be $\Delta = -logP(match|\delta(l_1, l_2))$. The log has been introduced here so that adding costs will produce desirable results.

3.2 Anchors

The published approaches for a Hungarian-English language pair judged the words containing capital letters or digits of equal amount in the text to be the most trusted anchors, but any mistakenly assigned anchors have to be filtered. Unlike other algorithms our novel method needs no filtering of anchors because the alignment works with the help of exact anchors like Named Entities. The following example illustrates the difference between using capitalized words as anchors against using Named Entities as anchors:

```
Az új európai dinamizmus és a változó geopolitikai helyzet arra
késztetett három országot, név szerint Olaszországot, Hollandiát
és Svédországot, hogy 1995. január 1-jén csatlakozzon az Európai
Unióhoz.

The new European dynamism and the continent's changing geopolitics
led three more countries - Italy, Netherlands and Sweden - to join
the EU on 1 January 1995.
```

In the Hungarian sentence there are 5 capitalized words (Olaszországot, Hollandiát, Svédországot, Európai, Unióhoz), unlike its English equivalent which contains 7 ones (European, Italy, Netherlands and Sweden, EU, January) so using this feature as an anchor would give false results, but an accurate Named Entity recognizer could help it. This example demonstrates as well that cognates cannot be used for a Hungarian-English language pair.

Thus we suggest modifying the base cost of a sentence alignment with the help of the following anchors: special characters, the normalized form of the numbers and Named Entity recognition instead of a bilingual dictionary of anchor words or the

number of capital letters in the sentences. These result in a text-genre independent anchor method that does not require any anchor filtering at all.

3.2.1 Named Entities

Instead of using capitalized words present in the sentences we use the Named Entity Recognition module. It is used because in English more words are written with a capital letter than their Hungarian equivalents. Some examples from the Hungarian-English parallel corpus indeed demonstrate this fact:

- I (én) personal pronoun
- Nationality names: ír söröző = Irish pub
- Location terms: Kossuth Street/Road/Park
- When repeating an expression, the expressions become shorter: pl: European Union = Unió
- Names of countries: Soviet Union = Szovjetunió
- The names of months and days are written with capital letters.

The identification and classification of proper nouns in a plain text is of key importance in numerous natural language processing applications. It is useful in sentence alignment because Named Entities cannot be ignored in any translation process, so a sentence and its translation equivalent contains the same number (and types) of Named Entities. As far as we know our work is the first sentence alignment method for a language pair that uses Named Entities as anchors.

A slightly modified version of the multilingual Named Entity recognition system described in [22] was used here in this work. This system (which appears to be currently the only statistical Named Entity recognition for Hungarian) achieved an accuracy[2] of over 98.7% on unknown documents in Hungarian (Szeged NE corpus [21]) and 97% for documents in English (CoNNL 2003 shared task [18]). The main aim of [22] was to recognize Named Entities and place them into one of four classes (person, organization, location and miscellaneous). The accessible tagged datasets concentrated on the business domain, unlike our parallel texts which dealt with a wide range of domains. Because of the lack of a suitable training corpora we chose an easier problem, namely recognizing Named Entity phrases (a multiword chain) without classification.
Our statistical approach worked as follows:

1. It extracts features from a tagged train corpora. We collected various types of numerically uncodable information describing each term and its surroundings. A subset of the features used tried to capture the orthographical regularities of proper nouns like capitalization, inner-word punctuation and so on. Another

[2]considering the two class (named entity/non named entity) phrase level evaluation metric

set of attributes described the role of the word and its neighboring words in the sentence. The remaining parameters were various lists of trigger words and ratios of capitalized and lowercase words in large corpora.

2. In this way the problem could be treated as a supervised (more precisely, a two-class classification) task. The C4.5 decision tree [16] with pruning and AdaBoost [19] after 30 iterations was trained on Hungarian and English texts. Different models were learned for the two languages but they were based on the same feature set.

3. The learned models tagged the Named Entity phrases in the input parallel texts. Because the correct tagging was not known we could not measure the accuracy of this tagging, but the experiments described in Section 4.2 revealed that it was definitely helpful.

3.2.2 Special characters

We used the special characters in the sentences like %, §, $, @, & as anchors, because they may be present in the source language and a target language sentence in the same form. Other special characters are used as anchors in the literature as well (like quotation, exclamation mark, question mark [9]), but they were not included because they can confuse the program in the Hungarian-English alignment task.

Take the following examples:

```
angol = I wish I had a bike.
magyar = Bárcsak lenne egy biciklim!
```

The Hungarian exclamation mark is usually (90% in our Hungarian-English parallel corpora) replaced by a full-stop in its English counterpart. As the next example shows, there are differences in the usage of the apostrophes and quotations in the English and Hungarian sentences. In most cases Hungarian quotations have an apostrophe equivalent in an English sentence, and vice versa.

```
"Tibi!" - mondtam az uramnak.

'Tibi!'  - I said to my husband.
```

3.2.3 Normalized form of numbers

Our efficient sentence alignment method treats the normalized form of Arabic or Roman digits. During the normalisation of the digits all characters that are not digits are deleted then we get a digit in a normalized form. With this method we got a language independent form (1.2 = 1,2) that can be compared during the alignment process.

4 Experiments

In this section our results on sentence segmentation and on sentence alignment are presented, and the reference algorithms and corpora are given.

4.1 Sentence segmentation

Here the reference corpora for Hungarian and English and the baseline algorithms are introduced for evaluating our expert rule-based sentence segmentation approach.

4.1.1 Reference corpora

The test corpus for Hungarian sentence segmentation was compiled from sentences that came from three subcorpora of the Szeged Treebank[3] [5] (Népszabadság, Népszava and Heti Világgazdaság). The first two subcorpora were chosen here because a sentence segmentation algorithm will be used on general texts and these truly mirror everyday language. The third is written in business language, and can be used for testing our algorithm on a harder text genre.

The English sentence track evaluation was carried out on the Wall Street Journal Corpus[4], which contains articles from the Wall Street Journal, and consists of 5000 randomly selected sentences. We choose this corpus because we wanted to test our algorithm on articles similar to Hungarian ones, and it was written in normal everyday language.

4.1.2 Baseline algorithms

We compared our algorithm against two algorithms. The first was the baseline algorithm that labels each punctuation mark as a sentence boundary. The description of the second one – called Huntoken – has not yet been published, but some of its results has been used in our three subcorpora[17].

4.1.3 Results

To assess the quality of sentence segmentation precision, recall and F-measure scores of correct segmentations were used.

Tables 1 below list the results of the segmentation on the three subcorpora compared with the results of the first baseline algorithm. We could not compare our results in such detail with Huntoken because only the precision scores have been published so far.

Our expert rule-based algorithm performs significantly better on all subcorpora than the baseline algorithm. These experiments highlight the effect of meaning

[3]The Szeged Treebank is a manually annotated natural language corpus. This is the largest manually processed Hungarian database that can be used as a reference material for research in natural language processing

[4]http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2000T43

Algorithm	Correct	Incorr.	All f.	Etalon	prec.	rec.	$F_{\beta=1}$
Baseline	6450	697	7147	7797	0.9009	0.8216	0.8593
Expert Rule b.	7781	17	7798	7797	0.9980	0.9981	0.9980

Table 1: The three subcorpora together

differences of potential punctuation marks as well. The baseline algorithm achieved poor precision and recall scores on the Heti Világgazdaság corpus (which contains economic texts) but our algorithm gave much the same results as those for the two other corpora. This is probably due to the higher frequency of abbreviations, ambiguous sentence boundaries and special punctuation marks.

Table 2 lists the precision scores of the three algorithms. Our results turned out to be similar to those published in [17]. The difference can be said to be significant only on the economy texts (a 66% error reduction)

Algorithm	Népszabadság	Népszava	Heti Világgazdaság	all
Baseline	0.9133	0.9113	0.8780	0.9009
Huntoken	0.9976	0.9977	0.9937	0.9963
Expert Rule based	0.9976	0.9985	0.9979	0.9980

Table 2: Precision of the three sentence segmentation algorithm

To evaluate the English text, the first baseline algorithm were used. With the English test corpora the baseline method performed very badly, but our algorithm kept the error rate below 1% (see Table 3). The reason for this is that in this corpus there were a lot of parentheses and quotation marks in the words, and there were also quite a few abbreviation and ordinal numbers.

Algorithm	Correct	Incorr.	All f.	Etalon	prec.	rec.	$F_{\beta=1}$
Baseline	3152	3257	6409	5021	0.4918	0.6278	0.5515
Expert Rule b.	4972	38	5010	5021	0.9924	0.9902	0.9913

Table 3: English results

These results demonstrate that our effective sentence segmentation algorithm generates errors of 1% or less on both Hungarian and English texts. This achievment means that our approach is competitive with the best published results for Hungarian and English to date.

4.2 Sentence alignment

Soon we will discuss the results of experiments on our alignment algorithms. But first we need to elaborate on the built corpora and two baseline algorithms from Hungarian literature.

4.2.1 The corpora for sentence alignment

Parallel corpus Currently, there are two sentence-level-aligned Hungarian-English parallel corpora at our disposal. One of them is the so-called Orwell corpus[5] that is based on Georg Orwell's novel 1984 and the other one is a Hunglish corpus [6]. These corpora often contain special words, phrases and jargon, that is why we decided to build our own corpus.

With high quality translation and representability in mind, in the course of Hungarian-English parallel corpus building the following texts were collected:

- **Language book sentences**: This subcorpus includes detached parallel sentences from Dévainé Angeli Mariann's *Angol nyelvtani gyakorlatok* and Dohár Péter's *Kis angol nyelvtan.* These books were compiled for students preparing for a language exam and therefore their wording is not very realistic. There are sentences which truly represent present-day English but, at the same time, there are some overly artificial, 'fabricated' sentences too. These books were written to represent the characteristics of English and not present-day parlance. This subcorpus currently contains over 5000 sentences.

- **Texts on the EU**: These texts were gathered from an official EU website http://europa.eu.int. Under the title *Europe in 12 lessons* there are 13 general descriptions about the EU. This subcorpus is a general Hungarian-English text collection.

- **Bilingual magazines**: This subcorpus is comprised of articles taken from the magazines of Malév Horizon and Máv Intercity.

- **Speech corpus of the Multext-East**: The Multext-East corpus consists of 40 items of 5-sentence long units. The 5 sentences of a unit are correlated and they are available in written form in both Hungarian and English. Text units include topics written in everyday parlance, tell one how to order a taxi, find a restaurant, or call a customer service end so on.

Named Entity training corpora. To train our model on Hungarian texts, we used a a sub-corpus of the Szeged Treebank [21] where the correct classification of Named Entities had also been added[7]. It contains business news articles taken from 38 NewsML topics (9600 sentences) ranging from acquisitions to stock market changes or the opening of new industrial plants.

The Named Entity system for English was trained on a sub-corpus of the Reuters Corpus, consisting of newswire articles from 1996 provided by Reuters Inc. (–the shared task of the CoNLL 2003 Named Entity challenge). It contains texts from domains ranging from sports news to politics and the economy.

[5] http://nl.ijs.si/ME/CD/docs/1984.html
[6] http://mokk.bme.hu/resources/hunglishcorpus
[7] Both Hungarian and English datasets can be downloaded free of charge for research purposes.

4.2.2 Reference alignment methods

Hunglish, translation- and length-based alignment In the first step the algorithm loads the English-Hungarian dictionary that was based on a unified version of the Vonyó and Hókötő dictionaries[8]. The first step of the aligning algorithm provides a rough translation of the Hungarian sentence by substituting each word with its most frequently occurring dictionary translation or, when absent, with the word itself. Then this rough translation is compared, sentence by sentence, with the actual target text.

The similarity rate between sentences is found by looking at the number of mutual occurrences (the very frequent words having been removed from both the raw translation and the original English text) and the sentence length which is measured in characters, but the algorithm also specifically recognizes numbers written in numerical form. The task is then solved with the help of dynamic programming methods [23].

Length- and anchor matching-based alignment of Pohl The other sentence-synchronizing algorithm was worked out by Pohl [15]. The implemented algorithm was built on Gale & Church's sentence-length alignment, and it also included dynamic programming techniques to determine the sentences to be aligned. The only real difference from the original algorithm was that it had to take into consideration the cost of anchor-synchronisation when calculating the overall costs. When running it uses a heuristic method to calculate the gain, which helps it to recognize sentence insertions and deletions in the text. The gain is defined here as follows. It is the number of common anchors in text units divided by the total number of anchors in the text units, then this fraction is divided by the number of text units involved. Pohl regards on the other hand the number of words containing numbers or capital letters as the most reliable anchors. He employed the method published by Ribeiro et. al. to filter out the mistaken anchors. It defines two statistical filters, both of which apply a linear regression margin calculated on the basis of the anchor-candidate's position in the text. In the first step the points outside a certain range – determined using an adaptive histogram-based filter applied around the linear regression margin – were disregarded, then the points outside the confidence bracket of the regression margin were found.

4.2.3 Results

Our hybrid algorithm was compared with Pohl's length- and anchor matching-based one and with the Hunglish's dictionary- and sentence length-based hybrid ones. Pohl's algorithm also had to be reimplemented. The comparison was not complete, but it used just one-to-one, two-to-one and one-to-two alignment types.

The first row shows what kind of alignments are possible in the reference alignment, like one-to-one, one-to-two or many-to-many. There is no one-to-zero alignment in our parallel corpus even through there could be. The second row shows

[8] http://almos.vein.hu/ vonyoa/SZOTAR.HTM

	1:1	1:2&2:1	2:2	N:M
suggested alignment	4875	415	0	0
correct of sugg. align.	4556	165	0	0

Table 4: Pohl's Results

how many of these alignment types were found by Pohl's algorithm, and the last row shows how many of the suggested alignments were correct. Table 5 gives the corresponding results for our algorithm, which, as the reader will notice, are not so different.

	1:1	1:2&2:1	2:2	N:M
suggested alignment	4957	339	3	1
correct of sugg. align.	4698	252	1	0

Table 5: Our Results

The algorithm of Pohl's chose one-to-two and two-to-one alignments with a poor precision (just 39%). Our hybrid algorithm on the other hand was more accurate in these cases and it even handles two-to-two and n:m alignments as well. In the one-to-one alignment task they achieved similar results. Our algorithm was better here as well, but this is probably only due to Pohl choosing too many one-to-two alignments instead of more one-to-one alignments.

Table 6 summaries the results of the three hybrid methods. Precision and recall are the commonly accepted metrics for evaluating the quality of a suggested alignment with respect to a test corpus. We employ the F-measure here as well, which combines these metrics into a single efficiency measure:

$$precision = \frac{number\ of\ correct\ alignments}{number\ of\ proposed\ alignments}$$

$$recall = \frac{number\ of\ correct\ alignments}{number\ of\ reference\ alignments}$$

$$F_{\beta=1} = 2\ \frac{recall * precision}{recall + precision}$$

Algorithm	Precision	Recall	$F_{\beta=1}$
Pohl hybrid	0.9016	0.9016	0.9016
Hunalign	0.8993	0.9786	0.9370
NE-based	0.9341	0.9456	0.9398

Table 6: Results of Hungarian hybrid methods

The high recall of the hybrid dictionary-based method is largely due to the dictionary (it offers a huge number of one-to-one alignments), but it did not attain

a 90% precision score. Contrary to our algorithm, it has a recall and precision of over 90 % thanks to the good choice of anchors.

After a manual analysis, we found that the bigger part of the errors come from paragraphs where there are not any anchor (neither Named Entities, numbers nor punctuation) in the sentences. On the other hand the recognition of Named Entities is far from perfect, its error is propagated to the alignment. If a larger and more general Named Entity training corpus will be available a more accurate recogniser model could be trained and different types of entities could be used which could further improve our results.

Viewed overall, our new hybrid algorithm is approximately 4% better than the approach which inspired our study (Pohl's anchor matching based algorithm) and it achieved slightly better results than those for Hunalign. The real advantage over Hunalign is its speed of alignment. We used a very fast (in alignment time) Named Entity recognizer that did not need to search through a huge database dictionary.

5 Conclusions and future work

In this paper we introduced a language independent, expert rule-based sentence segmentation method (which we found has a typical error rate of $< 1\%$), a Hungarian-English parallel corpus containing everyday language – which was designed for machine translation – and a novel Named Entity-based hybrid sentence alignment method (the first step of machine translation) that combines accuracy (a roughly 6% error rate) with speed.

The results of the previous section demonstrate that our system is competitive with other sentence alignment methods published for the Hungarian-English language pair. The reason for our good results is that, with the help of Named Entity recognition, more anchors can be matched so the problem of low hits of the cognate pairs for a Hungarian-English language pair is effectively solved. The use of multilingual Named Entity recognition systems also provides a way of finding appropriate anchors for language pairs even when they belong to distinct language families.

In the future it would be useful to build and to learn on a Named Entity corpus that incorporates everyday language. Then the advantage of using a Named Entity classifier would probably become apparent and it should improve the precision of Named Entity anchors. In addition, we would like to test our system on diverse text sources to see how well it performs.

References

[1] Brown, P. F., Lai, J. C., and Mercer, R. L. Aligning sentences in parallel corpora. In *Meeting of the Association for Computational Linguistics*, pages 169–176, 1991.

[2] Chen, S. F. Aligning sentences in bilingual corpora using lexical information. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, pages 9 – 16. Columbus, Ohio, 1993.

[3] Chuang, Thomas C. and Chang, Jason S. Adaptive bilingual sentence alignment. In *AMTA '02: Proceedings of the 5th Conference of the Association for Machine Translation in the Americas on Machine Translation: From Research to Real Users*, pages 21–30, London, UK, 2002. Springer-Verlag.

[4] Collier, N., Ono, K., and Hirakawa, H. An experiment in hybrid dictionary and statistical sentence alignment. In *COLING-ACL*, pages 268–274, 1998.

[5] Csendes, D., Csirik, J., and Gyimóthy, T. The szeged corpus: A pos tagged and syntactically annotated hungarian natural language corpus. In *Proceedings of the 7th International Conference on Text, Speech and Dialogue (TSD 2004)*, pages 41–47, 2004.

[6] Fattah, Mohamed Abdel, Ren, Fuji, and Kuroiwa, Shingo. Probabilistic neural network based english-arabic sentence alignment. In *CICLing*, pages 97–100, 2006.

[7] Francis, W. and Kucera, H. *Frequency analysis of English usage: Lexicon and grammar*. Houghton Mifflin, Boston, 1982.

[8] Gale, W. A. and Church, K. W. A program for aligning sentences in bilingual corpora. In *Meeting of the Association for Computational Linguistics*, pages 177–184, 1991.

[9] Hofland, K. and Johansson, S. The translation corpus aligner: A program for automatic alignment of parallel texts. In Johansson, S. and Oksefjell, S., editors, *Corpora and Cross-linguistic Research: Theory, Method, and Case Studies*, pages 87–100. Amsterdam: Rodopi, 1998.

[10] Kay, M. and Röscheisen, M. Text-translation alignment. volume 19, pages 121–142, 1993.

[11] Liberman, M. Y. and Church, K. W. Text analysis and word pronunciation in text-to-speech synthesis. In *Sadaoki Furui and Man Mohan Sondhi, editors, Advances in Speech Signal Processing*, pages 791–831. Marcel Dekker, Inc., 1992.

[12] McEnery, A. M. and Oakes, M. P. Cognate extraction in the crater project. In *S. Armstrong-Warwick and E. Tzoukerman (eds.), Proceedings of the EACL-SIGDAT Workshop (Dublin)*, pages 77 – 86, 1995.

[13] Moore, R. C. Fast and accurate sentence alignment of bilingual corpora. In *AMTA '02: Proceedings of the 5th Conference of the Association for Machine Translation in the Americas on Machine Translation: From Research to Real Users*, pages 135–144, London, UK, 2002. Springer-Verlag.

[14] Muller, H., Amerl, V., and Natalis, G. *Worterkennungsverfahren als Grundlage einer Universalmethode zur automatischen Segmentierung von Texten in Sätze. Ein Verfahren zur maschinellen Satzgrenzenbestimmung im Englischen. Sprache und Datenverarbeitung, 1.* 1980.

[15] Pohl, G. Szövegszinkronizációs módszerek, hibrid bekezdés- és mondatszinkronizációs megoldás. In *Proceedings of Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2003)*, pages 254–259, 2003.

[16] Quinlan, J. R. *C4.5: Programs for machine learning.* Morgan Kaufmann, 1993.

[17] és Kommunikáció Tanszék Média Oktató és Kutató Központ, BME Szociológia. Hunglish cd-rom, http://szotar.mokk.bme.hu/hunglish/search/corpus. 2006.

[18] Sang, E. F. Tjong Kim and Meulder, F. De. Introduction to the conll-2003 shared task: Language-independent named entity recognition. In Daelemans, Walter and Osborne, Miles, editors, *Proceedings of CoNLL-2003*, pages 142–147. Edmonton, Canada, 2003.

[19] Schapire, Robert E. *The Strength of Weak Learnability*, volume 5. 1990.

[20] Simard, M., Foster, G., and Isabelle, P. Using cognates to align sentences in bilingual corpora. In *Proceedings of the Fourth International Conference on Theoretical and Methodological Issues in Machine translation (TMI92), (Montreal)*, pages 67–81, 1992.

[21] Szarvas, Gy., Farkas, R., Felfoldi, L., Kocsor, A., and Csirik, J. A highly accurate named entity corpus for hungarian. In *Proceedings of LREC2006*, 2006.

[22] Szarvas, Gy., Farkas, R., and Kocsor, A. *A Multilingual Named Entity Recognition System Using Boosting and C4.5 Decision Tree Learning Algorithms.* Springer-Verlag, 2006.

[23] Varga, D., Halacsy, P., Kornai, A., Nagy, V., Nemeth, L., and Tron, V. Parallel corpora for medium density languages. pages 590 – 596. Borovets, Bulgaria, 2005.

Regular Papers

Acta Cybernetica 18 (2008) 481–501.

A New Concept of Effective Regression Test Generation in a C++ Specific Environment

Mihály Biczó*, Krisztián Pócza*, István Forgács† and Zoltán Porkoláb*

Abstract

During regression testing test cases from an existing test suite are run against a modified version of a program in order to assure that the underlying modifications do not cause any side effects that would demolish the integrity and consistency of the system. Since the ultimate goal of a regression test set is to effectively test all modifications and reveal errors in the earliest possible stage, the maintenance of a relevant test set containing effective test cases is of utmost importance. In this paper we present an efficient, C++ specific framework to automatically manage the regression test suite. Our two main contributions are a new interpretation of reliable test cases and a dynamic forward impact analyzer method that eases the transformation of existing tests to meet the definition of reliability. Using this approach we complement the test set with test cases that pass through a modification and have an impact on at least one output. Our approach is designed to be applicable to large-scale applications.

Keywords: regression testing, dynamic impact analysis, software maintenance, C++

1 Introduction

Regression testing is an important tool of software engineers to successfully manage issues rising during the evolution of software systems. During the lifetime of large systems numerous modifications are performed over possibly many years, yet it is of vital importance that none of these modifications is allowed to remain untested, or cause unwanted and undiscovered side effects to other previously tested parts.

In order to achieve this goal, a regression test set that covers the whole system has to be maintained and adjusted according to the modifications performed. Therefore, it is desirable to find a test selection method that selects those and only

*Eötvös Loránd University, Fac. of Informatics, Dept. of Prog. Languages and Compilers, Pázmány Péter sétány 1/C. H-1117, Budapest, Hungary E-mail: mihaly.biczo@t-online.hu, kpocza@kpocza.net, gsd@elte.hu

†4D SOFT Ltd. Telepy u. 24. H-1212, Budapest, Hungary E-mail: forgacs@4dsoft.hu

those test cases that might reveal an error [6]. However, it is equally important to re-use and transform existing test cases so that the coverage of the modified system would not be affected.

An important subset of regression tests contains *modification revealing tests* for which the original and modified programs give different output. All modification revealing tests are *modification traversing*, they reach at least one modified statement. Consequently, the set of modification traversing tests also contains all error revealing test cases [16]. Unfortunately, the reverse case is not true: a modification traversing test is not necessarily modification revealing. Existing methods consider a test case successful if the outputs of the original and modified programs are identical. As it can be seen easily, using this approach it is not assured that the modification is really tested. In other words, the test case is not necessarily reliable.

In this paper we alter the existing definition of reliability: the definition of a reliable test pair will be established. According to this definition, we develop an approach that eases the generation of reliable test pairs in a C++ specific environment. We will not cover test data generation techniques, related work can be found in [2, 3, 9, 10]. Instead, we identify those input variables from the whole state space on which the new generation process can be started. For the generation process, the method described in [17] can be used initiated on a reduced input variable set.

Our main contribution is a simple forward dynamic impact analyzer algorithm which, if there is a given modification, will efficiently select the set of influencing input variables and help boost the performance of the test pair generation process. As opposed to existing methods [14], instead of directly comparing the output of the original and modified programs for a given test case, the modified program and the underlying test case are considered. The test suite will be extended with a reliable test pair that is derived from the original test. This test pair will assure that the modification is tested and that some output statements are affected in the modified version of the program. An additional benefit of our approach is that it is designed to work for real C++ based systems, since many C++ specific constructs are covered including pointers and function pointers, as well as object-oriented constructs and paradigms like classes, inheritance and polymorphism.

The structure of the paper is the following: Section 2 defines the problem we are going to solve and presents the general overview of the generator framework through simple examples. We also give an insight into test categorization methods we are going to employ.

In Section 3 we discuss some related work and research directions we are aware of. We will primarily focus on the motivating ideas behind existing techniques.

In Section 4 an overview of the used notations and necessary language specific instrumentation mechanism will be described in detail.

In Section 5 and Section 6 the two stages of the dynamic forward input analyzer algorithm that detects affecting input variables will be discussed. While the first stage of the algorithm categorizes test cases and identifies affected statements; the second stage selects the underlying input variables based on the results of the

first phase.

In Section 7 a full example of our approach will be presented in C++.

In Section 8 we summarize our results and discuss the limitations of the approach as well as some possible research directions.

2 Framework overview

2.1 The necessity of a concept change

As we have mentioned in the introductory section, traditional regression testing approaches categorize test cases based on the outcome of the test case run against the original and the modified programs. However, numerous anomalies might prevent this comparison from being a good filter of errors.

The fundamental issue is that if a modification traversing test gives identical output for the original and modified programs, this does not mean that any of the modifications have really been tested. This is the case when the given modification does not affect any output statements. The reverse case - when the outcome of the original and modified programs differs - can also be problematic, because the test might not be modification traversing for a given modification. As a consequence, if there are more than one modifications (which is typically the case), classical modification revealing tests might not be effective, and once again untested modifications might lurk in the source code. A further example for different output is when the mistakenly modified statement is a predicate, and the test takes another execution branch, although it should go along the original path.

Listing 1 Three versions of a simple program

```
int main()
{
  double a,b,c, d;

  cin >> a;
  b=2;
  c=3;

  d=a+c;

  if(a>0)
    cout << b << endl;
  else
    cout << c << endl;

  //Use d...

  exit(0);
}
```

```
int main()
{
  double a,b,c, d;

  cin >> a;
  b=2;
  c=3;

  //Mod. #1: d=a+c;
  d=a-c;

  if(a>0)
    cout << b << endl;
  else
    cout << c << endl;

  //Use d...

  exit(0);
}
```

```
int main()
{
  double a,b,c, d;

  cin >> a;
  //Mod. #1: b=2;
  b=3;
  c=3;

  d=a+c;

  if(a>0)
    cout << b << endl;
  else //Mod. #2
    cout << c+2 << endl;

  //Use d...

  exit(0);
}
```

Let's consider the three different versions of a simple program in Listing 1. If the input of the program (the test case) is a=1, then the outcome of the original program is that 2 is printed on the screen. The second version still prints 2 for a=1, which is a modification traversing test case, and is successful even though no

modifications have been tested. In the third version there are two modifications. Although for a=1 the outcome of the original and modified programs differs, the test case is still not modification traversing for Modification #2. Although these simple examples show only two possible anomalies, theoretically, there are four of them: if the test case does not traverse any modifications, the output cannot be affected (S0). The other three types of the same output symptom are *coincidental correctness* (S1); *predicate-only symptom*, e.g. the modification influences (either directly or indirectly) only a predicate (S2); or the *modified statement does not affect any output* (S3).

2.2 The changed concept

In order to overcome the above mentioned shortcomings, we have to introduce a new regression testing concept and criterion. Our goal is to test each modification in such a way that - if possible - after the test traverses the location of the modification at least one output statement would be affected. Of course the original test suite might not contain tests that meet this criterion, so it is desirable to establish a method that transforms all possibly usable regression test cases. This way, errors can be revealed with a much higher probability and in an earlier stage.

In order to detect a faulty modification, the underlying test case has to

1. reach the fault (it has to be modification traversing with respect to the faulty modification)
2. the inner state of the program has to be erroneous (the behavior of the program has to differ from the expected)
3. the fault has to reach an output statement resulting in a failure (after the traversal through the erroneous statement, an output statement should be reached)

Common methods consider a test case successful if the outputs of the original and modified programs are identical. The biggest concept change is that we fulfill these requirements using a pair of test cases derived from the original test case instead of just one test and these tests should affect output statements.

2.3 The test generator framework

We build our framework around the above set of criteria. We have had a strong cooperation with an industrial partner, and the framework we present in this paper is part of their project.

In order to fulfill the first requirement, modification traversing test cases have to be selected for a given (possibly erroneous) modification. Different techniques can be found in [5, 7, 8]. Identifying modification traversing test cases requires two steps:

1. The modification needs to be detected
2. Appropriate test cases have to be identified

Of course in the case of real systems extending over possibly millions of lines of code, it is far from being trivial to identify each modification in source code. Our industrial partner has a static analyzer solution that identifies modifications within due time for millions of lines of code. Although the algorithm and the implementation is part of a commercial application (which means that it is copyrighted and cannot be published), for publicly accessible implementation the Columbus framework [11] could be used.

In order to fulfill the second requirement, we establish the following definition of reliable test cases:

Definition (Reliable test pair). Let GI=$\{I_1, I_2, \ldots I_M\}$ the set of input variables, $I \subseteq \{1, \ldots M\}$, I=$\{i_1, i_2, \ldots i_n\}$, $J \subseteq \{1, \ldots M\}$, J=$\{j_1, \ldots j_k\}$ index sets. Consider the following test cases: t_1:=<i_{i1}, i_{i2}, ... i_{in}>, t_2:=<i_{j1},i_{j2},...i_{jk}>, where i_{i1}, i_{i2},... i_{in}, i_{j1}, i_{j2}, ... i_{jk} are the values of the corresponding input variables. A pair (t_1, t_2) of test cases is a reliable test pair with respect to statement s^q (where s^q represents the q^{th} execution of statement s) if t_1 and t_2 travels along the same execution path until s^q, the result of s^q differs for t_1 and t_2, and t_1 and t_2 are 'close' to each other (for numeric values the difference should be minimized according to some metric).

Informally, the above definition states that a test case is reliable with respect to a given modification if and only if the two test cases generate the same execution path as far as s^q, both of them have an influence on at least one output statement, and the result of the output statements differ for the two test cases. Besides this, their difference should be minimized according to the following rule: the number of common variables in set I and J should be minimized, and for the common variables, the difference between them should be minimized according to some metric. For the Eucledian metric, this would be

$$\sqrt{\sum_{\gamma \in I \cap J} i_{i_\gamma} - i_{j_\gamma}}$$

As for the third requirement, we will assume that all test cases in this case reach a modification. Some of them will have an influence on the output, some of them will not. However, in both cases it is highly desirable to transform them to a test pair that meets the definition of reliability.

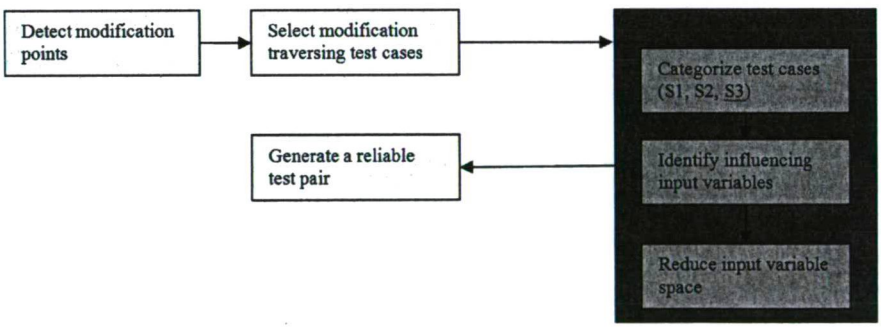
For the effective generation of test cases that meet the definition of our regression testing criterion, we need a reduced set of input variables. This is the main task we solve in this paper: according to the altered definition of reliable test cases (reliable test pair), we are to reduce the set of input variables to so-called influencing input variables. These are input variables that can be used to generate reliable test pairs based on a modification traversing test case.

Our suggested solution for finding influencing input variables is a two stage process. In the first stage the symptom (the anomaly, previously categorized as

S0-S3) is determined. In the second phase the set of influencing input variables are identified which can be used to turn the underlying test case to a reliable test pair. Both stages of the algorithm are based on *forward dynamic impact analysis*. Consequently, there is no need for large data structures in memory, and all results can be obtained on-the-fly. The high-level structure of the framework is the following:

1. Identify modifications
2. Select modification traversing test cases from the original test suite
3. For each selected test case
   a) Identify symptom (S1, S2, S3)
   b) Identify reduced set of influencing input variables
4. Generate a test pair in the reduced variable space using influencing input variables

Our main contributions are 3a and 3b. The schematic structure can be seen in the following figure (Our contributions are in the dark rectangle).



3 Related work, research directions

In this section we present related work that motivated our research. Paper [18] deals with the empirical comparison of test selection techniques. Besides the commonly used but rather desperate random and retest all techniques, minimization, dataflow and safe test selection families are also covered in that article. Our suggested approach has common properties with dataflow techniques that require that every definition-use pair that is deleted, changed, or inserted into the changed program should be tested. In [19] Harrold and Soffa select test cases that exercise the definition-use pairs affected by the modification. Our approach is quite similar with the important remark that we employ test pairs that are safer in case of predicates

and require not only the testing of the modification, but also the employment of at least one output statement.

The two most relevant papers that motivated our research are [15] and [16]. The first paper deals with slicing algorithms [4] that do not use traditional data structures, only dependence analysis to calculate program slices. The second paper categorizes regression test cases based on their effect on the program output. We compose and further simplify these approaches to reduce the set of input variables on which new reliable regression test case generation can be based.

3.1 Graph-less dynamic slicing and impact analysis

In [15] a new approach of producing dynamic program slices is proposed. The main idea of the work is to apply dependence analysis to dynamic slicing [1, 13, 12] instead of employing traditional techniques that usually require a graph-based representation and might seriously confine application possibilities due to memory consumption. The dynamic dependences that are tracked are the same as in the case of the graph representation, but instead of one huge graph, various smaller data structures are maintained.

Besides introducing alternative dependence-based methods, slicing scenarios are categorized [4] based on slicing direction, processing direction and global or demand-driven nature of the algorithm. Our impact analysis that will be presented in Section 5 relate closely to the forward, demand driven algorithm in [15]. The difference between the two approaches lies in the fact that we will not produce dynamic slices; therefore different data structures will be maintained. The reason why we do not apply dynamic slicing is that we need only a set of variables, and not a slice of the entire program.

3.2 Mutation-based regression testing

Paper [16] deals with regression test generation. The generation process has two stages: in the first stage existing test cases are categorized similarly to the previously mentioned (S0, S1, S2, S3) cases. Based on the outcome of the first stage, a new test case will be generated that effectively tests a modification. Our work is derived from that article; however, there are a few important improvements. First of all, we allow more than one modification to occur in the source code, and generate not only a test case, but a test pair. The test pair should match the changed definition of reliability.

4 Tools and notations

In order to perform dynamic impact analysis, the source code has to be carefully instrumented. During the execution of the instrumented code each traversal through a previously inserted sensor is registered. We will show that it is not necessary to maintain a log file (which again can grow huge) and log the registered traversal.

The relatively complex instrumentation that is required for real C++ code can be performed using various tools, like the Columbus framework [11]. In the following we briefly describe the used notations and information that instrumentation must provide. We are going to employ sequence-point level instrumentation, which means that sensors are inserted after each sequence point. This might imply that the trace can grow too large to handle, however, as we will see, it can be produced and processed on the fly.

For the identification of types, their fully qualified name is used

```
(namespace1::namespace2::...::Class1::Class2..).
```

All typedefs have to be resolved so that their corresponding type that can be identified.

For the unique identification of variables, we use the following notation:

D(v, s, q, A_v, A_p),

where v is the fully qualified name of the variable, s is the identification number of the statement which runs the q^{th} time, and v appears in the q^{th} run of s. Since C++ support pointer types, we have to distinguish between the memory location where the variable resides, and the memory location it points to in case it is a pointer. A_v represents the memory location of the variable, and A_p is the pointed memory location (for non-pointer typed variables, A_v and A_p are equivalent). A variable can be either global, static, local, or member variable. For the latter the

DD(D_{object}, D_{member})

notation is used. Let's consider the example when there is a class named Foo and there is a Bar typed member variable called b. When we instantiate an object of Foo at a uniquely identified program location, both members of the DD pair can be filled in.

```
(s, q):  Foo f;
```

Let's suppose we would like to describe member b. Then the following entry will be generated:

```
DD(D(Foo::f, s, q, 0x13217ffa4, 0x13217ffa4),
     D(Bar::b, s, q, 0x1322a4c28, 0x1322a4c28))
```

Static, local or global variables can also be described this way with D_{object} being NULL in these cases.

For local variables the fully qualified name has to be integrated with the exact block number where the local variable is defined. Pointer and function pointer variables can be described similarly.

At each sequence point along the execution path we have to record the defined (DEF) and used (USE) variables, and each variable has to be identified with the above specified granularity. (Both of them contain variables that are identified using the DD notation above.)

At each function we store the exact signature along with the source code location in the call stack.

C++ rigorously defines destructors to be run deterministically when execution leaves scope, or when an explicit delete is requested. Destructors should be instrumented just like ordinary functions, but with virtual or estimated line or column number.

Another instrumentation requirement is in connection with the lazy evaluation strategy of C++. Only those variables should appear in the instrumentation log that are really used or defined. In the following we will refer to these variables as actually defined/actually used variables.

5 Forward symptom analyzer algorithm

In this section we present our forward dynamic impact analyzer algorithm.

As we have shown previously, it is possible to identify memory locations for each variable. Consequently, it is not necessary to start our algorithm from the beginning of the program, rather from the location of the first occurrence of the modified statement. However, this approach implicitly implies that the execution history of the test case is the same for the original and modified programs until the first occurrence of the modification. Unfortunately, the execution history can be too large to log and to keep in memory, and the solution would not have a significant advantage over dynamic slicing.

To overcome this difficulty, it is also possible to start the algorithm from the very beginning of the program, and employ an online algorithm that processes log entries on-the-fly. By online we mean that the instrumentation sensors write the log entries to a buffered stream, and the impact analyzer fetches them on-the-fly. Although this way a slight performance loss occurs, but we gain significant advantage in the field of storage and memory consumption, which are usually the critical factors.

Therefore, the input of the algorithm is not the execution history, rather a test case that has previously been selected. The algorithm will identify both the same-output symptom (S1/S2/S3) and the affected output statement or predicate (P).

Along the execution path, all variables have to be meticulously identified and tracked in order to easily maintain the DEF and USE sets at each sequence point. To achieve this goal, all kinds of assignment operations between variables need to be described in terms of the above notations. Originally, we treated simple (built-in) and user-defined types separately, but it turned out that it is not necessary to make distinction between the two categories. Since these two elements take the same form, we explain only the assignment of simple (local) variables. Different

cases are shown in Table 1. In the first column the possible types of underlying variables and the location of their definition is shown. The second column contains the assignment operations again with the location, while in the third column the instrumentation entry can be seen. The format of the entry is in the following form: L stands for assignment location, R for actually referenced variable, D for defined variable. Please note that the location means a sequence point.

Variable types and locations	**Assignment location/ operation**	**Instrumentation entry**
(Si,Qi) int i; (Sj,Qj) int j;	(S,Q) i= j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Avj)* D: *D(i, Si, Qi, Avi, Avi)*
(Si,Qi) int *i; (Sj,Qj) int *j;	(S,Q): i = j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Apj)* *D: D(i, Si, Qi, Avi, Apj)*
(Si,Qi) int *i; (Sj,Qj) int j;	(S,Q): i = &j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Avj)* *D: D(i, Si, Qi, Avi, Avj)*
(Si,Qi) int *i; (Sj,Qj) *j; int a;	(S,Q): i = j+2+a;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Apj)* *D: D(i, Si, Qi, Avi,* *Apj+sizeof(*j)*(2+a))*
(Si,Qi) int *i; (Sj,Qj) int *j;	(S,Q): *i = *j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Apj)* *D: D(i, Si, Qi, Avi, Api)*
(Si,Qi) int *i; (Sj,Qj) int j;	(S,Q): *i = j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Avj)* *D: D(i, Si, Qi, Avi, Api)*
(Si,Qi) int *i;	(S, Q): i = new int;	*L: (S,Q)* *R: -* *D: D(i, Si, Qi, Avi, Api-New)*

Table 1: Assignment of primitive types

As we have previously mentioned, the assignment of primitive types can be applied to user-defined types as well, although there are some important extensions. C++ allows programmers to overload default operators, including the assignment operator. If there is no explicit user defined assignment operator in a class, then the default assignment operator (member-wise assignment) will be applied. On the other hand, if there is a custom assignment operator, its effect has to be preserved in the execution history.

At this point we have all of the necessary information to describe the intra-procedural version of the forward dynamic symptom analyzer algorithm. Later, it will be extended to its final inter-procedural form.

The input of the algorithm is the test case, the location of the modification,

and the set of variables that are defined at the modified statement. Because the set of defined variables at the modification is not known, and generating the whole execution trace is not acceptable, the values of the actually defined variables have to be calculated in a preprocessing step.

The preprocessing step requires the introduction of a set that stores variables of interest along the execution path. This set will be referred to as *Varstore*. *Varstore* is set to empty. At each variable assignment the defined variable calculated based on rules listed in is added to *Varstore*. When execution leaves the scope, all local variables will be removed. The same case holds when an explicit delete operation is requested. When we reach the first occurrence of the modified statement, the memory location of the actually defined variables can be calculated, and the set *Varstore* can be deleted. The introduction of *Varstore* is important because of the pointer typed variables of C++. Consider the example in Listing 2. All three variables (*n*, *ip*, *jp*) will be added to *Varstore*, and at the predicate we can detect that we refer to the same variable that was modified.

Listing 2 Pointer example

```
int n = 2;

int *ip = *jp = &n;

//modification:

*ip = 3; //original: *ip = 6;

...

if(*jp > 5)
```

After the initialization step we introduce a set called *Affect* for storing variables that are directly or indirectly affected by variables defined at the modified statement. The main steps of our algorithm without the preprocessing step are the following.

1. *Affect* is initialized with variables that are defined at the modified statement i_{mod} and refer to the same memory location (based on *Varstore*), starting point is set to i_{mod}.

2. From i_{mod} we traverse over statements (i_q) along the execution path according to test case T, and based on the type of this statement, we take one of the following actions:

   a) If i_q is an output statement (in other words it is not a predicate and does not define variables), and there is at least one used variable from *Affect*,

then the underlying symptom is trivially S1, and the affected statement is sq, in addition the algorithm can safely terminate.

b) If i_q is a predicate, then all actually used variables are considered. If the intersection of this set and *Affect* is not empty, then S2 is identified, and the affected statement is i_q. When predicate $i_q$is the modified statement then S2 is also identified at the modified statement.

c) If i_q is an assignment, then for each w variable used at i_q the presence of the variable in *Affect* is checked. If a w variable is in *Affect* then the defined variables in i_q are added to *Affect*. The statement defining w is marked as effective.
For each w variable in *Affect* it is checked if the w variable is defined at i_q and the last definition of w is not at i_q. If the previous condition is true then w is removed from *Affect*, moreover if the last definition is not effective the S3 is identified.

In order to successfully extend this algorithm to the inter-procedural case, we have to address parameter passing methods, and return values as well.

Method	Modelling
By value The same as (int i=j)	*D(j, Sj, Qj, Avj, Avj)* *D(i, Si, Qi, Avi, Avi)*
By address The same as (int *i=&j) OR (int *i=j) if j is a pointer	*D(j, Sj, Qj, Avj, Avj)* *D(i, Si, Qi, Avi, Avj)* OR *D(j, Sj, Qj, Avj, Apj)* *D(i, Si, Qi, Avi, Api)*
By reference	*D(j, Sj, Qj, Avj, Avj)* *D(i, Si, Qi, Avj, Avj)*

Table 2: Parameter passing methods and their representation

In C++ parameters can be passed via one of the following methods: by value, by address, and by reference.

Within a function any assignment to a parameter that has previously been passed by reference will not take effect outside the function, yet these assignments can alter the execution path through the return value of the function. A parameter passing by value can be modeled as an assignment to a local variable.

Parameter passing by address means the passing of pointer variables. Any assignment to a pointer variable within a function will not cause any side effect outside. Nevertheless, with the modification of the pointed memory location via dereference (*) operator side effects may occur. Passing by address can be thought of as introducing a new local pointer variable originally set to the same memory location as the pointed memory location of the actual parameter.

Since a reference can be regarded as the synonym of a memory location, any modification to a parameter passed by reference will take effect outside the function.

Algorithm 1 Symptom analyzer

Function SymptomAndLocation(T, S, P)

```
1: Affect={i_mod.Def}
2: S = Nothing
3: for each statement i_q in ExecutionPath(T) from i_mod to i_last do
4:   if i_q is output and i_q.Use∩Affect ≠ ∅ then
5:     S = S1;
6:     P = i_q;
7:     terminate;
8:   end if
9:   if i_q is predicate and (i_q.Use∩Affect ≠ ∅ or i_q is the modified statement)
     then
10:    S = S2;
11:    P = i_q;
12:    terminate;
13:  end if
14:  if i_q is definition or function return then
15:    for each w ∈ i_q.Use do
16:      if w ∈ Affect then
17:        Affect=Affect ∪ i_q.Def
18:        mark d^w as effective
19:      end if
20:    end for
21:    for each w ∈ Affect do
22:      if w ∈ i_q.Def and d^w ≠ i_q then
23:        Affect=Affect\{w}
24:        if d^w is not effective then
25:          S=S3
26:          P=i_q
27:        end if
28:      end if
29:    end for
30:  end if
31:  if i_q is function call then
32:    AssignParams(i_q)
33:  end if
34: end for
```

The three cases are summarized in Table 2 using the previously introduced notation. Please note that j represents the actual parameter, while i is the formal parameter.

In order to complete our extension, we have to cover the handling of return values. The return statement can also be substituted by a virtual assignment.

Namely, return I; can be exchanged to the *actal_retvar=I* assignment operation. This way we can trace back the problem of return values to different parameter passing methods.

The pseudo-code of the inter-procedural algorithm (which is basically the same as the intra-procedural version) can be seen in Algorithm Listing 1.

6 Finding influencing input variables

In theory, we could either apply a backward or a forward algorithm to find those input variables that have an influence on the statement that causes ineffectiveness. However, since in the first step we developed and applied a forward method, it would be more comfortable to extend that and keep the key concept. As we will see, with very little adjustment the previous algorithm can be tuned to solve our second problem. Consequently, the two stages can share the same implementation.

The main steps of the algorithm include:

1. Finding variable definitions of input variables. An input variable can be a constant definition, data read from standard input/file, any parameter of main, or a default parameter of a function.
2. We perform the previously described impact analysis with the difference that we also keep track of effective input variables and omit those parts of the algorithm that identify symptoms. Since we are unaware of at which predicate should the execution path be altered, we monitor each predicate.

In the following we detail only the intra-procedural version of the algorithm, since the inter-procedural version remains unchanged.

- The set *Affect* will contain variables directly or indirectly affected by any input variables. We index the elements of Affect with the input variable that has an influence on that specific variable. This means that *Affect* might contain the same variable multiple times with different indices related to input variables.
- Traverse along the execution path from the first statement, and consider each statement i_q. Based on the type of i_q., the behavior of the algorithm differs.
- If i_q. is an input statement, and variable d will be assigned at i_q., then $d_{(d)} \in$ *Affect*
- If i_q. is a predicate, then only actually executed conditions should be evaluated. For each actually executed condition and for each actually used variable if $u(v) \in$ *Affect*, then v is added to effective input variable set of the predicate
- If i_q. is an assignment statement, then the defined variable is deleted from *Affect* with all indices. If some input variables are used, then the defined variable is added to *Affect* indexed with the input variable. If a j non-input variable is used, for which $j_{(v)} \in$ *Affect*, then $d_{(v)}$ is added to Affect.

- The influencing input variables can be calculated as the the union of effective input variable sets of the predicates.

Algorithm 2 Influencing inputs

Function FindInfluencingInputs(T, V)

```
1: Affect={}
2: for each statement i_q in ExecutionPath(T) from i_first to i_last do
3:   if i_q is input statment then
4:     Affect=Affect ∪ i_q.Def_(iq.Def)
5:   end if
6:   if i_q is predicate then
7:     for each u ∈ i_q.ExecConditions.Use do
8:       if u_(v) ∈ Affect then
9:         V(i_q)=V(i_q) ∪ v
10:      end if
11:    end for
12:  end if
13:  if i_q is definition or function return then
14:    for each d ∈ i_q.Def do
15:      for each d_(v) ∈ Affect do
16:        Affect=Affect \ d_(v)
17:      end for
18:    end for
19:    for each j ∈ i_q.Use ∩ Affect.Indices do
20:      Affect=Affect ∪ {i_q.Def_(j)}
21:    end for
22:    for each w ∈ i_q.Use do
23:      for each w_(j) ∈ Affect do
24:        Affect=Affect ∪ {i_q.Def_(j)}
25:      end for
26:    end for
27:  end if
28:  if i_q is function call then
29:    AssignParams(i_q)
30:  end if
31: end for
```

7 Full example

In order to present the usability of the proposed solution, we show the two stages in work through a fully C++ compliant example.

The example deals with arithmetic operations. A general operation is represented as an abstract class, and all specific operations derive from this class. In this simplified source we use two operations: addition and multiplication.

```
1. #include <iostream>
2. #include <string>
3. #include <stdlib.h>

4. using namespace std;

5. namespace MathOperation
6. {
7.   class Operation //base class
8.   {
9.   public:
10.   //Pure virtual function
11.     virtual int DoOperation(int x,int y)=0;
12.     virtual string OpName()=0;
13.   };

14.   //Derived class 1
15.   class AddOperation: public Operation
16.   {
17.   public:
18.     int DoOperation(int x, int y)
                        {return x + y;}
19.     string OpName() {return "Add";}
20.   };

21.   //Derived class 2
22.   class MulOperation: public Operation
23.   {
24.   public:
25.     int DoOperation(int x, int y)
                        {return x * y;}
26.     string OpName() {return "Mul";}
27.   };
28. }

29. class CImpactAnal
30. {
31. public:
32.   void QueryMethod()
33.   {
34.     int oplocal;
35.     cin >> oplocal;
36.     this->opcode = oplocal;
37.   }
38.   void DoCalculation(int x, int y)
39.   {
40.     MathOperation::Operation *op = NULL;
41.     //original: int opcode2=opcode;
42.     int opcode2=opcode-1;
43.     if(opcode2 > 1)
          op = new MathOperation::AddOperation();
44.     else
          op = new MathOperation::MulOperation();
45.     lastOp = op;
46.     cout << " Result: " <<
          op->DoOperation(x,    y) << endl;
47.   }
48.   void LastOperation()
49.   {
        cout << "Last op: " <<
        lastOp->OpName() << endl;
        delete lastOp;
50.   }
```

```
51. private:
52.   int opcode;
53.   MathOperation::Operation *lastOp;
54. };
55. int main(int argc, char* argv[])
56. {
57.   CImpactAnal *ia = new CImpactAnal();
58.   int x = 2;
59.   int y = 3;
60.   ia->QueryMethod();
61.   ia->DoCalculation(x, y);
62.   ia->LastOperation();
63.   return 0;
64. }
```

The client code executes an arithmetic operation on two constants based on user input. According to our previous definition, the program has three input variables: x, y, and *oplocal*. Remember that an input variable is either a constant definition, data read from standard input/file, any parameter of main, or a default parameter of a function. x and y are constants (might be either parameters of the main function), *oplocal* is user input.

For the sake of clarity, we present an example with exactly one modification, which takes place at line no. 42. The modification affects an assignment statement, because *opcode2=opcode* was changed to *opcode2=opcode-1*. (In case of more than one modification, the same procedure applies until the first occurrence of the first modification.)

In the following part we review the stages of the previously introduced algorithm in order to identify test-case critical input variables.

The first section of the first stage is the preprocessing step. During this stage the aim is to identify those variables that possibly get a new value at the modified statement. In the current example the preprocessing step works as follows: The entry point of the algorithm is set to the first line of the main function (to the beginning of the program).

The *Varstore* set that stores variables of interest in the preprocessing step is initialized to empty. After traversing line no. 57, there would be two entries in the set *Varstore*. One of them represents the object pointer variable *ia*, and the other the member variable opcode of type int. Then variables x and y are added during the traverse over lines 58 and 59.

At line 60 there is a call to the *QueryMethod* member function of the *CImpactAnal* class. When we reach line no. 34, the local variable *oplocal* is also added to the *Varstore* set. At line 35 the value of *oplocal* is redefined, but its memory location is unaffected, therefore there is no need to update its entry in the *Varstore* set. At line 36 the value of class member variable *opcode* is set therefore it should be added to *Varstore*. Since *oplocal* introduced at line 34 is a local variable, and we leave the scope of this definition at line no. 37, at that point it is removed from *Varstore*. Then execution returns to line no. 61, where there is a call to *DoCalculation*. At line 40 variable op, at line 42 variable opcode2 is added to *Varstore*. At line 42 we reach the modified statement for the first time. The actually defined variables can be calculated (in this case it is only opcode2). If there are any pointer

variables in *Varstore*, we have to check whether these variables point to some used variables.

The next stage is the symptom analyzer algorithm. Variable *opcode2* will be added to the set *Affect*, and the algorithm is started from the modified statement. Since the modified statement is an assignment, the used variables should be removed from *Affect*, but *opcode* is not in Affect, so this step is not required. After that step variables that are defined at the underlying statement, but the last definition did not take place at the current statement, are removed from *Affect*. This step now does not execute because the previously mentioned conditions do not hold. Now the algorithm advances to the next statement that takes places at line 43, and is a predicate. Because the intersection of the used variables and the set *Affect* is not empty, the modification has an influence on this predicate, so S2 is identified, and the algorithm terminates.

The last step before automatic test generation is the identification of influencing input variables. This stage starts at the beginning of the program. At each input variable definition, the variable will be added to *Affect* indexed by itself. In our example that means that $x_{(x)}$, $y_{(y)}$, and $oplocal_{(oplocal)}$ are added to *Affect* during execution. At line 35 variable *oplocal* is redefined, there is no need to change *Affect*. When we reach line 36, $opcode_{(oplocal)}$ is also added to *Affect* After leaving method *QueryMethod* and entering *DoCalculation* we reach line 40 where the definition of *op* resides. There is no need to add it to *Affect* because it does not depend on any variables of *Affect*. As we previously mentioned there is an entry $opcode_{(oplocal)}$ in *Affect* therefore when we reach line 42 defining *opcode2* based on *opcode*, the $opcode2_{(oplocal)}$ is added to *Affect*. Now we reached the modified statement where the used variable is only *opcode2* indexed by *oplocal* therefore we can establish that the only influencing input variable is *oplocal*.

At this point we know that the identified symptom is S2, in other words the modification influences a predicate in line 43. We have also managed to identify the only input variable that influences that predicate.

The whole framework would then choose a regression test from the test suite that reaches the modification. From this test a pair of test cases will be created using a dataflow based generation algorithm. The variables that are allowed to modify are the ones that have an influence on the predicate, in our case 'oplocal'. The values of 'oplocal' should be close to each other, and close to that value where the predicate evaluates to different values (the test case should be sharp regarding the influenced predicate).

8 Conclusion

In this paper we introduced a changed concept of regression testing that was motivated by the shortcomings of existing techniques. We have shown that classical modification traversing and modification revealing regression tests are not necessarily error revealing. The new concept focuses on the reliability of test cases, it tries to assure that each test that reaches a modification has an influence on at

least one output statement.

In order to achieve this goal, existing test cases have to be filtered, and those that are usable, should be transformed. So the main task is automated test generation that is appropriate for the changed regression testing concept. Test generation can be thought of as a search in a space spanned by the input variables of the program. Unfortunately, the dimension of this search space can grow over the limits where a search can be comfortably managed. Consequently, our goal was to reduce the dimension of this search space, and find only input variables that have an influence on the modification for a given test case.

Instead of dynamic program slicing, we applied a custom dynamic impact analysis which is more appropriate for this problem. Besides operating in a forward manner, the algorithm is also superior to dynamic slicing in memory consumption, which can be a critical factor when dealing with large applications.

All of our methods have been developed to work in a C++ specific language environment. Therefore, many C++ constructs have been covered in detail including both procedural and object oriented constructs like pointers and function pointers, different parameter passing methods, classes, member and object variables and inheritance. Since C++ exposes a wider range of language constructs and gives more freedom to the programmer than most modern object oriented languages, we believe that the proposed approach can be successfully adjusted to work in other environments as well. The relative simplicity of the dynamic impact analyzer algorithm is due to a complex instrumentation mechanism. The instrumentation step is supported by the Columbus framework [11]. Since Columbus is currently not able to handle all requirements we described, first we should extend that tool. A further technical limitation is the handling of different C++ dialects. Although the ANSI standard is adopted by nearly all compilers that are used for production systems, many of them support additional features that do not comply with the standard. Consequently, the instrumentation step has to prepare for differences.

In order to fully cover the potential of C++, we also have to address issues related to the template mechanism. Technically, it is a must to be able to insert the instrumentation stage after the preprocessing step has been completed.

References

[1] A. Beszedes, T. Gergely, Zs. M. Szabo, J. Csirik, T. Gyimothy. Dynamic slicing method for maintenance of large C programs. CSMR 2001, pages 105–113.

[2] B. Korel, Ali M. Al-Yami. Automated Regression Test Generation. ISSTA 1998: 143–152

[3] B. Korel. Automated Test Data Generation for Programs with Procedures. ISSTA 1996: 209–215

[4] F. Tip. A survey of program slicing techniques. Journal of Programming Languages, 3(3):121–189, Sept. 1995.

[5] G. Rothermel and M.J. Harrold. A safe, efficient algorithm for regression test selection. Proceedings of the International Conference on Software Maintenance, pp. 358–367, September 1993.

[6] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software. Proceedings of the International Symposium on Software Testing and Analysis, pp. 169–184, August 1994.

[7] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. IEEE Transactions on Software Engineering, 22(8):529-551, August 1996.

[8] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. Journal of Software Testing, Verification and Reliability, 10(2), June 2000.

[9] I. Forgacs and A Hajnal. An Applicable Test Data Generation Algorithm for Domain Errors. In Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, Clearwater Beach, Florida, March, 1998.

[10] Pargas, R. P., Harrold, M. J., and Peck, R. R. Test data generation using genetic algorithms. The Journal of Software Testing, Verification and Reliability 9 (1999), 263–282.

[11] R. Ferenc, A. Beszedes and T. Gyimothy. Extracting Facts with Columbus from C++ Code. In Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), Tampere, Finland, pages 4-8, March 24-26, 2004.

[12] R. Gupta, M. Harrold, M. Soffa. An approach to regression testing using slicing. Conference on Software Maintenance, 1992, pp. 299–308.

[13] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. In Proceedings of ESEC/FSE'99, number 1687 in Lecture Notes in Computer Science, pages 303-321. Springer-Verlag, Sept. 1999.

[14] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In Proceedings of the Eighth International Symposium on Software Reliability Engineering, pages 230–238, Nov. 1997. 10.

[15] A. Beszedes, T. Gergely and T. Gyimothy. Graph-Less Dynamic Dependence-Based Dynamic Slicing Algorithms. In Proceedings of the 6th IEEE Int'l Workshop on Source Code Analysis and Manipulation, pages 21–30. IEEE Computer Society, 2006.

[16] I. Forgacs, E. Takacs. Mutation-Based Regression Testing. Conference proceedings. Tenth International Software Qualiti Week 1997. San Francisco, 1997. Vol. 2. San Francisco, Software Res. Inst., 1997.

[17] N. Gupta, A. Mathur, M. Soffa. Automated Test Data Generation Using an Iterative Relaxation Method. Foundations of Software Engineering, pages 231–244, 1998.

[18] Graves, T. L., Harrold, M. J., Kim, J., Porter, A., and Rothermel, G. An empirical study of regression test selection techniques. ACM Transactions on Software Engineering and Methodology. 10, 2 pages 184–208, 2001.

[19] M. Harrold and M. Soffa. An incremental approach to unit testing during maintenance. In Proceedings of the Conference on Software Maintenance, pages 362–367, Oct. 1988.

Received 10th October 2006

Acta Cybernetica 18 (2008) 503–527.

Two New Approximation Algorithms for the Maximum Planar Subgraph Problem*

Timo Poranen†

Abstract

The maximum planar subgraph problem (MPS) is defined as follows: given a graph G, find a largest planar subgraph of G. The problem is NP-hard and it has applications in graph drawing and resource location optimization. Călinescu et al. [J. Alg. 27, 269-302 (1998)] presented the first approximation algorithms for MPS with nontrivial performance ratios. Two algorithms were given, a simple algorithm which runs in linear time for bounded-degree graphs with a ratio 7/18 and a more complicated algorithm with a ratio 4/9. Both algorithms produce outerplanar subgraphs.

In this article we present two new versions of the simpler algorithm. The first new algorithm still runs in the same time, produces outerplanar subgraphs, has at least the same performance ratio as the original algorithm, but in practice it finds larger planar subgraphs than the original algorithm. The second new algorithm has similar properties to the first algorithm, but it produces only planar subgraphs. We conjecture that the performance ratios of our algorithms are at least 4/9 for MPS.

We experimentally compare the new algorithms against the original simple algorithm. We also apply the new algorithms for approximating the thickness and outerthickness of a graph. Experiments show that the new algorithms produce clearly better approximations than the original simple algorithm by Călinescu et al.

Keywords: maximum planar subgraph, maximum outerplanar subgraph, thickness, outerthickness, triangular cactus heuristic, approximation algorithm

1 Introduction

A graph is *planar* if it admits a plane drawing where no two distinct edges intersect apart from their endpoints, otherwise the graph is *non-planar*. Let $G = (V, E)$ be a

*Work funded by the Tampere Graduate School in Information Science and Engineering (TISE) and supported by the Academy of Finland (Project 51528). The results of this paper have originally published in the PhD thesis [32] of the author.

†Department of Computer Sciences, P.O. Box 607, FIN-33014 University of Tampere, Finland, E-mail: tp@cs.uta.fi

graph without loops and parallel edges. If a graph $G' = (V, E')$ is a planar subgraph of G such that every graph G'' obtained from G' by adding an edge from $E \setminus E'$ is non-planar, then G' is called a *maximal planar subgraph* of G. Let $G' = (V, E')$ be a maximal planar subgraph of G. If there is no planar subgraph $G'' = (V, E'')$ of G with $|E''| > |E'|$, then G' is a *maximum planar subgraph*. A maximal planar subgraph is maximal in the sense that adding edges is not possible and a maximum planar subgraph is maximal with respect to the cardinality of its edge set.

A planar graph is outerplanar if it admits a plane drawing where all its vertices lie on the same face and no two distinct edges intersect apart from their endpoints. *Maximal* and *maximum outerplanar subgraphs* are defined analogously.

Maximum planar subgraphs have applications in facility layout [17] and graph drawing [20, 22]. The problems of finding a maximum planar subgraph or a maximum outerplanar subgraph are denoted respectively throughout this work by MPS and MOPS. Both problems are known to be NP-hard [26, 38]. Therefore, heuristic algorithms are needed to find good approximations. Several methods for approximating MPS are given in the literature, see for example a survey by Liebers [25] and the references given there.

The *performance ratio* of an approximation algorithm for a maximization problem is the worst case ratio of solutions obtained to the cost of optimal solution. The performance ratio measures the solution quality of an approximation algorithm, the closer to 1 the ratio is, the better solutions the algorithm guarantees. A simple way to find an approximation for MPS is to produce a spanning tree for the input graph. Since a spanning tree of an n-vertex graph contains $n - 1$ edges and a maximum planar subgraph could contain at most $3n - 6$ edges, the performance ratio of this method is 1/3 [10].

Călinescu et al. [5] presented the first approximation algorithms with non-trivial performance ratios for MPS and MOPS. Their method, *triangular cactus heuristic*, gives a performance ratio of 4/9 for MPS and 2/3 for MOPS. These approximations can be achieved by a complicated algorithm having a running time of $O(m^{3/2} n \log^6 n)$ for a graph with n vertices and m edges. There is no known implementation of this algorithm. Călinescu et al. also presented a simple version of their algorithm having performance ratios 7/18 and 7/12 respectively. The simple algorithm runs in linear time for bounded-degree graphs.

In this paper we introduce two new algorithms based on the simple version of the algorithm presented by Călinescu et al. for MPS and MOPS. Our first algorithm also runs in linear time for bounded-degree graphs and it has at least the same performance ratio as the original simple algorithm. The second algorithm has properties similar to those of the first algorithm, but it produces only planar subgraphs. We conjecture that the new algorithms have at least the same performance ratio as the more complicated algorithm. Our experiments show that the new algorithms produce clearly better approximations than the original simple algorithm. Since the better algorithm by Călinescu et al. is difficult to implement, it is not included in our experiments.

The *thickness* of a graph is the minimum number of planar subgraphs into which the graph can be decomposed. The *outerthickness* of a graph is the minimum

number of outerplanar subgraphs into which the graph can be decomposed. The thickness and outerthickness are topological invariants that measure the graph's embeddability into the plane. Determining the thickness of a graph plays an important role in VLSI circuit design: the minimum number of planar subgraphs whose union is the graph corresponding to an electronic circuit provides an efficient way to find a decomposition for the distinct layers of the circuit [30].

Determining the thickness of a given graph is NP-hard [28] but the complexity status of determining the outerthickness is not known. The thickness is known for hypercubes [23], complete graphs [1, 2] and complete bipartite graphs [3]. Similar results for outerthickness have been reported by Guy and Nowakowski [14, 15].

Only one method to obtain approximations for thickness has been introduced in the literature: extract maximal planar subgraphs from the original graph until the remaining graph is planar [8, 30]. All earlier algorithms apply planarity tests to construct large planar subgraphs.

A new approach presented here for approximating the thickness of a graph is to extract planar subgraphs in such a way that the extracted graph is constructed without using planarity testing algorithms. In this paper we apply the simple algorithm by Călinescu et al. [5] with our new algorithms for approximating the thickness and outerthickness of a graph. Our experiments show that the new algorithms give better approximations than the original simple algorithm.

The rest of this paper is organised as follows. Next we give graph theoretical definitions and introduce a greedy algorithm for MPS with the extraction algorithm for the thickness problem. We also describe the triangular cactus heuristic. The new algorithms and their theoretical properties are discussed in Section 3. The experimental comparison of the algorithms for MPS is presented in Section 4 and then the algorithms are applied to the thickness problem in Section 5. The last section concludes our paper.

2 Preliminaries

2.1 Graph-theoretical definitions

For the basic graph-theoretical definitions, we refer to Harary [16]. Throughout this work we assume that graphs are simple and connected. An $m \times n$ *grid graph* is the product of paths of length m and n and contains mn vertices and $2mn - n - m$ edges.

A *triangular structure* is a graph in which every cycle is a triangle. A *triangular cactus* is a triangular structure in which every edge is in some cycle. A triangular structure is outerplanar, since the graph cannot contain a subdivision of K_4 or $K_{3,2}$.

A *maximal outerplanar graph* (mop) is an outerplanar graph such that inserting any edge produces a non-outerplanar graph. Next we present a useful characterization for mops [4] having at least three vertices.

Definition 2.1. *Mops having at least three vertices can be defined recursively as follows:*

1. *K_3 is a mop.*
2. *If G is a mop which is embedded in the plane so that every vertex lies on the outer face and G' is obtained by joining a new vertex to two vertices of an edge on the outer face of G, then G' is a mop.*
3. *H is a mop if and only if it can be obtained from K_3 by a finite sequence of applications of statement (2).*

2.2 A greedy algorithm for MPS

Throughout this work, all algorithms for MPS return a subgraph of the input graph. The cost of a solution is the number of edges in the returned approximation. Thickness algorithms return a partition of the edges of the input graph. The cost of a solution is the number of subsets in the partition.

A greedy algorithm to search for a maximal planar subgraph is to apply a planarity testing algorithm and to add as many edges as possible to a planar subgraph. See Algorithm 2.1 (GRE) for a detailed description of this edge adding method. The performance ratio of GRE is 1/3 for MPS [10].

GRE($G = (V, E), G' = (V, E')$)
1 $E'' = E \setminus E'$;
2 **while** there is an edge (u, v) in E''
3 **do** $E' \leftarrow E' \cup \{(u, v)\}$, $E'' \leftarrow E'' \setminus \{(u, v)\}$;
4 **if** (V, E') is not planar
5 **then** $E' \leftarrow E' \setminus \{(u, v)\}$;
6 **return** (V, E');

Algorithm 2.1: GRE for MPS.

GRE takes as input a graph $G = (V, E)$ and its planar subgraph $G' = (V, E')$. The algorithm returns a maximal planar subgraph containing the input graph as a subgraph. Our reason for assuming that a planar subgraph is given as input to the algorithm is that then we can apply GRE to improve solutions of other heuristics. This approach is described in Section 3. The running time of GRE heuristic is $O(|V||E|)$ if a linear time planarity testing algorithm [18] is applied at Step 4.

2.3 The thickness heuristic

Next we describe the basic approach to obtain approximations for thickness. The extraction method was first studied by Cimikowski [8] and Mutzel et al. [30]. For

a detailed description of the extracting method see Algorithm 2.2 (THICK). Step 3 of the algorithm is usually given as "find a maximal/maximum planar subgraph" instead of finding just a planar subgraph.

THICK($G = (V, E)$)
1 $P \leftarrow \emptyset$; $t \leftarrow 1$;
2 **while** $E \neq \emptyset$
3 **do** find a planar subgraph $G' = (V, E_t)$ of G;
4 $E \leftarrow E \setminus E_t$;
5 $P \leftarrow P \cup \{E_t\}$;
6 $t \leftarrow t + 1$;
7 **return** P;

Algorithm 2.2: Basic structure of the extraction algorithm for the thickness problem.

THICK takes as input a graph $G = (V, E)$ and it returns a partition of the edges into subsets inducing planar subgraphs. The running time of Algorithm 2.2 depends heavily on the method used in Step 3. If a maximal planar subgraph is recognised from the input graph by applying GRE, the running time of the algorithm is $O(|V|^2|E|)$.

2.4 Triangular cactus heuristics

Next we introduce the triangular cactus algorithms for MPS and MOPS [5]. Given a connected graph $G = (V, E)$, the triangular cactus heuristic is based on finding a subgraph $G' = (V, E')$ whose components are triangular cacti. The subgraph is constructed in the following way: E' is initialized to be empty. Triangles having all vertices in different components in G' are searched from G and added to E'. After all suitable triangles have been added to G', the subgraph is connected by adding edges until the resulting graph is a connected triangular structure. See Algorithm 2.3 (CA) for a detailed description of the triangular cactus heuristic. Steps 2 and 3 are called Phase 1 (the construction phase of a triangular cactus) and Steps 4 and 5 are called Phase 2 (the connection phase) of the algorithm.

Algorithm CA can be implemented to run in linear time as shown by Călinescu et al. [5], provided that the maximum degree of the graph is bounded by a constant. The theorem below concludes the properties of CA.

Theorem 2.2. *[5]* CA *runs in linear time for bounded-degree graphs. The performance ratio of* CA *for MPS is* 7/18 *and for MOPS* 7/12.

If a maximum triangular cactus is searched for in Phase 1 instead of the triangular cactus of CA, the performance ratio increases to 4/9 for MPS and 2/3 for MOPS [5]. The algorithm is denoted by CA_M.

CA($G = (V, E)$)
1 $E' \leftarrow \emptyset$;
2 **while** there is a triangle (v_1, v_2, v_3) in G such that
v_1, v_2 and v_3 belong to different components of (V, E')
3 **do** $E' \leftarrow E' \cup \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$;
4 **while** there is an edge $(v_1, v_2) \in E$ such that v_1 and v_2 belong to different components in (V, E')
5 **do** $E' \leftarrow E' \cup \{(v_1, v_2)\}$;
6 **return** (V, E');

Algorithm 2.3: CA for MPS and MOPS.

All the known algorithms for finding a maximum triangular structure are very complicated. The method proposed by Călinescu et al. [5] was based on reducing the problem of finding a maximum triangular structure to a graphic matroid parity problem [27] and then solving it with an algorithm by Gabow and Stallman [11]. This method leads to running time $O(m^{3/2} \log^6 n)$. There are no known implementations of the algorithm. The following theorem formulates the properties of CA_M.

Theorem 2.3. *[5] The performance ratio of* CA_M *for MPS is* 4/9 *and for MOPS* 2/3. CA_M *runs in* $O(m^{3/2} \log^6 n)$ *for a graph with* m *edges and* n *vertices.*

3 New algorithms for MPS and MOPS

In this section we introduce first our new algorithms, CA1 for MPS and MOPS and CA2 for MPS. We also study the theoretical properties of the algorithms and compare them with CA and CA_M.

When a triangle is found in CA, it always connects three vertices from different components of the subgraph. It is easy to see that not all the vertices of a triangle need belong to different components. It is enough to have two vertices v_1 and v_2 joined by an edge (v_1, v_2) in one component and the third vertex v_3 in another component forming a triangle (v_1, v_2, v_3). When triangles are added using this principle whenever possible, and otherwise requiring that the vertices of the triangle belong to different components, the planarity is not violated. If any triangle is added with this new principle, the resulting graph is no longer a triangular structure. To ensure that the constructed subgraph is also outerplanar, it is necessary and sufficient to demand that (v_1, v_2) belongs to at most two triangles at the same time. The algorithm applying this restriction and producing outerplanar subgraphs is denoted by CA1, and the algorithm without the restriction is denoted by CA2. The properties of CA1 are studied first. The exact description of CA1 is given in Algorithm 3.1.

CA1(G=(V,E))

1 $E' \leftarrow \emptyset$;
2 **repeat** **while** there is a triangle (v_1, v_2, v_3) in G such that (v_1, v_2) belongs to exactly one triangle in E' and v_3 to a different component of (V, E')
3 **do** $E' \leftarrow E' \cup \{(v_2, v_3), (v_3, v_1)\}$;
4 **if** there is a triangle (v_1, v_2, v_3) in G such that v_1, v_2 and v_3 belong to different components of (V, E')
5 **then** $E' \leftarrow E' \cup \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$;
6 **until** the number of edges in E' does not increase during the loop;
7 **while** there is an edge $(v_1, v_2) \in E$ such that v_1 and v_2 belong to different components in (V, E')
8 **do** $E' \leftarrow E' \cup \{(v_1, v_2)\}$;
9 **return** (V, E');

Algorithm 3.1: CA1 for MPS and MOPS.

CA1 was inspired by the recognition algorithm for maximal outerplanar graphs proposed by Mitchell [29]. The algorithm was based on extracting degree 2 vertices from the graph. In CA1, vertices of degree 2 are added to an outerplanar subgraph.

Figure 1 provides an illustration of the behaviour of CA and CA1 for the graph cimi-g4 [9], which is a non-planar graph with 10 vertices and 22 edges. A maximum planar subgraph of this graph contains 20 edges. The triangles are numbered in the order they are found. This order depends on the implementation of the algorithm and the representation of the graph. CA first finds four triangles and then it connects one remaining vertex with the rest of the subgraph. The planar subgraph contains 13 edges. CA1 finds first one triangle, then it adds 5 triangles that increase the number of edges by 2 and finally a triangle with three new edges is added. The size of the planar subgraph is now 16.

Next we show that CA1 can be implemented to run in linear time, if the maximum degree of the input graph is bounded by a constant.

Lemma 3.1. CA1 *runs in linear time for bounded-degree graphs.*

Proof. To establish that CA1 runs in linear time, it is sufficient to note that the steps where a triangle connecting two vertices from the same component and one vertex from another component take in total linear time provided that the degree of the graph is bounded. The total time for all other operations is linear for bounded-degree graphs by Theorem 2.2.

Suppose that the degree of a graph is bounded by a constant d. Each time an edge (v_1, v_2) is considered in Step 2, it takes at most d^2 time to check the adjacency lists of v_1 and v_2 to recognise a triangle. Since it is enough to consider each edge only once in the first while loop, CA1 runs in time $O(n)$ for bounded-degree graphs.

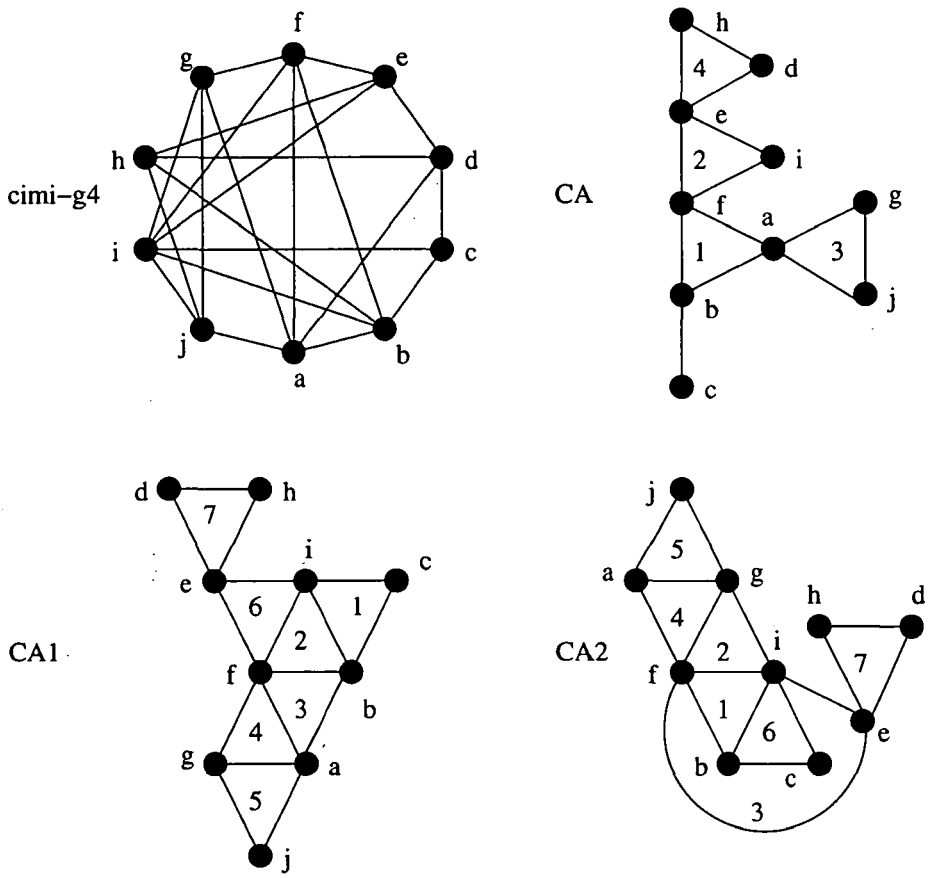


Figure 1: Illustrations of planar subgraphs for graph cimi-g4 found by CA, CA1 and CA2. Triangles are enumerated in the order they have been found in a sample run.

□

To show that the performance ratio of CA1 for MPS is at least 7/18 and for MOPS at least 7/12, the original proof of Theorem 2.2 can be applied directly. We only outline the important property of CA1 that makes it possible to apply the proof technique introduced by Călinescu et al. [5].

Lemma 3.2. *The performance ratio of* CA1 *for MPS is at least* 7/18 *and for MOPS at least* 7/12.

Proof. Let G_{CA} and G_{CA1} be the planar subgraphs produced by CA and CA1 after Phase 1 respectively. No triangle was added to G_{CA} if two of its vertices were in the same component. The same holds for G_{CA1}: there is no triangle in the input

graph with its vertices in different components in G_{CA1}. The original proof was based on this observation, and therefore, it follows that CA1 has at least the same performance ratio as CA. □

The proof of the upper bound given by Călinescu et al. [5] for the performance ratio of CA cannot be applied to CA1, but it is clear that the ratio cannot exceed 1/2, as shown by the following constructive proof.

Lemma 3.3. *The performance ratio of* CA1 *for MPS is at most* 1/2.

Proof. Let G be an $n \times n$ grid graph with $n \geq 2$. The graph has in total n^2 vertices and $2n^2 - 2n$ edges. Since G is planar, the maximum planar subgraph is the graph itself. CA1 finds a planar subgraph with $n^2 - 1$ edges by constructing a spanning tree of G. The ratio between the number of edges found by CA1 and the number of edges in G is

$$\frac{n^2 - 1}{2n^2 - 2n}.$$

The limit of the ratio is 1/2 as n tends to infinity.

□

Next we present a sample graph which shows that the performance ratio of CA1 for MOPS is at most 2/3.

Lemma 3.4. *The performance ratio of* CA1 *for MOPS is at most* 2/3.

Proof. Let G be a $2 \times n$ grid graph. G has in total $2n$ vertices and $3n - 2$ edges. Since G is outerplanar, the maximum outerplanar subgraph is the graph itself. CA1 finds an outerplanar subgraph with $2n - 1$ edges by constructing a spanning tree of G. The ratio between the number of edges found by CA1 and the number of edges in G is

$$\frac{2n - 1}{3n - 2}.$$

The limit of the ratio is 2/3 as n tends to infinity. □

We can now conclude the properties of CA1 for MPS and MOPS.

Theorem 3.5. *The performance ratio of* CA1 *for MPS is at least* 7/18 *and at most* 1/2. *The performance ratio of* CA1 *for MOPS is at least* 7/12 *and at most* 2/3. *The algorithm runs in linear time for bounded-degree graphs.*

There is a gap between the lower and upper bounds of the performance ratios of CA1 for MPS and MOPS, and the exact performance ratio is left open. One way to confirm or refute that the performance ratio is at least 4/9 for MPS is to show that a subgraph produced by CA1 has always at least the same number of edges as a maximum triangular structure of a graph. We present conjecture for the performance ratio of CA1 for MPS and MOPS. The computational experiments reported in the next section support the conjecture.

Conjecture 3.6. *The performance ratio of* CA1 *for MPS is at least* 4/9 *and for MOPS exactly* 2/3.

Next we study CA2. From the condition in the first while loop of CA1 it follows that at the end of the algorithm an edge of G' belongs at most to two triangles. It is not necessary in the case of planar subgraphs to require that one edge should belong to at most two triangles at the same time. The restriction "(v_1, v_2) belongs to exactly one triangle in E'" of the while loop of CA1 can be changed to "(v_1, v_2) belongs to E'". This observation leads to Algorithm CA2. Now outerplanarity is violated if at the end of the algorithm any edge belongs to more than two triangles (a forbidden subgraph $K_{3,2}$ is created [16]). The subgraph remains planar. In Figure 1, there is an illustration of the behaviour of CA2. Note that the edge (f, i) belongs to three triangles and hence, outerplanarity is violated. The planar subgraph found by CA2 contains 16 edges.

CA2($G = (V, E)$)
1 $E' \leftarrow \emptyset$;
2 **repeat** **while** there is a triangle (v_1, v_2, v_3) in G such that (v_1, v_2) belongs to E' and v_3 to a different component of (V, E')
3 **do** $E' \leftarrow E' \cup \{(v_2, v_3), (v_3, v_1)\}$;
4 **if** there is a triangle (v_1, v_2, v_3) in G such that v_1, v_2 and v_3 belong to different components in (V, E')
5 **then** $E' \leftarrow E' \cup \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$;
6 **until** the number of edges in E' does not increase during the loop;
7 **while** there is an edge $(v_1, v_2) \in E$ such that v_1 and v_2 belong to different components in (V, E')
8 **do** $E' \leftarrow E' \cup \{(v_1, v_2)\}$;
9 **return** (V, E');

Algorithm 3.2: CA2 for MPS.

The linear running time of CA2 for bounded-degree graphs follows directly by Theorem 2.2 for CA and by Lemma 3.1 for CA1. The bounds for the performance ratio of CA2 are the same as they are for CA1. The following theorem concludes the properties of CA2.

Theorem 3.7. *The performance ratio of* CA2 *for MPS is at least* 7/18 *and at most* 1/2, *and the algorithm runs in linear time for bounded-degree graphs.*

We give a similar conjecture for the performance ratio of CA2 as for CA1. This conjecture is also supported by the experiments reported in the next section.

Conjecture 3.8. *The performance ratio of* CA2 *for MPS is at least* 4/9.

Next we present three simple corollaries that describe the properties of the algorithms.

A difference between CA1, CA2, CA and CA_M is that CA1 and CA2 recognise maximal outerplanar graphs. This follows directly from Definition 2.1, which gave a recursive method to construct a maximal outerplanar graph.

Corollary 3.9. CA1 *and* CA2 *recognise maximal outerplanar graphs.*

The second corollary yields a graph class for which CA1 and CA2 find better approximations than CA_M.

Corollary 3.10. *There are graphs for which the limit of the ratio of the solutions of* CA1 *(or* CA2*) and* CA_M *is* 4/3.

Proof. Let G be a maximal outerplanar graph with n vertices. G has $2n-3$ edges. Since CA1 (CA2) finds all edges of a maximal outerplanar graph and a maximum triangular structure of a maximal outerplanar graph contains $3\lfloor(n-1)/2\rfloor$ edges [5], the ratio of the solutions of CA1 (CA2) and CA_M is 4/3 as n tends to infinity. □

Our third corollary describes the differences between CA2 and CA1 (CA_M).

Corollary 3.11. *There are graphs for which the limit of the ratio of the solutions of* CA2 *and* CA1 *(or* CA_M*) is* 2.

Proof. Let G be a graph with a single triangle containing vertices v_1, v_2 and v_3. Add to G a new vertex v_i, where $i > 3$, and two edges (v_i, v_1) and (v_i, v_2). Continue this process and denote the graph by G'. If G' has k vertices, it has $2(k-2)+1$ edges. Since CA2 finds all edges of G' and CA1 (CA_M) finds $k+1$ (k) edges, the ratio of the solutions of CA2 and CA1 (CA_M) is 2 as n tends to infinity. □

CA, CA1 and CA2 can be made greedy by giving the subgraph constructed in Phase 1 as input to GRE. These greedy versions are denoted by GCA, GCA1 and GCA2. Since GRE connects the subgraph, at least the same number of edges is added as in Phase 2 of CA, CA1 and CA2. Therefore, GCA, GCA1 and GCA2 have the same performance ratios as CA, CA1 and CA2 respectively.

4 MPS experiments

In this section, different algorithms for MPS are compared. More detailed comparison statistics for the algorithms can be found in [32].

4.1 MPS algorithms and comparison measures

We implemented the following algorithms for MPS: CA, CA1, CA2 and their greedy versions GCA, GCA1 and GCA2 with the pure greedy algorithm GRE. The results of CA and CA1 are valid for MOPS.

Algorithms CA, CA1 and CA2 were randomized by always choosing the edges and start vertices randomly. The greedy heuristics were randomized by handling the edges in a random order.

Table 1: Test graph statistics for MPS.

Graph data				The best solutions						
graph	$\|V\|$	$\|E\|$	ub	CA	CA1	CA2	GRE	GCA	GCA1	GCA2
cimi-g1	10	21	19*	11	13	13	19	19	19	19
cimi-g2	60	166	165*	88	117	117	165	165	165	165
cimi-g3	28	75	73*	38	49	49	73	73	73	73
cimi-g4	10	22	20*	13	16	16	20	20	20	20
cimi-g5	45	85	82*	59	73	73	82	82	82	82
cimi-g6	43	63	59*	42	42	42	59	59	59	59
g10	25	71	69*	36	47	47	69	69	69	69
g11	25	72	69*	36	47	47	69	69	69	69
g12	25	90	69*	36	47	47	67	66	67	67
g13	50	367	144	73	97	97	119	120	128	125
g14	50	491	144	73	97	97	127	132	134	133
g15	50	582	144*	73	97	97	133	136	138	137
g16	100	451	294	137	162	167	175	193	200	196
g17	100	742	294	147	194	196	196	224	237	229
g18	100	922	294	147	197	197	210	230	244	239
g19	150	1064	444	218	274	283	266	305	326	323
rg100.1	100	261	260	119	124	125	150	157	157	157
rg100.2	100	271	270	118	125	127	151	160	162	162
rg100.3	100	297	294	120	128	128	153	163	163	164
rg100.4	100	334	294	126	136	140	155	172	175	174
rg100.5	100	373	294	137	153	153	162	186	186	186
rg150.1	150	387	386	171	174	175	214	222	223	223
rg150.2	150	402	401	176	182	182	213	224	225	226
rg150.3	150	453	444	171	179	179	221	237	230	232
rg150.4	150	473	444	180	190	190	217	236	241	238
rg150.5	150	481	444	178	185	185	221	237	236	236
rg200.1	200	514	513	222	227	227	270	278	283	280
rg200.2	200	519	518	216	219	219	268	274	277	277
rg200.3	200	644	594	235	243	244	280	303	306	309
rg200.4	200	684	594	237	254	261	282	308	317	317
rg200.5	200	701	594	235	251	253	285	311	314	314
rg300.1	300	814	813	324	330	330	390	402	406	407
rg300.2	300	1159	894	355	376	377	412	455	461	461
rg300.3	300	1176	894	360	376	378	411	457	461	464
rg300.4	300	1474	894	389	422	426	432	497	508	509
rg300.5	300	1507	894	400	428	430	438	504	515	512
tg100.1	100	300	294*	138	188	197	292	292	294	290
tg100.3	100	324	294*	142	191	197	264	290	284	283
tg100.5	100	344	294*	138	187	197	251	262	268	272
tg100.7	100	364	294*	138	191	197	236	255	262	276
tg100.9	100	384	294*	140	189	196	226	260	263	272
tg200.1	200	604	594*	275	375	397	582	594	592	594
tg200.3	200	624	594*	279	382	397	558	579	592	586
tg200.5	200	644	594*	275	373	397	515	551	569	578
tg200.7	200	664	594*	275	372	397	492	552	578	589
tg200.9	200	684	594*	279	377	397	487	543	558	566

* Upper bound is known to be optimal.

All algorithms were written in C++ and their source codes are available as part of the the program *apptopinv* [31]. LEDA 4.3 [24] was used for the planarity test. All test runs were executed on a computer (1992 BogoMips) which has one AMD Athlon $1GHz$ processor with 256 Megabytes memory running under Linux Mandrake 8.1.

CA, CA1, CA2, GRE, GCA, GCA1 and GCA2 were repeated 100 times for graphs with no more than 100 edges and 25 times for larger graphs.

To compare the algorithms, we concentrated on studying the running time and performance differences between the algorithms. Methods and measures for the experimental analysis of the heuristics used in this work are mainly given by Golden and Stewart [12].

Running times for the algorithms were obtained by running all test runs as background processes and performing the *time* command to obtain the total running time. Finally, this total running time was divided by the number of repeats to obtain the average running time of a run.

For each algorithm, it is easy to select the best solution from all repeats for a test instance. We can then count the total number of best solutions for each algorithm, that is, an algorithm is awarded 1 point, if it obtained the best or tied the best solution for a test instance among all the algorithms.

Another measure is the total number of points for an algorithm: a heuristic is awarded p points, if it obtained the pth best solution for an instance. The average rank of an algorithm is the total number of points divided by the number of test instances.

4.2 Test graph set for MPS

Since MPS is a much studied optimization problem, there is already a wide variety of suitable test graphs. We mainly used the same test graph set as Resende and Ribeiro [35].[1]

The test graph set used in this work contains 46 graphs. Statistics for the graphs are given in Table 1. For all graphs we have listed the name of the graph (graph) and the number of vertices ($|V|$) and edges ($|E|$). Then we give the upper bound for MPS (ub). If the upper bound is known to be optimal, it is marked with a star ($\star$). Finally, the best solution found over all runs for the heuristics is given.

For graphs with an unknown optima, the upper bound was obtained by applying Euler's polyhedron formula [16]. If the number of edges was less than the bound obtained from the formula, the upper bound is the number of edges decreased by one for non-planar graphs.

The first six graphs (cimi-g1 – cimi-g6) in Table 1 were taken from the experiments of Cimikowski [7]. These graphs have relevance to applications or have their origin in other research papers. Graphs cimi-g4, cimi-g5 and cimi-g6 were introduced originally in [19], [21] and [37] respectively. Graph cimi-g6 does not contain

[1]The graphs can be downloaded from `http://www.research.att.com/~mgcr/data/planar-data.tar.gz` (April 27, 2006).

any triangles. Graphs g10 – g19 are Hamiltonian graphs constructed by Goldsmith and Takvorian [13].

Graphs rg100.1 – rg300.5 are random graphs with the number of vertices varying between 100 and 300 and the number of edges varying between 261 and 1507. Table 1 also contains graphs with a planar subgraph of maximum size (tg100.1 – tg200.9). The graphs were generated by Cimikowski [7].

4.3 Comparison of CA, CA1 and CA2 for MPS

The best solutions for the heuristics are reported in Table 1. The difference in the performance of the fast algorithms is clear. CA1 and CA2 find quite similar solutions, and they outperform CA with a clear margin. For all 46 test instances, algorithm CA2 finds the best solution, and CA1 finds the same solution as CA2 for 22 graphs. The solutions of CA are inferior to those of CA1 and CA2 for graphs that contain triangles. The only graph for which all the algorithms found the same solution was cimi-g6. The average rank of the heuristics is 2.96 for CA, 1.52 for CA1 and 1.00 for CA2. The comparison statistics are collected in Table 2.

Table 2: Comparison of the performance of the fast MPS heuristics.

	CA	CA1	CA2
Number of times heuristic is the best or tied for the best	1	22	46
Average rank	2.96	1.52	1.00

Figure 2 shows the average running times of one run for CA, CA1 and CA2 as a function of the number of edges. For graphs having more than 1600 edges, the running times are taken from the graphs used in the thickness algorithms comparison given in Section 5. Further, Figure 2 has the average running times of the greedy heuristics to illustrate the running time differences.

The running times of CA, CA1 and CA2 are less than one tenth of a second for all graphs up to 1600 edges. For the largest graphs, $r9$ with 449550 edges, used in the thickness comparison, we obtained the following average running times for the heuristics: 4.2, 5.8 and 6.5 seconds for CA, CA1 and CA2 respectively.

The running time differences between CA, CA1 and CA2 are in general very small. Only with graphs having more than 10000 edges can it be seen that CA is slightly faster than the other two algorithms. The sharp turns in the curve are the influence of the different ratios of the number of vertices and edges in a test graph. All three heuristics run faster for a sparse graph than for a dense graph with the same number of edges.

To further compare CA2 and CA we studied the relative differences of their solutions. See Figure 3 for the ratios of the poorest solutions (see [32] for these results) of CA2 and the best solutions of CA. The worst solutions by CA1 and CA2

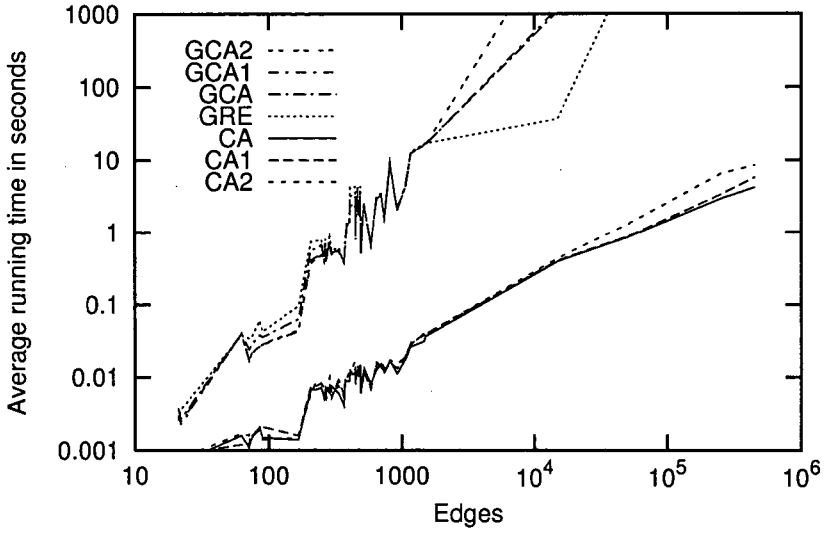


Figure 2: Average running times of MPS heuristics. Notice that the axes are logarithmic.

were always at least as good as the best solution by CA. The greatest improvement was for tg200.1 with a 1.44 times better solution when CA2 was used instead of CA. In general, the greatest improvements were obtained for graphs containing a planar subgraph of maximum size (tg100.1 – tg200.9). The solutions of CA2 were on average 20 percentages better than those of CA.

The worst case ratios of CA solutions and the optimal (marked with a dot) or the best known (marked with a circle) solution [32] are shown in Figure 4. Our experiments give evidence on the conjectured performance ratio 4/9 for the new algorithms: the solutions by CA1 and CA2 were never more than 4/9 away from the optima. For the ratios of the poorest found solutions and the optimal or the best known solution for CA2, see Figure 5. The solutions of CA1 and CA2 were never less than 0.61 and 0.65 times the optima respectively.

4.4 Comparison of GRE, GCA, GCA1 and GCA2 for MPS

It is clear that when a greedy method to add edges is applied instead of just connecting the subgraphs in Phase 2 of CA, CA1 and CA2, the solutions remain at least the same. The main questions are, thus, how much the greedy approach improves the solutions, how much longer running time is needed, and if there are graphs for which CA, CA1 or CA2 outperform GRE.

As shown in Subsection 4.3, CA2 outperformed CA1, but when the greedy algorithms were considered, GCA1 produced approximations similar to GCA2.

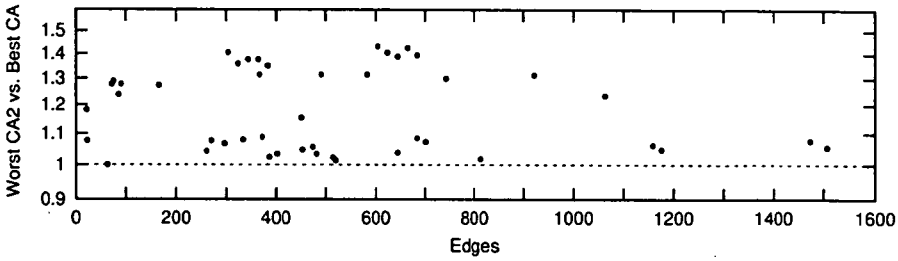


Figure 3: Ratios of the worst solutions of CA2 and the best solutions of CA.

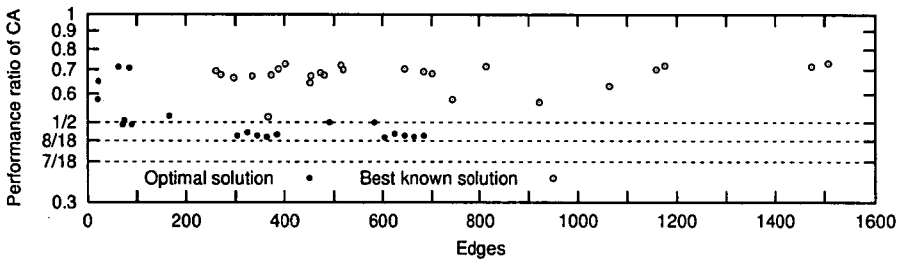


Figure 4: Ratios of the worst solutions of CA and the optimal or the best known solution.

GCA1 found the best or the tied best solution for 30 and GCA2 for 31 from 46 test instances, but the average ranks for these two heuristics were both 1.37. One explanation for the success of GCA1 is that the method of constructing a solution in Phase 1 of CA2 is greedier than that in CA1. The solution of CA2 could contain more edges than that of CA1, but it is more difficult to insert additional edges into the subgraph. GCA and GRE found the best or tied best solutions for 13 and 9 graphs and the average ranks were 2.41 and 3.41 respectively. These results are listed in Table 3.

Table 3: Comparison of the performance of the greedy MPS heuristics.

	GRE	GCA	GCA1	GCA2
Number of times heuristic is the best or tied for the best	9	13	30	31
Average rank	3.41	2.41	1.37	1.37

The running time differences of the greedy heuristics are in general very small as shown in Figure 2. For graphs with fewer than 1600 edges, GCA1 and GCA2 are

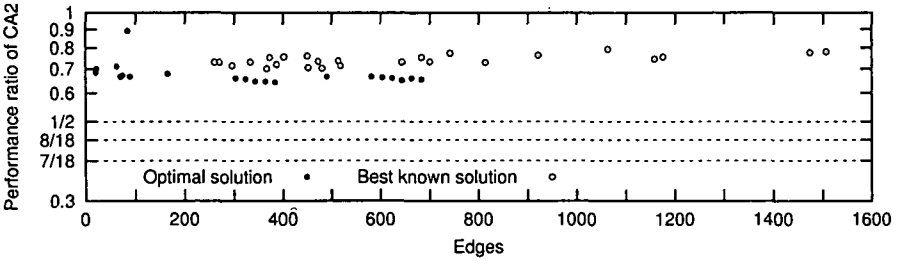


Figure 5: Ratios of the worst solutions of CA2 and the optimal or the best known solution.

the fastest heuristics and their average running time curves coincide. For the larger graphs (running times are taken from graphs used in the thickness comparison), GRE seems to be the fastest by a small margin.

We recognised test instances for which the new algorithms outperformed GRE. CA1 and CA2 found better solutions for graph g19. This shows that CA, CA1 and CA2 can find better solutions for large and sparse graphs than GRE in significantly shorter computation time. This coincides with the theoretical properties of CA1, CA2 and GRE.

CA did not find better solutions than GRE in our tests, but it has been reported that CA sometimes achieves better approximations for graphs with density varying between 0.03 and 0.15 when the algorithms are applied for MOPS [33].

GCA2 improves the solutions of CA2 on average by 30 percent. The same holds for GCA and GCA1.

5 Thickness experiments

5.1 Thickness algorithms and their implementation

For the thickness problem, we tested the extraction algorithm THICK by applying in Step 4 algorithms CA, CA1, CA2, GCA, GCA1, GCA2 and GRE. Also, we implemented an ST heuristic which in each iteration extracts the set of tree-edges found by a depth-first search. In what follows, these algorithms are simply denoted by the name of the extraction method. All these algorithms approximate thickness, but algorithms ST, CA and CA1 directly produce approximations for outerthickness.

ST, CA, CA1 and CA2 were repeated 25 times for graphs with fewer than 2000 edges, 10 times for graphs having more than 2000 edges but not more than 250000 edges and 5 times for larger graphs. Greedy heuristics for the thickness were applied only for graphs with 79800 edges or less. The number of repetitions was 25 for graphs with fewer than 2000 edges and 10 times for graphs with fewer

Table 4: Test graph statistics for the thickness problem.

Graph data				The best solutions							
graph	$\|V\|$	$\|E\|$	lb	ST	CA	CA1	CA2	GRE	GCA	GCA1	GCA2
K_{10}	10	45	3*	5	4	3	4	3	3	3	3
K_{15}	15	105	3*	8	7	5	5	4	4	4	4
K_{20}	20	190	4*	11	9	7	6	5	5	5	5
K_{30}	30	435	6*	16	13	10	9	8	7	7	7
K_{40}	40	780	7*	21	18	12	12	11	9	9	9
K_{50}	50	1225	9*	27	22	15	15	13	12	11	11
K_{60}	60	1770	11*	32	27	18	18	16	14	13	13
K_{70}	70	2415	12*	38	32	21	21	19	16	15	15
K_{80}	80	3160	14*	43	36	24	24	21	19	17	17
K_{90}	90	4005	16*	48	41	27	27	24	21	19	20
K_{100}	100	4950	17*	54	45	30	31	27	23	21	22
K_{150}	150	11175	26*	81	69	45	45	42	39	35	34
K_{200}	200	19900	34*	108	91	60	60	56	52	47	47
K_{300}	300	44850	51*	164	137	92	90	84	79	71	72
K_{400}	400	79800	67*	217	183	121	120	112	105	96	98
K_{500}	500	124750	84*	271	230	152	149	-	-	-	-
K_{600}	600	179700	101*	334	277	185	186	-	-	-	-
K_{700}	700	244650	117*	379	320	210	218	-	-	-	-
K_{800}	800	319600	134*	437	366	247	250	-	-	-	-
K_{900}	900	404550	151*	490	412	276	281	-	-	-	-
K_{1000}	1000	499500	167*	551	456	303	315	-	-	-	-
$r_{20,92}$	20	92	2*	6	4	4	4	2	3	3	3
$r_{40,311}$	40	311	3	9	7	6	5	4	5	4	4
$r_{60,556}$	60	556	4	10	8	7	7	6	5	5	5
$r_{80,939}$	80	939	5	13	10	8	8	7	7	6	6
$r_{100,1508}$	100	1508	5	17	13	10	10	9	8	8	8
r0	1000	14985	6	17	15	14	14	14	12	12	12
r1	1000	49950	17	53	43	32	31	36	30	27	27
r2	1000	99900	34	103	89	57	57	-	-	-	-
r3	1000	149850	51	160	130	82	83	-	-	-	-
r4	1000	199800	67	213	177	108	110	-	-	-	-
r5	1000	249750	84	264	224	133	138	-	-	-	-
r6	1000	299700	101	312	270	158	170	-	-	-	-
r7	1000	349650	117	363	316	184	198	-	-	-	-
r8	1000	399600	134	413	361	209	230	-	-	-	-
r9	1000	449550	151	465	411	235	261	-	-	-	-
rr1	1000	5000	2	6	6	6	6	5	5	5	5
rr2	1000	25000	9	26	22	19	19	20	17	16	16
rr3	1000	50000	17	51	43	32	31	36	30	27	27
rr4	1000	75000	26	76	65	44	44	50	43	37	37
rr5	1000	100000	34	101	88	57	57	-	-	-	-
rr6	1000	125000	42	126	112	69	70	-	-	-	-
rr7	1000	150000	51	151	136	82	83	-	-	-	-
rr8	1000	175000	59	176	160	94	96	-	-	-	-
rr9	1000	200000	67	201	184	107	108	-	-	-	-
rr10	1000	225000	76	226	209	119	121	-	-	-	-
rr11	1000	250000	84	251	233	132	134	-	-	-	-

* Lower bound is known to be optimal.

\- The algorithm is not applied for this graph.

than 5000 edges. For larger graphs only one run was performed. The comparison measures given in Section 4 also hold for the thickness experiments.

5.2 Test graph set for thickness

Algorithms for the thickness problem are compared in the literature using complete graphs, complete bipartite graphs and random graphs [8, 30, 34]. We use mainly the

same graphs as in the earlier experiments, but we have included larger complete and random graphs to the test graph set. Since CA, CA1 and CA2 behave similarly to ST for graphs without triangles, bipartite graphs are excluded from the comparison. Only ST, CA, CA1 and CA2 are run for the largest graphs. In previous works, graphs with fewer than 5000 edges have been used, while in this work the largest graph considered has 499500.

Information on the test graphs is collected in Table 4. For all graphs, we have listed the name of the graph (graph) and the number of vertices ($|V|$) and edges ($|E|$). Then the lower bound for thickness (lb) is given. If the lower bound is known to be optimal, it is marked with a star ($\star$). For graphs with unknown optimum, the lower bound is obtained by applying Euler's polyhedron formula [16].

The total number of test graphs is 47.[2] The first 21 graphs in Table 4 are complete graphs with the number of vertices varying between 10 and 1000. The next five graphs are random graphs with the number of vertices varying between 20 and 100 and the number of edges varying between 92 and 1508. Graphs r0 – r9 are random graphs with 1000 vertices and the number of edges varying between 5000 and 449500. Graphs rr1 – rr11 are random regular graphs generated with an algorithm by Steger and Wormald [36]. The degree of the vertices varies between 10 and 500.

5.3 Comparison of ST, CA, CA1 and CA2 for thickness

The total number of test graphs for the fast algorithms was 47. The best solutions for the fast heuristics and the greedy heuristics are listed in Table 4.

CA1 and CA2 outperformed ST and CA by a clear margin. CA1 and CA2 found the best solution for 39 and 27 graphs and the average ranks were 1.17 and 1.43 respectively. CA and ST respectively found only once and twice tied best solutions and their average ranks were 2.87 and 3.91. These comparison results are collected in Table 5. The reason for the relative performance of CA2 and CA1 seems to be that CA2 adds many triangles with a common edge, and therefore it constructs planar graphs with large degree. This means that the vertices that are added later get smaller degree. For large regular graphs, it seems to be a better strategy to extract subgraphs that are as regular as possible. CA1 found better solutions for large complete graphs (K_{600} – K_{1000}) and dense random graphs (r3 – r9 and rr6 – rr11). For complete graphs with fewer than 600 vertices and sparse random graphs (r1 – r2 and rr0 – rr5), CA2 obtained approximations at least as good as CA1.

The running time difference of the heuristics is clear. The relative order of the algorithms from the slowest to the fastest is CA, CA1, CA2 and ST. The explanation for the relative order of CA and CA1 (CA2) is that CA1 (CA2) extracts larger planar subgraphs and therefore the number of edges in the remaining graph decreases faster. See Figure 6 for the average running times in seconds of

[2]The random graphs can be downloaded from `http://http://www.cs.uta.fi/~tp/apptopinv` (April 27, 2006). The other graphs can be constructed by giving the command line parameters for *apptopinv* [31].

Table 5: Comparison of the performance of ST, CA, CA1 and CA2 for thickness.

	ST	CA	CA1	CA2
Number of times heuristic is the best or tied for the best	1	2	39	27
Average rank	3.91	2.87	1.17	1.43

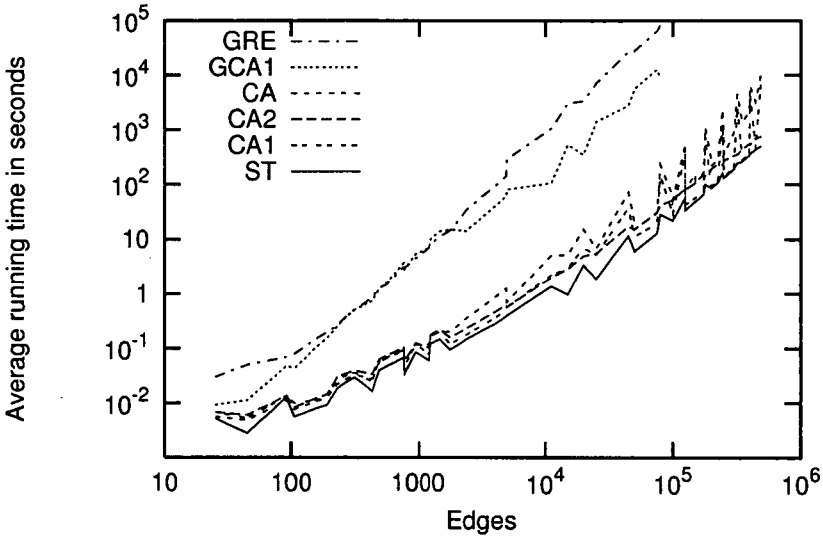


Figure 6: Average running times of ST, CA, CA1, CA2, GCA1 and GRE for thickness. Notice that the axes are logarithmic.

CA, CA1, CA2 and ST as a function of the number of the edges. The sharp turns in the curves are due to the influence of the random test graphs. The running time is higher for a random graph containing the same number of edges than for a complete graph. The average running times of GRE and GCA1 are drawn to illustrate the running time differences of the heuristics.

5.4 Comparison of GRE, GCA, GCA1 and GCA2 for thickness

The greedy algorithms achieved significantly better approximations than their non-greedy variants. For example, GCA1 decreased the solutions of CA1 to 30 percentages: CA1 got a solution of 27 for K_{90}, but the GCA1 solution was only 19. The average improvements were about 15 – 20 percentages.

GCA1 and GCA2 respectively found 24 and 21 times a best solution whereas GCA and GRE respectively found a best solution only 9 and 6 times. The average

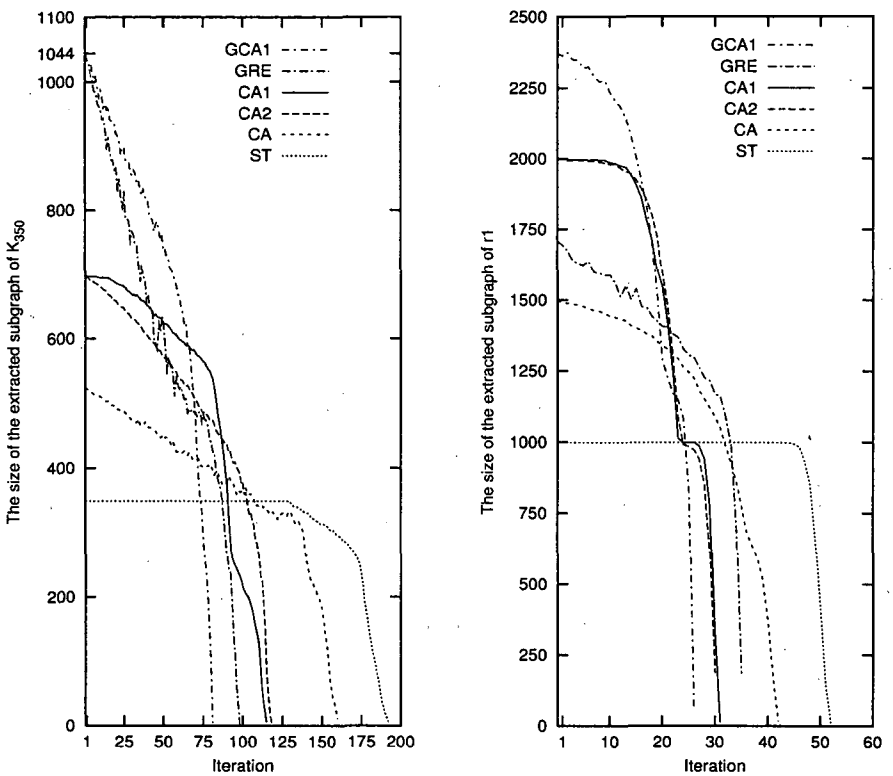


Figure 7: Sizes of the extracted planar subgraphs as a function of the number of iterations for all heuristics. In the left there are traces for K_{350} and in the right for r1.

Table 6: Comparison of the performance of GRE, GCA, GCA1 and GCA2 for thickness.

	GRE	GCA	GCA1	GCA2
Number of times heuristic is the best or tied for the best	6	9	24	21
Average rank	3.27	2.23	1.08	1.19

ranks for the heuristics were 3.27 for GRE, 2.23 for GCA, 1.08 for GCA1 and 1.19 for GCA2. These statistics are collected in Table 6.

The running times of the heuristics were very close to that of GRE, although GCA1 ran slightly faster in the case of the largest graphs. The average running times of GCA1 and GRE are illustrated in Figure 6. The running time differences between the greedy and fast heuristics are considerable: GCA1 runs 10 to 100

times slower than CA1.

There were graphs for which CA1 and CA2 outperformed GRE. CA1 and CA2 found better approximations for graphs r1, rr2, rr3 and rr4. These results are not very reliable, since for these graphs GRE was run only once. One explanation for the better performance of CA1 and CA2 against GRE for large random graphs is that the sizes of the extracted planar subgraphs are larger than they are with GRE. This is illustrated in Figure 7, where there are sample traces of ST, CA, CA1, CA2, GRE and GCA1 for a complete graph with 350 vertices (not included in the test graph set) and for r1. For K_{350}, GCA1 found the best solution and GRE found the second best solution. The extracted planar subgraphs are of maximum size in the beginning of the runs for both heuristics, but the number of edges in the extracted subgraphs of GRE decreases more rapidly than that of GCA1. The sizes of the extracted subgraphs of CA, CA1 and CA2 do not vary as much as with GRE and GCA1. The solutions of CA1 and CA2 are of the same quality, and the solutions of ST are slightly poorer. For the random graph r1 in the figure on the right, CA1 and CA2 extracts in the beginning much larger planar subgraphs than GRE. Now CA1 and CA2 yielded better approximations. The solutions of CA are worse than the solutions of GRE, but clearly better than the solutions of ST. The sizes of the extracted planar subgraphs of GCA1 were much larger at the beginning than those of CA1 and CA2 and the final solution of GCA1 was the best.

6 Conclusions

We presented two new approximation algorithms, CA1 and CA2, for the maximum planar subgraph problem and showed that the performance ratio of both algorithms is at least 7/18. The new algorithms run in linear time for bounded-degree graphs. We conjectured that the performance ratio of CA1 and CA2 is at least 4/9. All experiments performed support the conjecture. A clear goal for future research is to solve the performance ratios of CA1 and CA2. Moreover, the status of the relative performance of the better triangular cactus algorithm by Călinescu et al. [6] and the new algorithms is open.

Călinescu et al. applied their triangular cactus approach for approximating weighted MPS and MOPS. Whether our new algorithms are applicable for the weighted case remains an open question.

Acknowledgements

The author thanks the anonymous referees for their valuable comments.

References

[1] Alekseev, V.B. and Gonchakov, V.S. Thickness for arbitrary complete graphs. *Mat. Sbornik.*, 143:212–230, 1976.

[2] Beineke, L.W. and Harary, F. The thickness of the complete graph. *Can. J. Math*, 17:850–859, 1965.

[3] Beineke, L.W., Harary, F., and Moon, J.W. On the thickness of the complete bipartite graphs. *Proc. Camb. Phil. Soc.*, 60:1–5, 1964.

[4] Beyer, T., Jones, W., and Mitchell, S. Linear algorithms for isomorphism of maximal outerplanar graphs. *J. ACM*, 26(4):603–610, 1979.

[5] Călinescu, G., Fernandes, C.G., Finkler, U., and Karloff, H. A better approximation algorithm for finding planar subgraphs. *J. Algorithms*, 27(2):269–302, 1998.

[6] Călinescu, G., Fernandes, C.G., Karloff, H., and Zelikovsky, A. A new approximation algorithm for finding heavy planar subgraphs. *Algorithmica*, 36(2):179–205, 2003.

[7] Cimikowski, R. An analysis of heuristics for the maximum planar subgraph problem. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 322–331, 1995.

[8] Cimikowski, R. On heuristics for determining the thickness of a graph. *Info. Sci.*, 85:87–98, 1995.

[9] Cimikowski, R. An analysis of heuristics for graph planarization. *J. Inf. Opt. Sci.*, 18(1):49–73, 1997.

[10] Cimikowski, R. and Coppersmith, D. The sizes of maximal planar, outerplanar, and bipartite planar subgraphs. *Discr. Math.*, 149:303–309, 1996.

[11] Gabow, H.N. and Stallmann, M. Efficient algorithms for graphic matroid intersections and parity. In *Automata, Languages and Programming: 12th Colloquium*, volume 194 of *Lecture Notes in Computer Science*, pages 210–220, 1985.

[12] Golden, B.L. and Stewart, W.R. Empirical analysis of heuristics. In Lawler, E.L. and Lenstra, J.K., editors, *The Traveling Salesman Problem*, pages 207–249. John Wiley & Sons, 1985.

[13] Goldschmidt, O. and Takvorian, A. An efficient graph planarization two-phase heuristic. *Networks*, 24:69–73, 1994.

[14] Guy, R.K. and Nowakowski, R.J. The outerthickness and outercoarseness of graphs I. The complete graph & the n-cube. In Bodendiek, R. and Henns, R., editors, *Topics in Combinatorics and Graph Theory: Essays in Honour of Gerhard Ringel*, pages 297–310. Physica-Verlag, 1990.

[15] Guy, R.K. and Nowakowski, R.J. The outerthickness and outercoarseness of graphs II. The complete bipartite graph. In Bodendiek, R., editor, *Contemporary Methods in Graph Theory*, pages 313–322. B.I. Wissenchaftsverlag, 1990.

[16] Harary, F. *Graph Theory.* Addison-Wesley, 1971.

[17] Hasan, M. and Osman, I.H. Local search algorithms for the maximal planar layout problem. *Int. Trans. Oper. Res.*, 2(1):89–106, 1995.

[18] Hopcroft, J. and Tarjan, R.E. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[19] Jayakumar, R., Thulasiraman, K., and Swamy, M.N.S. $O(n^2)$ algorithms for graph planarization. *IEEE Trans. Comp.-Aided Design*, 8(3):257–267, 1989.

[20] Jünger, M. and Mutzel, P. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 16:33–59, 1996.

[21] Kant, G. An $O(n^2)$ maximal planarization algorithm based on PQ-trees. Technical Report RUU-CS-92-03, Utrecht University, Department of Computer Science, 1992.

[22] Kant, G. Augmenting outerplanar graphs. *J. Algorithms*, 21:1–25, 1996.

[23] Kleinert, M. Die Dicke des n-dimensionale Würfel-Graphen. *J. Comb. Theory*, 3:10–15, 1967.

[24] LEDA version 4.3 (commercial). Available at `http://www.algorithmic-solutions.com`.

[25] Liebers, A. Planarizing graphs - a survey and annotated bibliography. *JGAA*, 5(1):1–74, 2001.

[26] Liu, P.C. and Geldmacher, R. On the deletion of nonplanar edges of a graph. In *Proceedings of the 10th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 727–738, 1977.

[27] Lovász, L. and Plummer, M.D. *Matching Theory.* Elsevier, 1986.

[28] Mansfield, A. Determining the thickness of graphs is NP-hard. *Math. Proc. Camb. Phil. Soc.*, 93:9–23, 1983.

[29] Mitchell, S.L. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Inf. Proc. Lett.*, 9(5):177–189, 1979.

[30] Mutzel, P., Odenthal, T., and Scharbrodt, M. The thickness of graphs: a survey. *Graphs Comb.*, 14:59–73, 1998.

[31] Poranen, T. Apptopinv - user's guide. Technical Report A-2003-3, University of Tampere, Department of Computer Sciences, 2003.

[32] Poranen, T. *Approximation Algorithms for Some Topological Invariants of Graphs.* PhD thesis, University of Tampere, 2004.

[33] Poranen, T. Heuristics for the maximum outerplanar subgraph problem. *J. Heuristics*, 11:59–88, 2005.

[34] Poranen, T. A simulated annealing algorithm for determining the thickness of a graph. *Info. Sci.*, 172:155–172, 2005.

[35] Resende, M.G.C. and Ribeiro, C.C. A GRASP for graph planarization. *Networks*, 29:173–189, 1997.

[36] Steger, A. and Wormald, N.C. Generating random regular graphs quickly. *Comb. Probab. Comput.*, 8:377–396, 1999.

[37] Tamassia, R., Di Battista, G., and Batini, C. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, 18(1):61–79, 1988.

[38] Yannakakis, M. Node- and edge-deletion NP-complete problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 253–264, 1978.

Received 10th January 2005

Acta Cybernetica **18** (2008) 529–556.

Keys and Armstrong Databases in Trees with Restructuring

Attila Sali* and Klaus-Dieter Schewe†

Abstract

The definition of keys, antikeys, Armstrong-instances are extended to complex values in the presence of several constructors. These include tuple, list, set and a union constructor. Nested data structures are built using the various constructors in a tree-like fashion. The union constructor complicates all results and proofs significantly. The reason for this is that it comes along with non-trivial restructuring rules. Also, so-called counter attributes need to be introduced. It is shown that keys can be identified with closed sets of subattributes under a certain closure operator. Minimal keys correspond to closed sets minimal under set-wise containment. The existence of Armstrong databases for given minimal key systems is investigated. A sufficient condition is given and some necessary conditions are also exhibited. Weak keys can be obtained if functional dependency is replaced by weak functional dependency in the definition. It is shown, that this leads to the same concept. Strong keys are defined as principal ideals in the subattribute lattice. Characterization of antikeys for strong keys is given. Some numerical necessary conditions for the existence of Armstrong databases in case of degenerate keys are shown. This leads to the theory of bounded domain attributes. The complexity of the problem is shown through several examples.

1 Introduction

The relational datamodel gave rise to theoretical research in several directions. Dependency structures were investigated as first-order logical sentences that are supposed to hold for all database instances [3]. On the other hand, their combinatorial investigations were fruitful resulting in nice problems, concepts, even as far topics as design and coding theory [8, 9, 12, 5].

The relational model has been extended or generalized to nested relational model [19], object oriented models [23], and object-relational models. The important structures of all these were captured by the higher-order Entity-Relationship

*Alfréd Rényi Institute of Mathematics, Hungarian Academy of Sciences, Budapest, P.O.B.127, H-1364 Hungary, E-mail: sali@renyi.hu

†Massey University, Information Science Research Centre & Department of Information Systems, Private Bag 11 222, Palmerston North, New Zealand, E-mail: k.d.schewe@massey.ac.nz

model [24, 25]. The semi-structured data and XML treated in [1] can also be considered as an object-oriented model.

The major new structure in all these models is the introduction of constructors that allow us to form complex data values from simpler ones. The dependencies of the relational model can be generalized to these higher-order models, and the axiomatization of certain dependencies was carried out in [13, 15, 16, 18, 14]. On the other hand, the induced combinatorial structures have not been investigated thoroughly yet. It is important from the point of view of schema design, to identify what kind of attributes can form *key systems*. The aim of the present paper is to continue the investigations started in [20, 21], thus generalizing the work of [6, 7, 8].

In Section 2 the necessary definitions are recalled. In Section 3 keys and antikeys are defined and it is shown that they correspond to *closed subsets* of the subattribute lattice under a certain closure operator. This is in sharp contrast with the Relational Datamodel (RDM), where *any* subset of the attributes could be a key for an appropriate system of functional dependencies. Section 4 deals with question of existence of Armstrong-instances. This question was first investigated by Armstrong [4] and Demetrovics [6] for functional dependencies in the RDM. Later Fagin [10] gave a necessary and sufficient condition for general dependencies in the relational context. Fagin's results are quite general, if types of dependencies are considered, however, they are only valid for relational databases, as the conditions he gave depend on direct products of relations.

In the present paper we treat functional dependencies in the higher order datamodel and a sufficient condition is given for the existence of Armstrong instance. In addition, we illustrate the complexity of the problem through several examples. Section 5 is devoted to strong keys, that are the closest analogs of keys in the RDM. Finally, Section 6 contains some inequalities of parameters that give necessary conditions for the existence of Armstrong-instances.

2 Preliminaries

In this section we define our model of nested attributes, which covers the gist of higher-order datamodels including XML. In particular, we investigate the structure of the set $\mathcal{S}(X)$ of subattributes of a given nested attribute X. We show that we obtain a non-distributive Brouwer algebra, i.e. a non-distributive lattice with relative pseudo-complements.

2.1 Nested Attributes

We start with a definition of simple attributes and values for them.

Definition 1. A *universe* is a finite set $\mathcal{U}$ together with domains (i.e. sets of values) $dom(A)$ for all $A \in \mathcal{U}$. The elements of $\mathcal{U}$ are called *simple attributes*.

For the relational model a universe was enough, as a relation schema could be defined by a subset $R \subseteq \mathcal{U}$. For higher-order datamodels, however, we need nested

attributes. In the following definition we use a set $\mathcal{L}$ of labels, and tacitly assume that the symbol λ is neither a simple attribute nor a label, i.e. $\lambda \notin \mathcal{U} \cup \mathcal{L}$, and that simple attributes and labels are pairwise different, i.e. $\mathcal{U} \cap \mathcal{L} = \emptyset$.

Definition 2. Let $\mathcal{U}$ be a universe and $\mathcal{L}$ a set of labels. The set $\mathcal{N}$ of *nested attributes* (over $\mathcal{U}$ and $\mathcal{L}$) is the smallest set with $\lambda \in \mathcal{N}$, $\mathcal{U} \subseteq \mathcal{N}$, and satisfying the following properties:

- for $X \in \mathcal{L}$ and $X'_1, \ldots, X'_n \in \mathcal{N}$ we have $X(X'_1, \ldots, X'_n) \in \mathcal{N}$;
- for $X \in \mathcal{L}$ and $X' \in \mathcal{N}$ we have $X\{X'\} \in \mathcal{N}$, $X[X'] \in \mathcal{N}$, and $X\langle X'\rangle \in \mathcal{N}$;
- for $X_1, \ldots, X_n \in \mathcal{L}$ and $X'_1, \ldots, X'_n \in \mathcal{N}$ we have $X_1(X'_1) \oplus \cdots \oplus X_n(X'_n) \in \mathcal{N}$.

We call λ a *null attribute*, $X(X'_1, \ldots, X'_n)$ a *record attribute*, $X\{X'\}$ a *set attribute*, $X[X']$ a *list attribute*, $X\langle X'\rangle$ a *multiset attribute* and $X_1(X'_1) \oplus \cdots \oplus X_n(X'_n)$ a *union attribute*. As record and set attributes have a unique leading label, say X, we often write simply X to denote the attribute.

We can now extend the association *dom* from simple to nested attributes, i.e. for each $X \in \mathcal{N}$ we will define a set of values $dom(X)$.

Definition 3. For each nested attribute $X \in \mathcal{N}$ we get a *domain* $dom(X)$ as follows:

- $dom(\lambda) = \{\top\}$;
- $dom(X(X'_1, \ldots, X'_n)) = \{(v_1, \ldots, v_n) \mid v_i \in dom(X'_i) \text{ for } i = 1, \ldots, n\}$;
- $dom(X\{X'\}) = \{\{v_1, \ldots, v_k\} \mid k \in \mathbb{N} \text{ and } v_i \in dom(X') \text{ for } i = 1, \ldots, k\}$, i.e. each element in $dom(X\{X'\})$ is a finite set with (pairwise different) elements in $dom(X')$;
- $dom(X[X']) = \{[v_1, \ldots, v_k] \mid k \in \mathbb{N} \text{ and } v_i \in dom(X') \text{ for } i = 1, \ldots, k\}$, i.e. each element in $dom(X[X'])$ is a finite (ordered) list with (not necessarily different) elements in $dom(X')$;
- $dom(X\langle X'\rangle) = \{\langle v_1, \ldots, v_k\rangle \mid k \in \mathbb{N} \text{ and } v_i \in dom(X') \text{ for } i = 1, \ldots, k\}$, i.e. each element in $dom(X\langle X'\rangle)$ is a finite multiset with elements in $dom(X')$, or in other words each $v \in dom(X')$ has a *multiplicity* $m(v) \in \mathbb{N}$ in a value in $dom(X\langle X'\rangle)$;
- $dom(X_1(X'_1) \oplus \cdots \oplus X_n(X'_n)) = \{(X_i : v_i) \mid v_i \in dom(X'_i) \text{ for } i = 1, \ldots, n\}$.

Note that the relational model is covered, if only the tuple constructor is used. Thus, instead of a relation schema R we will now consider a nested attribute X, assuming that the universe $\mathcal{U}$ and the set of labels $\mathcal{L}$ are fixed. Instead of an R-relation r we will consider a finite set $r \subseteq dom(X)$. An element of r is called a *tuple* or *complex value*. The following example includes several constructors.

Example 4. The nested attribute *Concert* allows to define an instance that contains data of a (rock-)concert.

Concert(Band(Bname(BandName),Members{Musician(
Name(MusicianName),Role(Instrument(InstrumentName)⊕Vocal(Voice)))}),
Played(Songs[SongTitle]),Evaluation(Grade)).

Here BandName, MusicianName, InstrumentName, Voice, SongTitle and Grade are simple attributes, while Concert, Band, Bname,Members, Musician, Name, Role, Instrument, Vocal, Played and Evaluation are labels. An element of the domain of nested attribute *Concert* could be the following tuple:

(∅,{ (Greg Howe,(Instrument:Guitar)),
(Victor Wooten,(Instrument:Bassguitar)),
(Dennis Chambers,(Instrument:Drums)),}
[Tease, Contigo, Proto Cosmos], 10).

Note that this trio of jazz musicians plays under no specific band name.

In the following, we will need a bit more caution regarding syntax in order to avoid ambiguity. For this we mark the set label in an attribute of the form $X\{X_1(X_1') \oplus \cdots \oplus X_n(X_n')\}$ to indicate the inner union attribute, i.e. we should use $X_{\{1,\ldots,n\}}$ (or even $X_{\{X_1,\ldots,X_n\}}$) instead of X. As long as we are not dealing with subattributes of the form $X_{\{1,\ldots,k\}}\{\lambda\}$, the additional index does not add any information and thus can be omitted to increase readability. The same applies to the multiset- and the list-constructor. The reason for introducing these indices will become apparent after Definition 6.

2.2 Subattributes

In the dependency theory for the relational model we considered the powerset $\mathcal{P}(R)$ for a relation schema R. $\mathcal{P}(R)$ is a Boolean algebra with order $\subseteq$, intersection $\cap$, union $\cup$ and the difference $-$.

We will generalize these operations for nested attributes starting with a partial order $\geq$. However, this partial order will be defined on equivalence classes of attributes. We will identify nested attributes, if we can identify their domains.

In the relational model a functional dependency $X \to Y$ for $X, Y \subseteq R \subseteq \mathcal{U}$ is satisfied by an R-relation r iff any two tuples $t_1, t_2 \in r$ that coincide on all the attributes in X also coincide on the attributes in Y. Crucial to this definition is that we can project R-tuples to subsets of attributes.

Therefore, in order to define FDs on a nested attribute $X \in \mathcal{N}$ we need a notion of subattribute. For this we define a partial order $\geq$ on nested attributes in such a way that whenever $X \geq Y$ holds, we obtain a canonical projection $\pi_Y^X : dom(X) \to dom(Y)$. However, this partial order has to be defined on equivalence classes of attributes, as some domains may be identified.

Definition 5. $\equiv$ is the smallest *equivalence relation* on $\mathcal{N}$ satisfying the following properties:

- $\lambda \equiv X()$;
- $X(X'_1, \ldots, X'_n) \equiv X(X'_1, \ldots, X'_n, \lambda)$;
- $X(X'_1, \ldots, X'_n) \equiv X(X'_{\sigma(1)}, \ldots, X'_{\sigma(n)})$ for any permutation $\sigma \in \mathbf{S}_n$;
- $X_1(X'_1) \oplus \cdots \oplus X_n(X'_n) \equiv X_{\sigma(1)}(X'_{\sigma(1)}) \oplus \cdots \oplus X_{\sigma(n)}(X'_{\sigma(n)})$ for any permutation $\sigma \in \mathbf{S}_n$;
- $X(X'_1, \ldots, X'_n) \equiv X(Y_1, \ldots, Y_n)$ if $X'_i \equiv Y_i$ for all $i = 1, \ldots, n$;
- $X_1(X'_1) \oplus \cdots \oplus X_n(X'_n) \equiv X_1(Y_1) \oplus \cdots \oplus X_n(Y_n)$ if $X'_i \equiv Y_i$ for all $i = 1, \ldots, n$;
- $X\{X'\} \equiv X\{Y\}$ iff $X' \equiv Y$;
- $X[X'] \equiv X[Y]$ iff $X' \equiv Y$;
- $X\langle X' \rangle \equiv X\langle Y \rangle$ iff $X' \equiv Y$;
- $X(X'_1, \ldots, Y_1(Y'_1) \oplus \cdots \oplus Y_m(Y'_m), \ldots, X'_n) \equiv Y_1(X'_1, \ldots, Y'_1, \ldots, X'_n) \oplus \cdots \oplus Y_m(X'_1, \ldots, Y'_m, \ldots, X'_n)$;
- $X\{X_1(X'_1) \oplus \cdots \oplus X_n(X'_n)\} \equiv X(X_1\{X'_1\}, \ldots, X_n\{X'_n\})$;
- $X\langle X_1(X'_1) \oplus \cdots \oplus X_n(X'_n)\rangle \equiv X(X_1\langle X'_1\rangle, \ldots, X_n\langle X'_n\rangle)$.

Basically, the first four cases in this equivalence definition state that λ in record attributes can be added or removed, and that order in record and union attributes does not matter. The last three cases in Definition 5 cover restructuring rules, two of which were already introduced in [2]. Obviously, if we have a set of labeled elements with up to n different labels, we can split this set into n subsets, each of which contains just the elements with a particular label, and the union of these sets is the original set. The same holds for multisets. Of course, we can also split a list of labeled elements into lists containing only elements with the same label, thereby preserving the order, but in this case we cannot invert the splitting and thus cannot claim an equivalence.

In the following we identify $\mathcal{N}$ with the set $\mathcal{N}/_{\equiv}$ of equivalence classes. In particular, we will write $=$ instead of $\equiv$, and in the following definition we should say that Y is a subattribute of X iff $\tilde{X} \geq \tilde{Y}$ holds for some $\tilde{X} \equiv X$ and $\tilde{Y} \equiv Y$.

Definition 6. For $X, Y \in \mathcal{N}$ we say that Y is a *subattribute* of X, iff $X \geq Y$ holds, where $\geq$ is the smallest partial order on $\mathcal{N}/_{\equiv}$ satisfying the following properties:

- $X \geq \lambda$ for all $X \in \mathcal{N}$;
- $X(Y_1, \ldots, Y_n) \geq X(X'_{\sigma(1)}, \ldots, X'_{\sigma(m)})$ for some injective $\sigma : \{1, \ldots, m\} \to \{1, \ldots, n\}$ and $Y_{\sigma(i)} \geq X'_{\sigma(i)}$ for all $i = 1, \ldots, m$;
- $X_1(Y_1) \oplus \cdots \oplus X_n(Y_n) \geq X_{\sigma(1)}(X'_{\sigma(1)}) \oplus \cdots \oplus X_{\sigma(n)}(X'_{\sigma(n)})$ for some permutation $\sigma \in \mathbf{S}_n$ and $Y_i \geq X'_i$ for all $i = 1, \ldots, n$;

- $X\{Y\} \geq X\{X'\}$ iff $Y \geq X'$;
- $X[Y] \geq X[X']$ iff $Y \geq X'$;
- $X\langle Y\rangle \geq X\langle X'\rangle$ iff $Y \geq X'$;
- $X_{\{1,\ldots,n\}}[X_1(X_1') \oplus \cdots \oplus X_n(X_n')] \geq X(X_1[X_1'], \ldots, X_n[X_n'])$;
- $X_{\{1,\ldots,k\}}[X_1(X_1') \oplus \cdots \oplus X_k(X_k')] \geq X_{\{1,\ldots,\ell\}}[X_1(X_1') \oplus \cdots \oplus X_\ell(X_\ell')]$ for $k \geq \ell$;
- $X(X_{i_1}\{\lambda\}, \ldots, X_{i_k}\{\lambda\}) \geq X_{\{i_1,\ldots,i_k\}}\{\lambda\}$;
- $X(X_{i_1}\langle\lambda\rangle, \ldots, X_{i_k}\langle\lambda\rangle) \geq X_{\{i_1,\ldots,i_k\}}\langle\lambda\rangle$;
- $X(X_{i_1}[\lambda], \ldots, X_{i_k}[\lambda]) \geq X_{\{i_1,\ldots,i_k\}}[\lambda]$.

Attributes of types $X_{\{i_1,\ldots,i_k\}}\{\lambda\}$, $X_{\{i_1,\ldots,i_k\}}\langle\lambda\rangle$ and $X_{\{i_1,\ldots,i_k\}}[\lambda]$ are called *counter attributes.*

Note that the last four cases in Definition 6 cover further restructuring rules due to the union constructor. Obviously, if we are given a list of elements labeled with $X_1, \ldots, X_n$, we can take the individual sublists – preserving the order – that contain only those elements labeled by X_i and build the tuple of these lists. In this case we can turn the label into a label for the whole sublist. This explains the first of the last four subattribute relationships.

For the other restructuring rules we have to add a little remark on notation here explaining why we use additional indices. As we identify $X\{X_1(X_1') \oplus \cdots \oplus X_n(X_n')\}$ with $X(X_1\{X_1'\}, \ldots, X_n\{X_n'\})$, we obtain subattributes of the form $X(X_{i_1}\{X_{i_1}'\}, \ldots, X_{i_k}\{X_{i_k}'\})$ for each subset $I = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$. However, restructuring requires some care with labels. If we simply reused the label X in the third last property in Definition 6, we would obtain

$$\begin{aligned} X\{X_1(X_1') \oplus X_2(X_2')\} \equiv & X(X_1\{X_1'\}, X_2\{X_2'\}) \geq \\ & X(X_1\{X_1'\}) \geq X(X_1\{\lambda\}) \geq X\{\lambda\}. \end{aligned}$$

However, the last step here is wrong, as the left hand side is an indicator for the subset containing the elements with label X_1 being empty or not, whereas the right hand side is the corresponding indicator for the whole set, i.e. elements with labels X_1 or X_2. No such mapping can be claimed. However, if we mark the set label in an attribute of the form $X\{X_1(X_1') \oplus \cdots \oplus X_n(X_n')\}$ to indicate the inner union attribute, the ambiguity problem disappears.

Further note that due to the restructuring rules in Definitions 5 and 6 we may have the case that a record attribute is a subattribute of a set attribute and vice versa. This cannot be the case, if the union-constructor is absent. However, the presence of the restructuring rules allows us to assume that the union-constructor only appears inside a set-constructor or as the outermost constructor. This will be frequently exploited in our proofs.

Obviously, $X \geq Y$ induces a projection map $\pi_Y^X : dom(X) \to dom(Y)$. For $X \equiv Y$ we have $X \geq Y$ and $Y \geq X$ and the projection maps π_Y^X and π_X^Y are inverse to each other.

Example 7. Let $X = \text{Balls}\{\text{red}(\text{Number}) \oplus \text{blue}(\text{Number}) \oplus \text{green}(\text{Number})\}$. A complex value in $dom(X)$ represents a set of coloured balls carrying numbers, with the colours red, green and blue being available. Examples of such values are $v_1 = \{(\text{red} : 11), (\text{red} : 12), (\text{green} : 11), (\text{blue} : 6), (\text{blue} : 1)\}$, $v_2 = \{(\text{red} : 5), (\text{red} : 7), (\text{blue} : 3)\}$, and $v_3 = \{(\text{green} : 8)\}$.

Counter subattributes of X are $X_1 = \text{Balls}_{red,green}\{\lambda\}$, $X_2 = \text{Balls}_{green}\{\lambda\}$, and $X_3 = \text{Balls}_{blue}\{\lambda\}$. Projecting a value $v \in dom(X)$ to X_1 would give a non-empty set $\{\top\}$ iff v contains red or green balls. Analogously, the projection to X_2 or X_3 results in $\{\top\}$ iff v contains green or blue balls, respectively. For instance, we obtain

$$\begin{array}{lll} \pi^X_{X_1}(v_1) = \{\top\} & \pi^X_{X_2}(v_1) = \{\top\} & \pi^X_{X_3}(v_1) = \{\top\} \\ \pi^X_{X_1}(v_2) = \{\top\} & \pi^X_{X_2}(v_2) = \emptyset & \pi^X_{X_3}(v_2) = \{\top\} \\ \pi^X_{X_1}(v_3) = \{\top\} & \pi^X_{X_2}(v_3) = \{\top\} & \pi^X_{X_3}(v_3) = \emptyset \end{array}$$

We use the notation $\mathcal{S}(X) = \{Z \in \mathcal{N} \mid X \geq Z\}$ to denote the *set of subattributes* of a nested attribute X. Figure 2 shows the subattributes of $X\{X_1(A) \oplus X_2(B) \oplus X_3(C)\} = X(X_1\{A\}, X_2\{B\}, X_3\{C\})$ together with the relation $\geq$ on them.

Note that the subattribute $X\{\lambda\}$ would not occur, if we only considered the record-structure, whereas other subattributes such as $X(X_1\{\lambda\})$ would not occur, if we only considered the set-structure. This is a direct consequence of the restructuring rules.

Example 8. Consider the following subattributes of the nested attribute *Concert* of Example 4. Subattribute

$$\text{Concert}(\text{Band}(\text{Members}\{\text{Musician}(\text{Name}(\text{MusicianName}))\}))$$

represents the set of names of musicians performing at the concert. The projection of the tuple shown in Example 4 to this subattribute is the following complex value:

$$(\{(\texttt{Greg Howe}),(\texttt{Victor Wooten}),(\texttt{Dennis Chambers})\}).$$

The subattribute $\text{Concert}(\text{Played}(\text{Songs}[\lambda]))$ shows the number of songs played during the concert. The projection of the tuple of Example 4 to this subattribute is the tuple $([\top, \top, \top])$ showing that three songs were played. Finally, subattribute $\text{Concert}(\text{Band}(\text{Members}\{\text{Musician}(\text{Role}_{\{\text{Vocal}\}}(\lambda))\}))$ shows whether a singer performed at the concert. Projecting the tuple of Example 4 to this subattribute the tuple $(\{(\emptyset)\})$ is obtained that shows that only instrumental music was played.

Let us now investigate the structure of $\mathcal{S}(X)$. We obtain a non-distributive lattice with relative pseudo-complements.

Definition 9. Let $\mathcal{L}$ be a lattice with zero and one, partial order $\leq$, join $\sqcup$ and meet $\sqcap$. $\mathcal{L}$ has *relative pseudo-complements* iff for all $Y, Z \in \mathcal{L}$ the infimum $Y \leftarrow Z = \bigsqcap\{U \mid U \sqcup Y \geq Z\}$ exists. Then $Y \leftarrow 1$ (1 being the one in $\mathcal{L}$) is called the *relative complement* of Y.

If we have distributivity in addition, we call $\mathcal{L}$ a *Brouwer algebra.* In this case the relative pseudo-complements satisfy $U \geq (Y \leftarrow Z)$ iff $(U \sqcup Y \geq Z)$, but if we do not have distributivity this property may be violated though relative pseudo-complements exist.

Proposition 10. *The set $\mathcal{S}(X)$ of subattributes carries the structure of a lattice with zero and one and relative pseudo-complements, where the order $\geq$ is as defined in Definition 6, and λ and X are the zero and one, respectively.*

It is easy to determine explicit inductive definitions of the operations $\sqcap$ (meet), $\sqcup$ (join) and $\leftarrow$ (relative pseudo-complement). This can be done by boring technical verification of the properties of meets, joins and relative pseudo-complements and is therefore omitted here.

Example 11. Let $X = X\{X_1(A) \oplus X_2(B)\}$ with $\mathcal{S}(X)$, as shown in Figure 1. Furthermore let $Y_1 = X_{\{1,2\}}\{\lambda\}$, $Y_2 = X(X_2\{B\})$, and $Z = X(X_1\{A\})$. Note that $\sqcup$ is the least common upper bound, while $\sqcap$ is the largest common lower bound in the subattribute poset. Then we have

$$\begin{gathered} Z \sqcap (Y_1 \sqcup Y_2) = X(X_1\{A\}) \sqcap (X_{\{1,2\}}\{\lambda\} \sqcup X(X_2\{B\})) = \\ X(X_1\{A\}) \sqcap X(X_1\{\lambda\}, X_2\{B\}) = X(X_1\{\lambda\}) \neq \lambda = \lambda \sqcup \lambda = \\ (X(X_1\{A\}) \sqcap X\{\lambda\}) \sqcup (X(X_1\{A\}) \sqcap X(X_2\{B\})) = (Z \sqcap Y_1) \sqcup (Z \sqcap Y_2)\,. \end{gathered}$$

This shows that $\mathcal{S}(X)$ in general is not a distributive lattice. Furthermore, $Y' \sqcup Z \geq Y_1$ holds for all Y' except λ, $X(X_1\{\lambda\})$ and $X(X_1\{A\})$. So $Z \leftarrow Y_1 = \lambda$, but not all $Y' \geq \lambda$ satisfy $Y' \sqcup Z \geq Y_1$.

2.3 Functional Dependencies

Let us now define functional and weak functional dependencies on $\mathcal{S}(X)$ and derive some sound derivation rules. The first thought would be to consider single nested attributes, as in the RDM $\sqcup$ corresponds to the union $\cup$, and $\sqcap$ to the intersection $\cap$. However, if we treat functional dependencies in this way, we cannot obtain a generalization of the extension rule. Therefore, we have to consider sets of subattributes.

Definition 12. Let $X \in \mathcal{N}$. A *functional dependency* (FD) on $\mathcal{S}(X)$ is an expression $\mathcal{Y} \rightarrow \mathcal{Z}$ with $\mathcal{Y}, \mathcal{Z} \subseteq \mathcal{S}(X)$. A *weak functional dependency* (wFD) on $\mathcal{S}(X)$ is an expression $\{\!\{\mathcal{Y}_i \rightarrow \mathcal{Z}_i \mid i \in I\}\!\}$ with an index set I and $\mathcal{Y}_i, \mathcal{Z}_i \subseteq \mathcal{S}(X)$.

In the following we consider finite sets $r \subseteq dom(X)$, which we will call simply *instances* of X.

Definition 13. Let r be an instance of X. We say that r *satisfies the FD* $\mathcal{Y} \rightarrow \mathcal{Z}$ on $\mathcal{S}(X)$ (notation: $r \models \mathcal{Y} \rightarrow \mathcal{Z}$) iff for all $t_1, t_2 \in r$ with $\pi^X_Y(t_1) = \pi^X_Y(t_2)$ for all $Y \in \mathcal{Y}$ we also have $\pi^X_Z(t_1) = \pi^X_Z(t_2)$ for all $Z \in \mathcal{Z}$.

r *satisfies the wFD* $\{\!\{\mathcal{Y}_i \rightarrow \mathcal{Z}_i \mid i \in I\}\!\}$ on $\mathcal{S}(X)$ (notation: $r \models \{\!\{\mathcal{Y}_i \rightarrow \mathcal{Z}_i \mid i \in I\}\!\}$) iff for all $t_1, t_2 \in r$ there is some $i \in I$ with $\{t_1, t_2\} \models \mathcal{Y}_i \rightarrow \mathcal{Z}_i$.

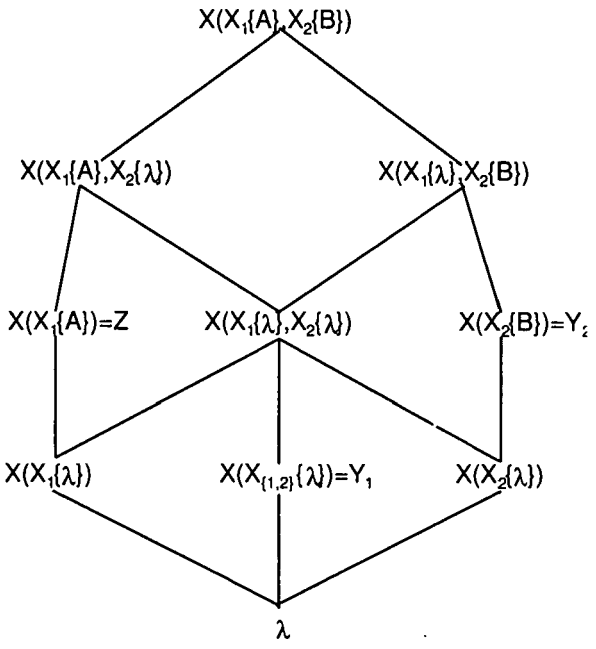


Figure 1: The lattice $\mathcal{S}(X\{X_1(A) \oplus X_2(B)\})$

According to this definition we identify a wfD $\{\!\{\mathcal{Y} \to \mathcal{Z}\}\!\}$, i.e. the index set contains exactly one element, with the "ordinary" FD $\mathcal{Y} \to \mathcal{Z}$.

2.4 Coincidence Ideals

The study of FDs and wFDs depends heavily on the notion of "coincidence ideal", i.e. sets of subattributes, on which two complex values coincide. For our purposes in this paper it is sufficient to take this as the definition.

In the following we investigate sets of subattributes, on which two complex values coincide. It is rather easy to see that these turn out to be ideals in the lattice $\mathcal{S}(X)$, i.e. they are non-empty and downward-closed. Therefore, we will call them *coincidence ideals*. However, there are many other properties that hold for coincidence ideals.

Definition 14. Two subattributes $Y, Z \in \mathcal{S}(X)$ are called *reconcilable* iff one of the following holds:

1. $Y \geq Z$ or $Z \geq Y$;
2. $X = X[X']$, $Y = X[Y']$, $Z = X[Z']$ and $Y', Z' \in \mathcal{S}(X')$ are reconcilable;
3. $X = X(X_1, \ldots, X_n)$, $Y = X(Y_1, \ldots, Y_n)$, $Z = X(Z_1, \ldots, Z_n)$ and $Y_i, Z_i \in \mathcal{S}(X_i)$ are reconcilable for all $i = 1, \ldots, n$;

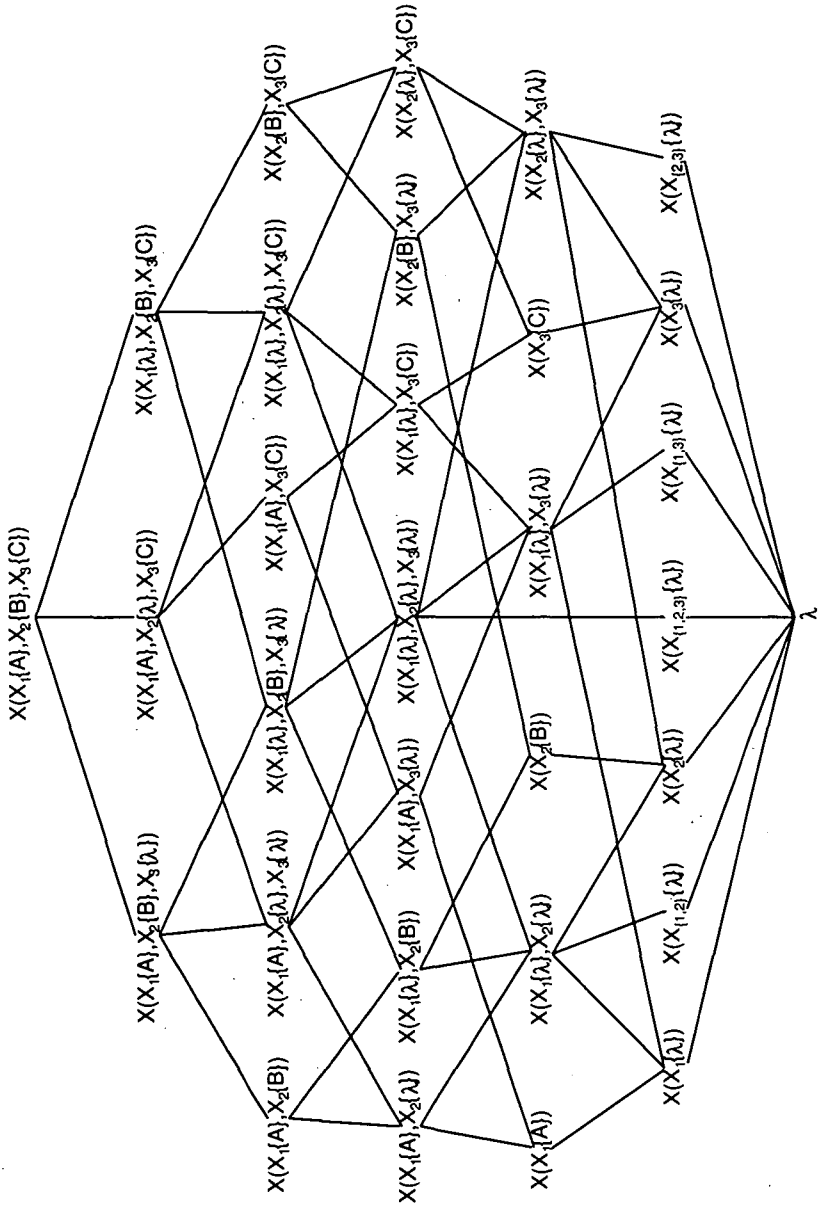


Figure 2: The lattice $\mathcal{S}(X\{X_1(A) \oplus X_2(B) \oplus X_3(C)\})$

4. $X = X_1(X_1') \oplus \cdots \oplus X_n(X_n')$, $Y = X_1(Y_1') \oplus \cdots \oplus X_n(Y_n')$, $Z = X_1(Z_1') \oplus \cdots \oplus X_n(Z_n')$ and $Y_i', Z_i' \in \mathcal{S}(X_i')$ are reconcilable for all $i = 1, \ldots, n$;

5. $X = X[X_1(X_1') \oplus \cdots \oplus X_n(X_n')]$, $Y = X(Y_1, \ldots, Y_n)$ with $Y_i = X_i[Y_i']$ or $Y_i = \lambda = Y_i'$, $Z = X[X_1(Z_1') \oplus \cdots \oplus X_n(Z_n')]$, and Y_i', Z_i' are reconcilable for all $i = 1, \ldots, n$.

Note that for the set- and multiset-constructor we can only obtain reconcilability for subattributes in a $\geq$-relation.

Example 15. Consider $\mathcal{S}(X\{X_1(A) \oplus X_2(B) \oplus X_3(C)\})$ shown on Figure 2. Here the subattributes $X(X_1\{A\}, X_2\{B\})$ and $X(X_1\{\lambda\}, X_2\{\lambda\}, X_3\{C\}$ are reconcilable. Indeed, $X(X_1\{A\}, X_2\{B\}) \equiv X(X_1\{A\}, X_2\{B\}, \lambda)$ and $X_1\{A\} \geq X_1\{\lambda\}$, $X_2\{B\} \geq X_2\{\lambda\}$, $\lambda \leq X_3\{C\}$, thus the components of the two subattributes are pairwise comparable in the subattribute lattice $\mathcal{S}(X)$, thus they are reconcilable by 1. of Definition 14. Applying 3. of the same definition the reconciliabiliy of $X(X_1\{A\}, X_2\{B\})$ and $X(X_1\{\lambda\}, X_2\{\lambda\}, X_3\{C\}$ is obtained.

On the other hand, $X(X_1\{A\}, X_2\{B\})$ and $X(X_{\{1,3\}}\{\lambda\})$ are not reconcilable.

The following definition of coincidence ideals looks formally self-referential. However, it is not hard to see that a *rank* of a nested attribute can be defined via the recursive construction as follows. The rank of a simple attribute is 0. When a nested attribute is constructed using some constructor, like record, set, list, multiset or disjoint union, then the rank of the new attribute is one plus the maximum rank of the parts it is constructed from. In this setting, whenever a coincidence ideal or defect coincidence ideal is referred to in the definition of coincidence ideal, then it is of subattributes of a nested attribute of strictly lower rank, hence there is no circularity in the definition.

Definition 16. A *coincidence ideal* on $\mathcal{S}(X)$ is a subset $\mathcal{F} \subseteq \mathcal{S}(X)$ with the following properties:

1. $\lambda \in \mathcal{F}$;

2. if $Y \in \mathcal{F}$ and $Z \in \mathcal{S}(X)$ with $Y \geq Z$, then $Z \in \mathcal{F}$;

3. if $Y, Z \in \mathcal{F}$ are reconcilable, then $Y \sqcup Z \in \mathcal{F}$;

4. a) if $X_I\{\lambda\} \in \mathcal{F}$ and $X_J\{\lambda\} \notin \mathcal{F}$ for $I \subsetneq J$, then $X(X_{i_1}\{X_{i_1}'\}, \ldots, X_{i_k}\{X_{i_k}'\}) \in \mathcal{F}$ for $I = \{i_1, \ldots, i_k\}$;

   b) if $X_I\{\lambda\} \in \mathcal{F}$ and $X(X_i\{\lambda\}) \notin \mathcal{F}$ for all $i \in I$, then there is a partition $I = I_1 \cup I_2$ with $X_{I_1}\{\lambda\} \notin \mathcal{F}$, $X_{I_2}\{\lambda\} \notin \mathcal{F}$ and $X_{I'}\{\lambda\} \in \mathcal{F}$ for all $I' \subseteq I$ with $I' \cap I_1 \neq \emptyset \neq I' \cap I_2$;

   c) if $X_{\{1,\ldots,n\}}\{\lambda\} \in \mathcal{F}$ and $X_{I^-}\{\lambda\} \notin \mathcal{F}$ (for $I^- = \{i \in \{1, \ldots, n\} \mid X(X_i\{\lambda\}) \notin \mathcal{F}\}$), then there exists some $i \in I^+ = \{i \in \{1, \ldots, n\} \mid X(X_i\{\lambda\}) \in \mathcal{F}\}$ such that for all $J \subseteq I^-$ $X_{J \cup \{i\}}\{\lambda\} \in \mathcal{F}$ holds;

d) if $X_J\{\lambda\} \notin \mathcal{F}$ and $X_{\{j\}}\{\lambda\} \notin \mathcal{F}$ for all $j \in J$ and for all $i \in I$ there is some $J_i \subseteq J$ with $X_{J_i \cup \{i\}}\{\lambda\} \notin \mathcal{F}$, then $X_{I \cup J}\{\lambda\} \notin \mathcal{F}$, provided $I \cap J = \emptyset$;

e) if $X_{I^-}\{\lambda\} \in \mathcal{F}$ and $I' \subseteq I^+$ such that for all $i \in I'$ there is some $J \subseteq I^-$ with $X_{J \cup \{i\}}\{\lambda\} \notin \mathcal{F}$, then $X_{I' \cup J'}\{\lambda\} \notin \mathcal{F}$ for all $J' \subseteq I^-$ with $X_{J'}\{\lambda\} \notin \mathcal{F}$;

5. a) if $X_I\{\lambda\} \in \mathcal{F}$ and $X_J\{\lambda\} \in \mathcal{F}$ with $I \cap J = \emptyset$, then $X_{I \cup J}\{\lambda\} \in \mathcal{F}$;

b) if $X_I[\lambda] \in \mathcal{F}$ and $X_J[\lambda] \in \mathcal{F}$ with $I \cap J = \emptyset$, then $X_{I \cup J}[\lambda] \in \mathcal{F}$;

c) if $X_I\langle\lambda\rangle \in \mathcal{F}$ and $X_J\langle\lambda\rangle \in \mathcal{F}$ with $I \cap J = \emptyset$, then $X_{I \cup J}\langle\lambda\rangle \in \mathcal{F}$;

d) if $X_I[\lambda] \in \mathcal{F}$ and $X_J[\lambda] \in \mathcal{F}$ with $J \subseteq I$, then $X_{I-J}[\lambda] \in \mathcal{F}$;

e) if $X_I\langle\lambda\rangle \in \mathcal{F}$ and $X_J\langle\lambda\rangle \in \mathcal{F}$ with $J \subseteq I$, then $X_{I-J}\langle\lambda\rangle \in \mathcal{F}$;

f) if $X_I[\lambda] \in \mathcal{F}$ and $X_J[\lambda] \in \mathcal{F}$, then $X_{I \cap J}[\lambda] \in \mathcal{F}$ iff $X_{(I-J) \cup (J-I)}[\lambda] \in \mathcal{F}$;

g) if $X_I\langle\lambda\rangle \in \mathcal{F}$ and $X_J\langle\lambda\rangle \in \mathcal{F}$, then $X_{I \cap J}\langle\lambda\rangle \in \mathcal{F}$ iff $X_{(I-J) \cup (J-I)}\langle\lambda\rangle \in \mathcal{F}$;

6. a) for $X = X\{\bar{X}\{X_1(X_1') \oplus \cdots \oplus X_n(X_n')\}\}$, whenever $I \subseteq \{1, \ldots, n\}$, there is a partition $I = I^- \cup I_{+-} \cup I_+ \cup I_-$ such that

i. $X\{\bar{X}_{\{i\}}\{\lambda\}\} \in \mathcal{F}$ iff $i \notin I^-$,

ii. $X\{\bar{X}_{I'}\{\lambda\}\} \in \mathcal{F}$, whenever $I' \cap I_+ \neq \emptyset$,

iii. $X\{\bar{X}_{I'}\{\lambda\}\} \in \mathcal{F}$ iff $X\{\bar{X}_{I' \cap (I_{+-} \cup I^-)}\{\lambda\}\} \in \mathcal{F}$, whenever $I' \subseteq I_{+-} \cup I^- \cup I_-$;

b) for $X = X\langle\bar{X}\{X_1(X_1') \oplus \cdots \oplus X_n(X_n')\}\rangle$, whenever $I \subseteq \{1, \ldots, n\}$, there is a partition $I = I^- \cup I_{+-} \cup I_+ \cup I_-$ such that

i. $X\langle\bar{X}_{\{i\}}\{\lambda\}\rangle \in \mathcal{F}$ iff $i \notin I^-$,

ii. $X\langle\bar{X}_{I'}\{\lambda\}\rangle \in \mathcal{F}$, whenever $I' \cap I_+ \neq \emptyset$,

iii. $X\langle\bar{X}_{I'}\{\lambda\}\rangle \in \mathcal{F}$ iff $X\langle\bar{X}_{I' \cap (I_{+-} \cup I^-)}\{\lambda\}\rangle \in \mathcal{F}$, whenever $I' \subseteq I_{+-} \cup I^- \cup I_-$;

7. a) if $X = X(X_1', \ldots, X_n')$, then $\mathcal{F}_i = \{Y_i \in \mathcal{S}(X_i') \mid X(\lambda, \ldots, Y_i, \ldots, \lambda) \in \mathcal{F}\}$ is a coincidence ideal;

b) if $X = X[X']$, such that X' is not a union attribute, and $\mathcal{F} \neq \{\lambda\}$, then $\mathcal{G} = \{Y \in \mathcal{S}(X') \mid X[Y] \in \mathcal{F}\}$ is a coincidence ideal;

c) If $X = X_1(X_1') \oplus \cdots \oplus X_n(X_n')$ and $\mathcal{F} \neq \{\lambda\}$, then the set $\mathcal{F}_i = \{Y_i \in \mathcal{S}(X_i') \mid X_1(\lambda) \oplus \cdots \oplus X_i(Y_i) \oplus \cdots \oplus X_n(\lambda) \in \mathcal{F}\}$ is a coincidence ideal;

d) if $X = X\{X'\}$, such that X' is not a union attribute, and $\mathcal{F} \neq \{\lambda\}$, then $\mathcal{G} = \{Y \in \mathcal{S}(X') \mid X\{Y\} \in \mathcal{F}\}$ is a defect coincidence ideal;

e) if $X = X\langle X'\rangle$, such that X' is not a union attribute, and $\mathcal{F} \neq \langle\lambda\rangle$, then $\mathcal{G} = \{Y \in \mathcal{S}(X') \mid X\langle Y\rangle \in \mathcal{F}\}$ is a defect coincidence ideal.

A *defect coincidence ideal* on $\mathcal{S}(X)$ is a subset $\mathcal{F} \subseteq \mathcal{S}(X)$ satisfying properties 1, 2, 4(a)-(d), 6(a),(b), 7(d)-(e) and

8. a) if $X = X(X_1', \ldots, X_n')$, then $\mathcal{F}_i = \{Y_i \in \mathcal{S}(X_i') \mid X(\lambda, \ldots, Y_i, \ldots, \lambda) \in \mathcal{F}\}$ is a defect coincidence ideal;

   b) if $X = X[X']$, such that X' is not a union attribute, and $\mathcal{F} \neq \{\lambda\}$, then $\mathcal{G} = \{Y \in \mathcal{S}(X') \mid X[Y] \in \mathcal{F}\}$ is a defect coincidence ideal;

   c) If $X = X_1(X_1') \oplus \cdots \oplus X_n(X_n')$ and $\mathcal{F} \neq \{\lambda\}$, then the set $\mathcal{F}_i = \{Y_i \in \mathcal{S}(X_i') \mid X_1(\lambda) \oplus \cdots \oplus X_i(Y_i) \oplus \cdots \oplus X_n(\lambda) \in \mathcal{F}\}$ is a defect coincidence ideal.

The name "coincidence ideal" was chosen, because these ideals characterize sets of subattributes, on which two complex values coincide. This is formally shown in the following theorem. In [16, 20] the term "SHL-ideal" was used instead; in [17] in a restricted setting the term "HL-ideal" was used. Note that in all these cases not all the conditions from Definition 16 were yet present.

For the purposes of the present paper the following three statements from [22] are important and not the particular details of the definition above.

Theorem 17 (Theorem 3.1 in [22]). *Let $X \in \mathcal{N}$ be a nested attribute. For complex values $t_1, t_2 \in dom(X)$ let $\mathcal{F} = \{Y \in \mathcal{S}(X) \mid \pi_Y^X(t_1) = \pi_Y^X(t_2)\} \subseteq \mathcal{S}(X)$ be the set of subattributes, on which they coincide. Then $\mathcal{F}$ is a coincidence ideal.*

Theorem 18 (Theorem 3.2 in [22]). *Let $\mathcal{G} \subseteq \mathcal{S}(X)$ be a defect coincidence ideal for the nested attribute $X \in \mathcal{N}$ such that the union constructor appears in X only directly inside a set-, list or multiset-constructor. Then the following holds:*

1. *There exist two finite sets $S_1, S_2 \subseteq dom(X)$ such that $\{\pi_Y^X(\tau) \mid \tau \in S_1\} = \{\pi_Y^X(\tau) \mid \tau \in S_2\}$ holds iff $Y \in \mathcal{G}$. For $\mathcal{G} \neq \{\lambda\}$ both sets are non-empty.*

2. *There exist two finite multisets $M_1, M_2 \subseteq dom(X)$ such that $\langle \pi_Y^X(\tau) \mid \tau \in M_1 \rangle = \langle \pi_Y^X(\tau) \mid \tau \in M_2 \rangle$ holds iff $Y \in \mathcal{G}$. For $\mathcal{G} \neq \{\lambda\}$ both multisets are non-empty.*

Theorem 19 (Central Theorem,Theorem 3.3 in [22]). *Let $\mathcal{F} \subseteq \mathcal{S}(X)$ be a coincidence ideal for the nested attribute $X \in \mathcal{N}$ such that the union constructor appears in X only directly inside a set-, list or multiset-constructor. Then there exist two complex values $t_1, t_2 \in dom(X)$ such that $\pi_Y^X(t_1) = \pi_Y^X(t_2)$ holds iff $Y \in \mathcal{F}$.*

The long and technical proofs of the above theorems are included in [22].

3 Keys and Antikeys

In this section we assume that a set Σ of functional dependencies is given over $\mathcal{S}(X)$ and every statement is understood as "with respect to Σ". Since functional dependencies are defined between *sets* of subattributes, the following is a natural generalization of the concept of keys to the higher-order datamodel.

Definition 20. $\mathcal{K} \subseteq \mathcal{S}(X)$ is a *key* (with respect to Σ) if $\Sigma \models \mathcal{K} \to \mathcal{S}(X)$ holds. In other words, if r is an instance of $\mathcal{S}(X)$ satisfying Σ, then for any two distinct complex valued tuples $t_1, t_2 \in r$ there exists $K \in \mathcal{K}$ such that $\pi_K^X(t_1) \neq \pi_K^X(t_2)$ holds.

The following closure operation is important in the characterization of minimal key systems.

Definition 21. The *closure* of a set $\mathcal{Y} \subseteq \mathcal{S}(X)$ is defined as the intersection of all coincidence-ideals containing $\mathcal{Y}$:

$$cl(\mathcal{Y}) = \bigcap_{\substack{\mathcal{F} \text{ is a coincidence-ideal} \\ \mathcal{Y} \subseteq \mathcal{F}}} \mathcal{F}. \tag{1}$$

The idea behind Definition 21 is simple. We are interested in the following: assume that two tuples agree on a set of subattributes, where do they need to agree besides those? Since it was proved in [22] that the set of subattributes where two tuples coincide form a coincidence ideal, if $\pi_Y^X(t_1) = \pi_Y^X(t_2)$ for all $Y \in \mathcal{Y}$, then $\mathcal{Y}$ is a subset of the set of subattributes where t_1, t_2 coincide, which is a coincidence ideal. Because $cl(\mathcal{Y})$ is a subset of that ideal, t_1, t_2 coincide on all $Z \in cl(\mathcal{Y})$.

Proposition 22. *The operator cl is a closure operator, that is*

1. $\mathcal{Y} \subseteq cl(\mathcal{Y})$;
2. *If* $\mathcal{Y} \subseteq \mathcal{Z}$, *then* $cl(\mathcal{Y}) \subseteq cl(\mathcal{Z})$;
3. $cl(cl(\mathcal{Y})) = cl(\mathcal{Y})$.

Clearly, if $\mathcal{K}$ is a key and $\mathcal{K} \subset \mathcal{H}$, then $\mathcal{H}$ is a key, as well. In particular, the closure of a key is also a key. The interesting fact is that the converse also holds.

Theorem 23. $\mathcal{K} \subseteq \mathcal{S}(X)$ *is a key iff* $cl(\mathcal{K})$ *is a key.*

Proof. According to the previous note, only the implication "$cl(\mathcal{K})$ is a key $\Longrightarrow$ $\mathcal{K}$ is a key" needs to be proven. Suppose that $\mathcal{K}$ is not a key, that is $\Sigma \not\models \mathcal{K} \to \mathcal{S}(X)$. Thus, there exists an instance r of $\mathcal{S}(X)$ satisfying Σ and two complex-valued tuples $t_1 \neq t_2$ in r such that $\forall K \in \mathcal{K}\colon \pi_K^X(t_1) = \pi_K^X(t_2)$ holds. Let $\mathcal{F} = \{Z \mid \pi_Z^X(t_1) = \pi_Z^X(t_2)\}$. Since $\mathcal{F}$ is a coincidence-ideal that contains $\mathcal{K}$, $cl(\mathcal{K}) \subseteq \mathcal{F}$ holds. This implies, that $cl(\mathcal{K})$ is not a key either.

□

Antikeys are defined in the relational model as any subset of attributes that are not keys. Here, the same works.

Definition 24. A subset $\mathcal{A} \subset \mathcal{S}(X)$ is an *antikey* (with respect to Σ), if $\Sigma \not\models \mathcal{A} \to \mathcal{S}(X)$. In other words, there exists an instance r of $\mathcal{S}(X)$ satisfying Σ and two complex-valued tuples $t_1 \neq t_2$ in r such that $\forall A \in \mathcal{A}\colon \pi_A^X(t_1) = \pi_A^X(t_2)$ holds.

It is clear, that if $\mathcal{A}$ is an antikey, and $\mathcal{B} \subseteq \mathcal{A}$, then $\mathcal{B}$ is an antikey, as well. In particular, if $cl(\mathcal{A})$ is an antikey, then so is $\mathcal{A}$, as well. Again, the interesting fact is that the converse is also true follows from Theorem 23.

Corollary 25. *$\mathcal{A} \subset \mathcal{S}(X)$ is an antikey, iff $cl(\mathcal{A})$ is an antikey.*

Theorem 23 and Corollary 25 allow considering only closed sets as keys or antikeys. $\mathcal{H}$ is *closed* if $\mathcal{H} = cl(\mathcal{H})$. Indeed, if we have a key $\mathcal{K}$, then its closure $cl(\mathcal{K})$ is also a key and every $\mathcal{K}'$ with $cl(\mathcal{K}') = cl(\mathcal{K})$ is a key. as well. This means that the system of closed sets that are keys uniquely determines the system of all keys. Thus, we concentrate on only closed sets in the following.

We are interested in *minimal keys* and *maximal antikeys*, where minimal and maximal is with respect (set-wise) containment. Given Σ, the system $\mathfrak{K} = \{\mathcal{K}_1, \ldots, \mathcal{K}_k\}$ of all minimal keys forms a *Sperner system* or *antichain* of sets of subattributes, that is for every pair of indices i and j $\mathcal{K}_i \subsetneq \mathcal{K}_j$ holds. Analogously, maximal antikeys form a Sperner system $\mathfrak{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_a\}$ of sets of subattributes.

Proposition 26. *The system of minimal keys and the system of maximal antikeys mutually determine each other.*

Proof. Consider the poset $\mathfrak{P}$ of closed subsets of $\mathcal{S}(X)$ ordered by (set-wise) inclusion. Keys form an *up-set*, or *filter*, that is a subset $\mathbb{K} \subseteq \mathfrak{P}$ such that if $\mathcal{K} \in \mathbb{K}$ and $\mathcal{K} \subset \mathcal{H}$, then $\mathcal{H} \in \mathbb{K}$. Similarly, antikeys form a *down-set*, or *ideal*, that is a subset $\mathbb{A} \subseteq \mathfrak{P}$ such that if $\mathcal{A} \in \mathbb{A}$ and $\mathcal{B} \subset \mathcal{A}$, then $\mathcal{B} \in \mathbb{A}$. Most importantly, $\mathbb{K} \cup \mathbb{A} = \mathfrak{P}$ and $\mathbb{K} \cap \mathbb{A} = \emptyset$. The system of minimal keys is the set of minimal elements of $\mathbb{K}$, while the system of maximal antikeys is the set of maximal elements of $\mathbb{A}$.

□

Proposition 26 allows the following notation. If $\mathfrak{K}$ is the system of minimal keys, then the corresponding system of maximal antikeys $\mathfrak{A}$ is denoted by $\mathfrak{K}^{-1}$, as well. Observe, that this notation can be extended to any Sperner-system $\mathfrak{S}$ of closed sets by $\mathfrak{S}^{-1}$ being the collection of the maximal elements of the ideal that is the complement in $\mathfrak{P}$ of the filter $\mathbb{S}$ generated by $\mathfrak{S}$.

4 Armstrong Instances

The principal interest of the present paper is to investigate which Sperner systems of closed subsets of $\mathcal{S}(X)$ can occur as systems of minimal keys for some suitable family of functional dependencies Σ. The idea of Armstrong instance is that given a family of constraints (e.g. functional dependencies) and a subset Σ of that family, one looks for a model (database) that satisfies only those constraints in Σ and no others. The practical use of this concept is that during conceptual schema design, the designer is able to check whether some constraints are logical consequences of the constraints of the design by obtaining an Armstrong instance and checking what dependencies are satisfied besides the ones designed. For the relational model, there are even software packages constructing Armstrong instances.

In the relational model Armstrong [4] and Demetrovics [6] proved that *every* Sperner system arises as set of minimal keys, i.e., has an Armstrong instance. Later Fagin [10] gave necessary and sufficient conditions for constraints that can be described by Horn clauses, to have Armstrong instance in the framework of the relational datamodel. However, in [21] it was shown that in the higher-order datamodel, although in the restricted "counter-free" case, the same statement does not hold.

Definition 27. Let r be an instance of a nested attribute X, with subattribute lattice $\mathcal{S}(X)$. A subset $\mathcal{K} \subseteq \mathcal{S}(X)$ is *key with respect to* r, if $r \models \mathcal{K} \rightarrow \mathcal{S}(X)$, i.e., there exist no two distinct complex-valued tuples $t_1, t_2 \in r$ such that $\forall K \in \mathcal{K}\colon \pi^X_K(t_1) = \pi^X_K(t_2)$ holds. r is an *Armstrong-instance* for a Sperner system $\mathfrak{K}$ of closed subsets of $\mathcal{S}(X)$, if the system of minimal keys with respect to r is exactly $\mathfrak{K}$.

A simple characterization can be given for Armstrong instances.

Proposition 28. *Let $\mathfrak{K}$ be a Sperner system of closed subsets of $\mathcal{S}(X)$. An instance r is Armstrong-instance for minimal key system $\mathfrak{K}$ iff*

*[**Key**] For all $\mathcal{K} \in \mathfrak{K}$ and any two complex-valued tuples $t_1, t_2 \in r$ there exists $K \in \mathcal{K}$ such that $\pi^X_K(t_1) \neq \pi^X_K(t_2)$ holds.*

*[**Antikey**] For all $\mathcal{A} \in \mathfrak{K}^{-1}$ there exists two complex-valued tuples $t_1 \neq t_2$ in r such that $\forall A \in \mathcal{A}\colon \pi^X_A(t_1) = \pi^X_A(t_2)$ holds.*

□

The [**Antikey**] property of Proposition 28 gives an immediate necessary condition for existence of an Armstrong-instance. Indeed, every maximal antikey must be a coincidence ideal. This is a real restriction, since not all closed sets are coincidence ideals. For example, consider $\mathcal{S}(X\{X_1(A) \oplus X_2(B) \oplus X_3(C)\})$ of Figure 2. For the sake of convenience the principal ideal $\{Y \in \mathcal{S}(X) \mid Y \leq Z\}$ of $\mathcal{S}(X)$ generated by $Z \in \mathcal{S}(X)$ is denoted by $Z{\downarrow}$. The principal ideal $\mathcal{J} = X(X_1\{\lambda\}, X_2\{\lambda\}){\downarrow}$ is closed see Proposition 37, but not a coincidence ideal, since it violates property 4(a) of Definition 16. Taking $\mathbb{K} \subset \mathfrak{P}$ be the set of closed subsets of $\mathcal{S}(X)$ that do not contain $\mathcal{J}$, we obtain that the unique maximal antikey corresponding to the key system $\mathbb{K}$ is $\mathcal{J}$. However, since $\mathcal{J}$ is not a coincidence ideal, $\mathbb{K}$ cannot have an Armstrong-instance.

On the other hand, minimal keys can indeed be closed sets that are not coincidence ideals. Consider again $\mathcal{S}(X\{X_1(A) \oplus X_2(B) \oplus X_3(C)\})$ of Figure 2. Let $\mathcal{A} = X(X_1\{A\}){\downarrow}$. It is not hard to see that $\mathcal{A}$ is a coincidence ideal. Indeed, properties 1-3 of Definition 16 are trivially satisfied by any principal ideal. Property 4(a) is satisfied, because the only possible choice of I that satisfies the conditions is $I = \{1\}$. Conditions in points (b), (c), and (e) of property 4 do not apply to $\mathcal{A}$, while 4(d) is satisfied trivially. Finally, the conditions in properties 5-7 do not apply to $\mathcal{A}$, hence by the Central Theorem (Theorem 19) there exists two tuples

$t_1, t_2 \in dom(X)$ with $\pi_A^X(t_1) = \pi_A^X(t_2)$ iff $A \in \mathcal{A}$. In fact the proof of the Central Lemma constructs the tuples $t_1 = \emptyset$ and $t_2 = \{(X_1 : a_1)\}$. $\mathcal{K} = X(X_1\{\lambda\}, X_2\{\lambda\})\downarrow$ is a minimal key with respect to the instance $r = \{t_1, t_2\}$.

The trouble with Armstrong-instances are caused by *degenerate keys.*

Definition 29. A key $\mathcal{K}$ is called *degenerate*, if every $K \in \mathcal{K}$ is constructed using only λ, set-constructor, record-constructor and union-constructor. That is, K does not contain simple attributes, multiset- or list-constructors.

Similar question was considered by Fagin and Vardi for the relational model in [11], where functional dependencies with non-empty left hand side were called *standard*, and the problems of working with non-standard functional dependencies were investigated. The following theorem gives a sufficient condition for the existence of Armstrong-instance.

Theorem 30. *Let $\mathfrak{K} = \{\mathcal{K}_i \mid i = 1, 2, \ldots, k\}$ be a Sperner system of closed subsets of $\mathcal{S}(X)$. There exists an Armstrong-instance r for $\mathfrak{K}$ as system of minimal keys provided the following two conditions hold:*

1. *$\mathfrak{K}$ does not contain degenerate keys;*
2. *Each element of $\mathfrak{A} = \mathfrak{K}^{-1}$ is a coincidence ideal.*

Proof. Let $\mathfrak{K}^{-1} = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$. The restructuring rules allow us to assume that the union-constructor only appears inside a set-constructor or as the outermost constructor, hence Theorem 19 provides complex values t_0^i, t_1^i $i = 1, 2, \ldots, k$ such that

$$\pi_Y^X(t_0^i) = \pi_Y^X(t_1^i) \iff Y \in \mathcal{A}_i. \tag{2}$$

This ensures that $\mathcal{A}_i$ is an antikey for all i. On the other hand, we have to show that each $\mathcal{K}_i \in \mathfrak{K}$ is a key. In order to do so, the complex valued tuples will be modified preserving (2) so that if $\pi_Z^X(t_a^i) = \pi_Z^X(t_b^j)$ for some $Z \in \mathcal{S}(X)$, $a, b \in \{0, 1\}$, and $1 \leq i < j \leq k$, then Z cannot contain simple attributes or list or multiset subattributes. Hence no two complex values can agree on every subattribute in $\mathcal{K}_s$ for all s, which implies that each $\mathcal{K}_s$ is a key. Note, that the number of complex values in this Armstrong instance is exactly $2|\mathfrak{K}^{-1}|$.

The modification of the tuples is as follows. For simple attributes we have to take care of that during the inductive construction of the tuples the constants from the domains of simple attributes used for $\mathcal{A}_i$ must be distinct from those used for $\mathcal{A}_j$ if $i \neq j$. This ensures that values constructed for $\mathcal{A}_i$ and those constructed for $\mathcal{A}_j$ for $i \neq j$ cannot agree on subattributes containing a simple attribute.

For the list attribute case if one of $\mathcal{A}_i$'s is $\{\lambda\}$, then we have only one coincidence ideal by the Sperner property, so there is nothing to prove. Otherwise, consider the inductive construction of the tuples t_0^i, t_1^i for $\mathcal{A}_i$. We modify that in a sequential order for $i = 2, 3, \ldots, k$. When we encounter a list subattribute $X[X']$, (X' could be a union) the proof of Theorem 19 constructs two tuples $t_0^{i'}, t_1^{i'}$ that are of the form $t_a^{i'} = [t_a^{i''}]$, $a = 0, 1$. Let m be the largest multiplicity of any element in any

list in t_0^j and t_1^j, $j = 1, 2, \ldots, i-1$. Now, we replace $t_0^{i\prime}, t_1^{i\prime}$ with $t_a^{i*} = [(m+1) \cdot t_a^{i\prime\prime}]$, i.e. t_a^{i*} is a list with $m+1$ occurrences of the same element $t_a^{i\prime\prime}$. This modification ensures that multiplicities inside lists cannot agree in tuples constructed for distinct $\mathcal{A}_i$'s while preserving the property (2). The multiset attribute case is similar.

□

4.1 The Case $\mathcal{S}(X\{X_1(A_1) \oplus X_2(A_2) \oplus \cdots \oplus X_n(A_n)\})$

In the present section we study a special case, which is archetypical. This nested attribute exhibits most of the problems with respect to Armstrong-instances, thus showing the complexity of the problem. We believe that effective treatment of this case would lead to general insight of the nature of Armstrong-instances of nested attributes. As a beginning in that direction, a characterization is given for the existence of such instances. Let r be an instance of $\mathcal{S}(X\{X_1(A_1) \oplus X_2(A_2) \oplus \cdots \oplus X_n(A_n)\})$. According to Definition 5, complex value $t \in r$ can be considered as a tuple $t = (X_1 : a_1, \ldots, X_n : a_n)$, where a_i is a finite subset of the domain of A_i for $i = 1, 2, \ldots, n$. The *pattern of* t is an n-tuple p_t of $+$'s and $-$'s, such that the i^{th} coordinate of p_t is $+$, if $a_i \neq \emptyset$, and $-$, if $a_i = \emptyset$.

Proposition 31. *Let r be an instance of $X\{X_1(A_1) \oplus X_2(A_2) \oplus \cdots \oplus X_n(A_n)\}$, and let $\mathfrak{K} = \{\mathcal{K}_1, \ldots, \mathcal{K}_k\}$ be the system of minimal keys with respect to r. If there exists an i such that $\mathcal{K}_i$ is degenerate, then r contains at most one complex valued tuple of each possible pattern. Consequently, $|r| \leq 2^n$.*

Proof. Attributes in a degenerate key can only have the form $X_I\{\lambda\}$ for some $I \subseteq \{1, 2, \ldots, n\}$ or $X(X_{i_1}\{\lambda\}, X_{i_2}\{\lambda\}, \ldots X_{i_s}\{\lambda\})$. The projection of a complex-valued tuple t to such an attribute is determined by which coordinates of t are non-empty, hence depend only on the patter p_t.

□

Any two subattributes that are not of type $X_I\{\lambda\}$ for some $|I| > 1$ are reconcilable since the possible i^{th} components of a tuple are $\lambda, X_i\{\lambda\}$ and $X_i\{A_i\}$ that are pairwise comparable, that is reconcilable. Thus, a coincidence ideal $\mathcal{A}$ contains the $\sqcup$ of any pair of non-counter attributes belonging to $\mathcal{A}$. It follows then that $\mathcal{A}$ consists of a principal ideal of non-counter attributes extended with some counter attributes of type $X_I\{\lambda\}$. Recall, that a *principal ideal* generated by an element κ in a lattice consists of all elements μ of that lattice with $\mu \leq \kappa$.

Take a Sperner system of closed sets $\mathfrak{K} = \{\mathcal{K}_1, \ldots, \mathcal{K}_k\}$ and the Sperner system $\mathfrak{K}^{-1} = \mathfrak{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ as candidates for minimal keys and maximal antikeys, respectively. Assuming that each $\mathcal{A}_i$ is a coincidence ideal, a pair of tuple patterns is obtained via Theorem 19 for each $\mathcal{A}_i$, together with a constraint $\varphi_{i_1} \wedge \ldots \wedge \varphi_{i_b}$, where φ_{i_j} requires $=$ or $\neq$ on common $+$-component i_j. Indeed, let $X(Y_1, Y_2, \ldots, Y_n)$ be the largest (generator) element of the principal ideal part of $\mathcal{A}_i$. If both tuple patterns contain $+$ in component i_j, then either $Y_{i_j} = X_{i_j}\{\lambda\}$ or $Y_{i_j} = X_{i_j}\{A_{i_j}\}$. In the first case the tuples that agree on exactly $\mathcal{A}_i$ must contain different nonempty

sets in the i_jth component giving φ_{i_j} being $\neq$. Note that the tuples cannot agree on $X_{i_j}\{A_{i_j}\}$ in this case. On the other hand, if $Y_{i_j} = X_{i_j}\{A_{i_j}\}$, then the i_jth component of the tuples must contain the same non-empty set, thus φ_{i_j} is $=$.

Proposition 32. *The pair of patterns and the constraints are uniquely determined by the counter attributes contained in $\mathcal{A} \in \mathfrak{A}$, provided the pair consists of distinct patterns.*

Proof. Assume that $\mathcal{A} = X(Y_1, Y_2, \ldots, Y_n)\downarrow\cup\{X_I \mid I \in \mathcal{I}\}$ where Y_i is either λ, $X_i\{\lambda\}$ or $X_i\{A_i\}$. Furthermore, assume that two complex valued tuples $t_1 = (X_1 : a_1, \ldots, X_n : a_n)$ and $t_2 = (X_1 : b_1, \ldots, X_n : b_n)$ agree exactly on subattributes of $\mathcal{A}$. It means that if $Y_i = \lambda$, then one of a_i and b_i is empty and the other is nonempty, if $Y_i = X_i\{\lambda\}$, then $a_i \neq b_i$ and both are nonempty, while if $Y_i = X_i\{A_i\}$, then either $a_i = b_i$ and both are nonempty, or both are empty. Thus, patterns of t_1 and t_2 have the same symbol in coordinate j where $Y_j = X_i\{A_j\}$ or $Y_j = X_j\{\lambda\}$, and opposite symbols for $Y_j = \lambda$. $X_I\{\lambda\} \in \mathcal{A}$ for $|I| > 1$ means that both t_1 and t_2 have a nonempty coordinate whose index is in I, where the nonempty coordinates of t_1 and t_2 showing $X_I\{\lambda\} \in \mathcal{A}$ need not have the same index. Let us assume that $Y_{i_0} = \lambda$ and t_1 has empty $i_0{}^{\text{th}}$ coordinate. If $X_{\{i_0,j\}}\{\lambda\} \in \mathcal{A}$, then the j^{th} coordinate of t_1 is nonempty. On the other hand, if $X_{\{i_0,j\}}\{\lambda\} \notin \mathcal{A}$, then j^{th} coordinate of t_1 is empty. In both cases the j^{th} coordinate of pattern of t_2 is uniquely determined, as well.

The remaining case is when none of Y_i's is λ. In this case, using the same argument as before, the patterns of t_1 and t_2 are the same.

□

For example, if $\mathcal{A}_i = X(X_1\{A_1\}, X_2\{\lambda\})\downarrow\cup\{X_I \mid I \cap \{1,2\} \neq \emptyset\}$, then the values $\{(X_1 : v_1), (X_2 : v_2)\}$ and $\{(X_1 : v_1), (X_2 : v_2'), (X_3 : v_3), \ldots, (X_n : v_n)\}$ coincide exactly on $\mathcal{A}_i$.

The corresponding pair of tuple patterns is $\{(+,+,-\ldots,-), (+,+,\ldots,+)\}$, and the constraints are φ_1: $=_1$, φ_2: $\neq_2$.

Construct a graph on 2^n vertices with vertex set V being all possible patterns. Add a *green* edge between the two patterns given by $\mathcal{A}_i$ labeled with the appropriate constraint, for each candidate antikey $\mathcal{A}_i$. For each pair of patterns $\mathfrak{K}$ defines (a more complicated) constraint on the patterns. That is, each $\mathcal{K}_j$ need to have an element K where the two tuples corresponding to the pair of patterns have distinct projections. For each K, a disjunction of conjuncts can be formulated, and the disjunction Φ_j of these formulae expresses that $\mathcal{K}_j$ is a key. Finally, the constraint on the pair of patterns defined by $\mathfrak{K}$ is $\Phi_1 \wedge \ldots \wedge \Phi_k$. Add *red* edge between two patterns labeled by the appropriate constraint.

Theorem 33. *Let $\mathfrak{K} = \{\mathcal{K}_1, \ldots, \mathcal{K}_k\}$ be a Sperner system of closed sets that contains a degenerate key and assume that $\mathfrak{K}^{-1} = \mathfrak{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ consists of coincidence ideals. Let G be the graph on the patterns with green and red labeled edges as constructed above. $\mathfrak{K}$ has an Armstrong-instance iff the subgraph G' spanned by the* green *edges has edge labels (both green and red), that can be simultaneously satisfied by a set r of tuples that contain tuples of each pattern of G'.*

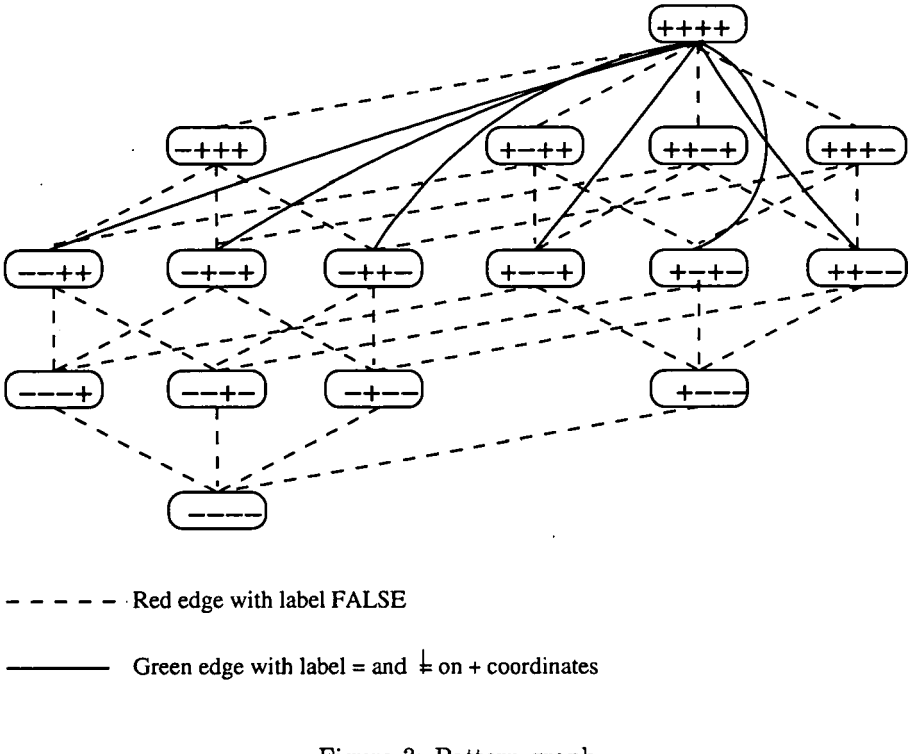


Figure 3: Pattern graph

Proof. If $\mathfrak{K} = \{\mathcal{K}_1, \ldots, \mathcal{K}_k\}$ has an Armstrong-instance r, then the patterns of tuples in r satisfy each edge constraint. On the other hand, assume that the edge labels of the subgraph G' spanned by the green edges are simultaneously satisfiable, let r be a set of tuples that satisfy all constraints in G'. Since there is a degenerate candidate key in $\mathfrak{K}$, $X(X_1\{\lambda\}, X_2\{\lambda\}, \ldots, X_n\{\lambda\})$ cannot be contained in any of the candidate antikeys, hence by Proposition 32 there is a unique green edge with label for each $\mathcal{A}_i$. The pair of tuples corresponding to the endpoints of the edge agree exactly on subattributes of $\mathcal{A}_i$, showing that it is an antikey. The red edge labels make sure that each $\mathcal{K}_i$ is a key. Since $\mathfrak{K}^{-1} = \mathfrak{A}$, we have that $\mathfrak{K}$ is the system of minimal keys, $\mathfrak{A}$ is the system of maximal antikeys with respect to r. □

Example 34. Let $n = 4$. For all four choices of $1 \leq i < j < k \leq 4$ let $\mathfrak{K}$ consist of the principal ideals $X(X_i\{\lambda\}, X_j\{\lambda\}, X_k\{\lambda\})\downarrow$. Then $\mathfrak{K}^{-1} = \mathfrak{A}$ consist of $\mathcal{A}_{i,j} = X(X_i\{A_i\}, X_j\{\lambda\})\downarrow \cup \{X_I \mid I \cap \{i,j\} \neq \emptyset\}$ for all six choices of $1 \leq i < j \leq 4$. The pattern graph for $\mathfrak{K}$ and $\mathfrak{A}$ is shown on Figure 3.

For instance, between $++++$ and $--++$ we have a green edge with label $\varphi_3 \wedge \varphi_4$ given by $\varphi_3 :=_3$ and $\varphi_4 :\neq_4$. This edge originates from $\mathcal{A}_{3,4}$. Between $-+++$ and $--++$ we get a red edge with label FALSE, because the key $\mathcal{K} =$

$X(X_1\{\lambda\}, X_2\{\lambda\}, X_3\{\lambda\})\downarrow$ with $K = X(X_1\{\lambda\}, X_2\{\lambda\}, X_3\{\lambda\})$ will always yield inequality for the second component. Similarly, between $---+$ and $--+-$ we get a red edge with label TRUE, because each key $\mathcal{K}_{ijk} = X(X_i\{\lambda\}, X_j\{\lambda\}, X_k\{\lambda\})\downarrow$ contains $K = X(X_3\{\lambda\})$ or $K = X(X_4\{\lambda\})$, so we know the required inequality will be satisfied.

The red (dotted) edges with constraint label TRUE are not drawn. They are between pairs of vertices that have at least two coordinates where one of them is + and the other is -, that is exactly the complement of the drawn dotted graph. It is easy to see that the labels on the subgraph induced by the green (continuous line) edges are satisfiable, $r = \{(X_1 : a, X_2 : b, X_3 : c, X_4 : d), (X_1 : a, X_2 : b'), (X_1 : a, X_3 : c'), (X_1 : a, X_4 : d'), (X_2 : b, X_3 : c'), (X_2 : b, X_4 : d'), (X_3 : c, X_4 : d')\}$ is an Armstrong-instance.

Note that the Armstrong instance constructed in Example 34 contains a value for each of the edges in the subgraph of the pattern graph spanned by the green edges. This is the construction used in the proof of Theorem 33.

4.1.1 Some Negative Results.

In [21] some examples were shown that did not have Armstrong-instance in the counter-free case. The proofs there were sometimes quite involved, which was caused by not considering the counter attributes. If those are taken into account, the proofs can be shortened, since the counter attributes contained in the maximal antikeys sort of determine the patterns of possible complex values in an Armstrong-instance.

Example 35. This example is from [21], but the proof is much shorter. Let $X = X\{X_1(A_1) \oplus X_2(A_2) \oplus X_3(A_3) \oplus X_4(A_4)\}$ and consider $\mathcal{S}(X)$. Let the Sperner system $\mathfrak{S}$ of closed sets consist of the principal ideals generated by

$$X(X_i\{A_i\}, X_j\{A_j\}) \text{ for } 1 \le i < j \le 4,$$
$$X(X_i\{A_i\}, X_j\{\lambda\}, X_k\{\lambda\}),$$
$$X(X_i\{\lambda\}, X_j\{A_j\}, X_k\{\lambda\}),$$
$$X(X_i\{\lambda\}, X_j\{\lambda\}, X_k\{A_k\}) \text{ for } 1 \le i < j < k \le 4 \text{ and}$$
$$X(X_1\{\lambda\}, X_2\{\lambda\}, X_3\{\lambda\}, X_4\{\lambda\}).$$

The system of maximal antikeys is the set of coincidence-ideals

$$X(X_i\{A_i\}, X_j\{\lambda\})\downarrow \cup \{X_I \mid |I| > 1\} \text{ for } i \ne j,\ i, j \in \{1, 2, 3, 4\} \text{ and}$$
$$X(X_i\{\lambda\}, X_j\{\lambda\}, X_k\{\lambda\})\downarrow \cup \{X_I \mid |I| > 1\} \text{ for } 1 \le i < j < k \le 4.$$

The patterns belonging to $X(X_1\{A_1\}, X_2\{\lambda\})\downarrow \cup \{X_I \mid |I| > 1\}$ are $(+, +, +, -)$ and $(+, +, -, +)$, the edge constraints are φ_1: $=_1$, φ_2: $\ne_2$. The same pair of patterns belong to $X(X_2\{A_2\}, X_1\{\lambda\})\downarrow \cup \{X_I \mid |I| > 1\}$, however the edge constraints are φ_1: $\ne_1$, φ_2: $=_2$. These two sets of constraints are clearly contradictory, hence by Theorem 33 there exists no Armstrong-instance for $\mathfrak{S}$.

The next example shows that there is a significant difference between the counter-free case and the general case.

Example 36. Consider again $X = X\{X_1(A_1) \oplus X_2(A_2) \oplus X_3(A_3) \oplus X_4(A_4)\}$ and $\mathcal{S}(X)$. Let $\mathfrak{K}$ be the Sperner system of the following closed sets of subattributes. $\mathfrak{K} = \{X(X_1\{\lambda\}, X_2\{\lambda\})\downarrow, X(X_1\{\lambda\}, X_3\{\lambda\})\downarrow, X(X_2\{\lambda\}, X_4\{\lambda\})\downarrow, X(X_3\{\lambda\}, X_4\{\lambda\})\downarrow\}$.

$\mathfrak{K}^{-1}$ consists of $(X(X_2\{A_2\}, X_3\{A_3\})\downarrow$ and $X(X_1\{A_1\}, X_4\{A_4\})\downarrow$ in the counter-free case. It is easy to see that the following three tuples $(X_1 : a, X_2 : b, X_3 : c, x_4 : d), (X_2 : b, X_3 : c), (X_1 : a, x_4 : d)$ form an Armstrong-instance. However, in the general case the maximal antikeys are

$$\mathcal{A}_1 = X(X_2\{A_2\}, X_3\{A_3\})\downarrow\cup\{X_I \mid |I| > 1\}$$

and

$$\mathcal{A}_2 = X(X_1\{A_1\}, X_4\{A_4\})\downarrow\cup\{X_I \mid |I| > 1\}.$$

The pair of patterns determined by $\mathcal{A}_2$ is $(+,-,+,+)$ and $(+,+,-,+)$, while $\mathcal{A}_1$ gives $(-,+,+,+)$ and $(+,+,+,-)$. However, tuples of patterns $(+,+,-,+)$ and $(+,+,+,-)$, respectively, agree on the key $X(X_1\{\lambda\}, X_2\{\lambda\})\downarrow$. Thus, $\mathfrak{K}$ does not have an Armstrong-instance in the case of counter attributes being considered.

4.2 Structural induction?

Most of the proofs about higher-order datamodels exploit structural induction. Some of the constructors allow lifting an Armstrong-instance. Consider the list constructor, for example. Let $X[X']$ be a nested attribute, and let $\mathfrak{K} = \{\mathcal{K}_1, \ldots, \mathcal{K}_m\}$ be a candidate key system in $\mathcal{S}(X')$ that has an Armstrong-instance $r = \{t_1, \ldots, t_s\}$. Then it is easy to see that $\overline{r} = \{[t_1], \ldots, [t_s]\}$ is an Armstrong instance for the candidate key system $\overline{\mathfrak{K}} = \{[\mathcal{K}_1], \ldots, [\mathcal{K}_m]\}$ of $\mathcal{S}(X)$. We use the notation $[\mathcal{K}_i] = \{[K] \mid K \in \mathcal{K}_i\}$.

However, the reverse is obviously not true. Consider $X = X[X'\{X_1(A_1) \oplus X_2(A_2) \oplus X_3(A_3) \oplus X_4(A_4)\}]$ and the candidate key system $\mathfrak{S}$ consisting of $X[X'(X_i\{A_i\}, X_j\{A_j\})]\downarrow$, $1 \le i < j \le 4$, $X[X'(X_i\{A_i\}, X_j\{\lambda\}, X_k\{\lambda\})]\downarrow$, $X[X'(X_i\{\lambda\}, X_j\{A_j\}, X_k\{\lambda\})]\downarrow$, $X[X'(X_i\{\lambda\}, X_j\{\lambda\}, X_k\{A_k\})]\downarrow$ $1 \le i < j < k \le 4$, and $X[X'(X_1\{\lambda\}, X_2\{\lambda\}, X_3\{\lambda\}, X_4\{\lambda\})]\downarrow$. This system consists of non-degenerate keys, thus by Theorem 30 it has an Armstrong-instance. Indeed, $\mathcal{A}$ is a maximal antikey for the candidate key system in Example 35 iff $[\mathcal{A}]$ is a maximal antikey for $\mathfrak{S}$. Since $\mathcal{A}$ is a coincidence ideal, according to property 5(b) of Definition 16 $[\mathcal{A}]$ is a coincidence ideal as well, thus both conditions of Theorem 30 are satisfied. If $\mathfrak{K}$ denotes the candidate key system in Example 35, then $\mathfrak{S} = [\mathfrak{K}]$. $\mathfrak{S}$ has Armstrong-instance, but $\mathfrak{K}$ does not.

This example shows that there is no hope for deciding about Armstrong-instance using structural induction. Another example of the same flavor can be given using the record constructor. Consider $X = X(X'\{X_1(A_1) \oplus X_2(A_2) \oplus X_3(A_3) \oplus X_4(A_4)\}, Y[B])$ and the subattribute lattice $\mathcal{S}(X)$. As before, let $\mathfrak{K}$ denote the candidate key system in Example 35, and let $\mathcal{G}_{\mathcal{K}} = \{X(K, Y[B]) \mid K \in \mathcal{K}\}$. The

Sperner system $\mathfrak{Q} = \{\mathcal{G}_\mathcal{K} \mid \mathcal{K} \in \mathfrak{K}\}$ consists of non-degenerate candidate keys. $\mathfrak{Q}^{-1}$ consists of $X(X'\{X_1(A_1) \oplus X_2(A_2) \oplus X_3(A_3) \oplus X_4(A_4)\}, Y[\lambda])\downarrow$ and $X(\mathcal{A}, Y[B])\downarrow$, where $\mathcal{A} \in \mathfrak{K}^{-1}$. These are coincidence ideals, thus by Theorem 30 $\mathfrak{Q}$ has an Armstrong-instance. However, the projection of $\mathfrak{Q}$ to the first component is exactly the system in Example 35.

5 Strong Keys

Keys correspond to ideals of the subattribute lattice with some additional properties. Principal ideals form an important subclass of ideals. Another reason for considering principal ideals is that in the relational datamodel each candidate key that is a closed set is a principal ideal.

Proposition 37. *Let $\mathcal{Y} = Y\downarrow$ be a principal ideal of the subattribute lattice $\mathcal{S}(X)$ of a nested attribute X. Then $\mathcal{Y}$ is closed.*

Proof. One has to show that

$$\mathcal{Y} = \bigcap_{\substack{\mathcal{F} \text{ is a coincidence-ideal} \\ \mathcal{Y} \subseteq \mathcal{F}}} \mathcal{F}, \tag{3}$$

or in other words, if $\mathcal{Y} \subset \mathcal{F}$ for a coincidence ideal $\mathcal{F}$ and $Z \in \mathcal{F} \setminus \mathcal{Y}$, then there exists a coincidence ideal $\mathcal{G}$ with $\mathcal{Y} \subset \mathcal{G}$ and $Z \notin \mathcal{G}$. If $\mathcal{Y}$ is not a coincidence ideal itself, then it violates some of the properties of Definition 16. However, a principal ideal can only violate 2(a)-(e). These always give choice that either one or another subattribute must be in a coincidence ideal. Thus to construct $\mathcal{G}$ one only has to avoid adding Z, when there is a choice. Since $\mathcal{S}(X)$ is finite, after finitely many extensions the coincidence ideal $\mathcal{G}$ is obtained.

□

Proposition 37 states that principal ideals are candidate keys. Thus, the next definition is meaningful.

Definition 38. A Sperner system of closed sets of $\mathcal{S}(X)$ is called a *strong* candidate key system if it consists of principal ideals.

Note that in case of record constructor only, that is in the relational datamodel, all keys are strong.

5.1 Record attributes with only one set component

Consider the following restricted record constructor: attribute $Y(Y_1, \ldots, Y_n)$, where $Y_1 = Y_1\{Y_1'\}$, while Y_i is not a set attribute for $i > 1$. Let X be obtained by repeated applications of this constructor. If $\mathcal{K}$ is a degenerate strong candidate key, then it is $X(Y^k\{\lambda\}, \lambda, \ldots, \lambda)\downarrow$ for some k, where $Y^k\{\lambda\}$ stands for $Y\{Y'\{Y''\{\cdots\{\lambda\}\cdots\}$ with k being the nesting depth of set constructors. Let $\mathfrak{S} = \{\mathcal{K}_1, \ldots, \mathcal{K}_m\}$ be

a strong candidate key system that contains a degenerate candidate key $\mathcal{K}_1 = X(Y^k\{\lambda\}, \lambda, \ldots, \lambda)\downarrow$. By the Sperner property, $\mathcal{K}_i = X(Y^{j_i}\{\lambda\}, Y_2^i, \ldots, Y_n^i)\downarrow$ with $j_i < k$ for $i > 1$. Let $k = i_0 > i_1 > \ldots > i_p$ be the set of distinct values of j_i's $i = 1, 2, \ldots, n$. Furthermore, let X' be the nested attribute $X'(Y_2, Y_3, \ldots, Y_n)$, that is the "set-free" component of X. Let $\mathfrak{K}_{i_m}$ be the set of principal ideals in $\mathfrak{S}(X')$ defined by $\mathfrak{K}_{i_m} = \{X'(Y_2^i, \ldots, Y_n^i)\downarrow | \; j_i = i_m\}$. Since $\mathfrak{S}$ is a Sperner system, $\mathfrak{K}_{i_m}$ is a Sperner system, as well. Also, if $i_f < i_g$, then for all $\mathcal{K} \in \mathfrak{K}_{i_f}$ and $\mathcal{K}' \in \mathfrak{K}_{i_g}$ we have that $\mathcal{K} \not\supseteq \mathcal{K}'$ holds.

Let $\mathcal{A} \in \mathfrak{S}^{-1}$ be a maximal candidate antikey. Since $X(Y^k\{\lambda\}, \lambda, \ldots, \lambda)\downarrow$ is a candidate key, every subattribute in $\mathcal{A}$ must have first component of form $Y^h\{\lambda\}$ for some $h < k$. Suppose that $X(Y^h\{\lambda\}, Y_2, \ldots, Y_n)$ and $X(Y^{h'}\{\lambda\}, Y_2', \ldots, Y_n')$ are two elements of $\mathcal{A}$. Using that $\mathcal{A}$ is an ideal we obtain that

$$X(Y^h\{\lambda\}, \lambda, \ldots, \lambda), X(Y^{h'}\{\lambda\}, \lambda, \ldots, \lambda) \in \mathcal{A}$$

holds. Clearly $X(Y^h\{\lambda\}, Y_2, \ldots, Y_n)$ and $X(Y^{h'}\{\lambda\}, \lambda, \ldots, \lambda)$ are reconcilable, and so are $X(Y^{h'}\{\lambda\}, Y_2', \ldots, Y_n')$ and $X(Y^h\{\lambda\}, \lambda, \ldots, \lambda)$. Thus by property 1 of Definition 16

$$X(Y^h\{\lambda\}, Y_2, \ldots, Y_n) \sqcup X(Y^{h'}\{\lambda\}, \lambda, \ldots, \lambda) = X(Y^{\max(h,h')}\{\lambda\}, Y_2, \ldots, Y_n) \in \mathcal{A}$$

holds. $X(Y^{\max(h,h')}\{\lambda\}, Y_2', \ldots, Y_n') \in \mathcal{A}$ is obtained by the same argument. Thus, the first components of the maximal elements of $\mathcal{A}$ are uniquely determined.

Proposition 39. *Let $Y^h\{\lambda\}$ be this unique first component of maximal elements of $\mathcal{A}$. Then $h = i_j - 1$ for some $0 \leq j \leq p$.*

Proof. Let us assume in contrary, that $i_{j+1} \leq h < i_j - 1$ for some j, and let $X(Y^h\{\lambda\}, Y_2, \ldots, Y_n)$ be a maximal element of $\mathcal{A}$. Since $Y^h\{\lambda\} > Y^{i_m}\{\lambda\}$ for all $m \geq j+1$, $X'(Y_2, \ldots, Y_n)$ cannot be larger than any element of $\mathfrak{K}_{i_m}$. Thus, denoting the projection of $\mathcal{A}$ onto the last $n-1$ components by $\mathcal{A}'$, then it is a candidate antikey (not necessarily maximal) for the (not necessarily Sperner) candidate key system $\mathfrak{K}_{i_{j+1}} \cup \ldots \cup \mathfrak{K}_{i_p}$. However, if $\mathcal{A}$ is enlarged by adding $X(Y^{i_j-1}\{\lambda\}, Y_2, \ldots, Y_n)$ and elements that must also be added by the ideal property for all maximal element $X(Y^h\{\lambda\}, Y_2, \ldots, Y_n)$ of $\mathcal{A}$, then by the observation above, the coincidence ideal obtained remains a candidate antikey, in contradiction with the maximality of $\mathcal{A}$. □

Proposition 39 gives a list of candidate antikeys of $\mathfrak{S}$ that contains all maximal candidate antikeys. For $0 \leq j \leq p$ take a system of maximal candidate antikeys of $\mathfrak{K}_{i_{j+1}} \cup \ldots \cup \mathfrak{K}_{i_p}$ $\mathcal{A}_{m_1}^j, \ldots, \mathcal{A}_{m_u}^j$, then extend each with first coordinate $Y^{i_j-1}\{\lambda\}$, finally add all elements of $\mathfrak{S}(X)$ that are under some of the obtained maximal elements.

Since $X(Y^k\{\lambda\}, \lambda, \ldots, \lambda)\downarrow$ is a candidate key, in an Armstrong-instance any two complex values must have distinct projections onto $X(Y^k\{\lambda\}, \lambda, \ldots, \lambda)$ that allows 2^k tuples at most.

6 Numerical Conditions

In the combinatorial investigation of Armstrong-instances of the relational model the following fundamental inequality of comparing the minimal size of Armstrong-instance for a minimal key system $\mathcal{K}$ and the size of the set of maximal antikeys $\mathcal{A} = \mathcal{K}^{-1}$ was proven by Demetrovics and Katona [7].

Lemma 40. *Let $R = (R_1, \ldots, R_n)$ be a relational schema, $\mathcal{K}$ be Sperner system of subsets of R. The minimum number of tuples $s(\mathcal{K})$ in an Armstrong-instance of minimal key system $\mathcal{K}$ satisfies*

$$|\mathcal{K}^{-1}| \leq \binom{s(\mathcal{K})}{2} \quad \textit{and} \quad s(\mathcal{K}) \leq |\mathcal{K}^{-1}| + 1. \tag{4}$$

The analog for the higher-order datamodel was given in [21] for the counter-free case. The same can be stated in the present general case, the similar proof is omitted. Let $\mathcal{S}(X)$ be the subattribute lattice of a nested attribute X. Furthermore let $\mathfrak{K}$ be a Sperner system of closed subsets of $\mathcal{S}(X)$. If $\mathfrak{K}$ has an Armstrong-instance as minimal key system, then $s(\mathfrak{K})$ denotes the minimum number of complex values in an Armstrong-instance of $\mathfrak{K}$. Otherwise, set $s(\mathfrak{K}) = \infty$.

Lemma 41.

$$|\mathfrak{K}^{-1}| \leq \binom{s(\mathfrak{K})}{2} \tag{5}$$

Using Theorem 30 an upper bound can be given in the case when $\mathfrak{K}$ does not contain degenerate keys.

Proposition 42. *Let $\mathcal{S}(X)$ be the subattribute lattice of a nested attribute X, and let $\mathfrak{K}$ be a Sperner system of closed subsets of $\mathcal{S}(X)$. Furthermore, assume that the conditions of Theorem 30 are satisfied. Then*

$$s(\mathfrak{K}) \leq 2|\mathfrak{K}^{-1}|. \tag{6}$$

Proof. The proof of Theorem 30 constructs two complex-valued tuples for each maximal antikey in $\mathfrak{K}^{-1}$.

□

6.1 Only degenerate keys

Having a degenerate key in the candidate key system gives a finite upper bound on the possible number of complex-valued tuples. If the candidate key system consists of only degenerate keys, then a lower bound for the number of maximal antikeys can be established. These two give necessary conditions for the existence of an Armstrong instance via Lemma 41.

Let us consider $\mathcal{S}\left(X\{X_1(A_1) \oplus X_2(A_2) \oplus \cdots \oplus X_n(A_n)\}\right)$ and let $\mathfrak{K}$ be a Sperner system of closed sets, with $\mathfrak{K} = \{X(X_v\{\lambda\} \mid v \in E)\downarrow \mid E \in \mathfrak{E}\}$, where $\mathfrak{E}$ is a Sperner system of subsets of $\{1, 2, \ldots, \}$.

Theorem 43. *Let $X = X\{X_1(A_1) \oplus X_2(A_2) \oplus \cdots \oplus X_n(A_n)$ and let $\mathfrak{K}$ be defined as above. If $\mathfrak{K}$ has an Armstrong-instance, then*

$$\sum_{\substack{V \text{ maximal independent set of} \\ \text{hypergraph } (\{1,2,\ldots,n\},\mathfrak{E})}} \max\left(2^{n-|V|-1}-1,1\right) \leq \binom{2^{\min(|E|:\, E\in\mathfrak{E})}}{2}. \tag{7}$$

Proof. In the proof of Proposition 32 it was shown that maximal candidate antikeys in $\mathcal{S}(X\{X_1(A_1) \oplus X_2(A_2) \oplus \cdots \oplus X_n(A_n))$ consist of principal ideals extended with some counter attributes. Thus, the system of candidate maximal antikeys $\mathfrak{K}^{-1}$ consists of coincidence ideals of type $\mathcal{A}_V^{\mathfrak{I}} = X(X_v\{A_v\} \mid v \in V)\downarrow \cup \{X_I \mid I \in \mathfrak{I}\}$. Here V is a *maximal independent vertex set* of the *hypergraph* (set system) $(\{1,2,\ldots,n\},\mathfrak{E})$ and $\mathfrak{I}$ is a set of at least two-element subsets of $\{1,2,\ldots,n\}$. Indeed, $\mathcal{A}_V^{\mathfrak{I}}$ contains $X(X_v\{\lambda\} \mid v \in E)\downarrow$ iff $E \subseteq V$. According to property 4(b) of Definition 16 $X(X_v\{A_v\} \mid v \in V)\downarrow$ has a coincidence ideal extension $\mathcal{A}_V^{\mathfrak{I}}$ for all *non-trivial* partition of I^- into two parts, provided $|I^-| > 1$. Since $|I^-| = n - |V|$, the number of such partitions is $2^{n-|V|-1}-1$. Thus, the left hand side of (7) is a lower bound of the number of candidate maximal antikeys. A key $X(X_v\{\lambda\} \mid v \in E)\downarrow$ allows at most $2^{|E|}$ distinct tuples. Applying Lemma 41, (7) follows.

□

7 Conclusions

In the present paper we investigated keys and antikeys in the presence of various constructors in the higher order datamodel. We proved that keys, as well as antikeys, correspond to certain ideals with additional closure properties. These are closed sets, that is intersections of coincidence ideals defined in [21], subsets of the subattribute lattice. We showed that the system of minimal keys correspond to Sperner system of closed sets and exhibited a sufficient condition when such a Sperner system occurs as a system of minimal keys. The candidate key systems not covered by the sufficient condition of Theorem 30 are the ones containing degenerate keys. A characterization when Armstrong-instance exists for such key systems is given in the (possibly) most important special case. Strong keys are also introduced. Some interesting combinatorial problems arose and we are intended to continue our investigations in that direction, as well. Another future direction of research is to refine the existing necessary, or sufficient conditions for Armstrong-instances, preferably to find characterizations in important special cases.

References

[1] Abiteboul, Serge, Buneman, Peter, and Suciu, Dan. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.

[2] Abiteboul, Serge and Hull, Rick. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62(1-2):3–38, 1988.

[3] Abiteboul, Serge, Hull, Rick, and Vianu, Victor. *Foundations of Databases.* Addison-Wesley, 1995.

[4] Armstrong, W. W. Dependency structures of database relationships. *Information Processing*, pages 580–583, 1974.

[5] Brightwell, G. and Katona, G.O.H. A new type of coding theorem. *Studia Scientiarum Mathematicarum Hungarica*, 38:139–147, 2001.

[6] Demetrovics, J. On the equivalence of candidate keys with Sperner systems. *Acta Cybernetica*, 4:247–252, 1979.

[7] Demetrovics, J. and Katona, G.O.H. Extremal combinatorial problems in relational data base. In *Fundamentals of Computing Theory (FCT 1981)*, number 117 in LNCS, pages 110–119. Springer-Verlag, Berlin, 1981.

[8] Demetrovics, J., Katona, G.O.H., and Sali, A. The characterization of branching dependencies. *Discrete Applied Mathematics*, 40:139–153, 1992.

[9] Demetrovics, J., Katona, G.O.H., and Sali, A. Design type problems motivated by database theory. *Journal of Statistical Planning and Inference*, 72:149–164, 1998.

[10] Fagin, Ronald. Horn clauses and database dependencies. *Journal of the Association for Computing Machinery*, 29(4):952–985, 1982.

[11] Fagin, Ronald and Vardi, M. Y. Armstrong databases for functional and inclusion dependencies. *Information Processing Letters*, 16:13–19, 1983.

[12] Ganter, B., Gronau, H.-D. O. F., and Mullin, R. C. On orthogonal double covers of k_n. *Ars Combinatoria*, 37:209–221, 1994.

[13] Hartmann, S. and Link, S. Reasoning about functional dependencies in an abstract data model. *Electronic Notes in Theoretical Computer Science*, 84, 2003.

[14] Hartmann, Sven, Hoffmann, Anne, Link, Sebastian, and Schewe, Klaus-Dieter. Axiomatizing functional dependencies in the higher-order entity relationship model. *Information Processing Letters*, 87(3):133–137, 2003.

[15] Hartmann, Sven, Link, Sebastian, and Schewe, Klaus-Dieter. Reasoning about functional and multi-valued dependencies in the presence of lists. In Seipel, Dietmar and Turull Torres, José María, editors, *Foundations of Information and Knowledge Systems*, volume 2942 of *Springer LNCS*. Springer Verlag, 2004.

[16] Hartmann, Sven, Link, Sebastian, and Schewe, Klaus-Dieter. Weak functional dependencies in higher-order datamodels. In Seipel, Dietmar and Turull Torres, José María, editors, *Foundations of Information and Knowledge Systems*, volume 2942 of *Springer LNCS*. Springer Verlag, 2004.

[17] Hartmann, Sven, Link, Sebastian, and Schewe, Klaus-Dieter. Functional dependencies over XML documents with DTDs. *Acta Cybernetica*, 17(1):153–171, 2005.

[18] Hartmann, Sven, Link, Sebastian, and Schewe, Klaus-Dieter. Axiomatisation of functional dependencies in the presence of records, lists, sets and multisets. *Theoretical Computer Science*, 355:167–196, 2006.

[19] Paredaens, J., De Bra, P., Gyssens, M., and Van Gucht, D. *The Structure of the Relational Database Model.* Springer-Verlag, 1989.

[20] Sali, Attila. Minimal keys in higher-order datamodels. In Seipel, Dietmar and Turull Torres, José María, editors, *Foundations of Information and Knowledge Systems*, volume 2942 of *Springer LNCS*. Springer Verlag, 2004.

[21] Sali, Attila and Schewe, Klaus-Dieter. Counter-free keys and functional dependencies in higher-order datamodels. *Fundamenta Informaticae*, 70:277–301, 2006.

[22] Sali, Attila and Schewe, Klaus-Dieter. Weak functional dependencies on trees with restructuring. Technical Report 4/2006, Massey University, Department of Information Systems, 2006. available from http://infosys.massey.ac.nz/research/rs_techreports.html.

[23] Schewe, Klaus-Dieter and Thalheim, Bernhard. Fundamental concepts of object oriented databases. *Acta Cybernetica*, 11(4):49–85, 1993.

[24] Thalheim, Bernhard. Foundations of entity-relationship modeling. *Annals of Mathematics and Artificial Intelligence*, 6:197–256, 1992.

[25] Thalheim, Bernhard. *Entity-Relationship Modeling: Foundations of Database Technology.* Springer-Verlag, 2000.

Received 6th October 2006

CONTENTS

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János