VŠB-Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

2^{nd} Workshop

DATESO '02

Databases, Texts, Specifications, and Objects

Ostrava 2002

Depertment of Computers, Czech Technical University, Prague
Department of Software Engineering, Charles University, Prague
Department of Computer Science, VŠB-Technical University of Ostrava
ČSKI, OS Computer Science and Society

Czech Republic

Proceedings of the Workshop

DATESO '02

Databases, Texts, Specifications, and Objects

April 17-19, 2002
Desná – Černá Říčka
Czech Republic

editor Michal Krátký

DATESO '02 (Proceedings of the Workshop Databases, Texts, Specifications, and Objects), April 17-19, 2002 Desná – Černá Říčka, 1^{st} edition – Ostrava – VŠB-Technical University of Ostrava, Ostrava-Poruba, tř. 17. listopadu 15, 708 33, (Print: Repronis, Ostrava, Nádražní 53), page count 114.

ISBN 80-248-0080-2

Preface

DATESO'2002, the workshop on current trends on Databases, Information Retrieval, Algebraic Specification and Object Oriented Programming, was held on April 17 - 19, 2002 in Desná - Černá Říčka. This was the second annual workshop organized by FEL ČVUT Praha, Department of Computer Science and Engineering, MFF UK Praha, Department of Software Engineering and VŠB-Technical University Ostrava, Department of Computer Science. The DATESO aims for strengthening the connection between this various areas of informatics. The proceedings of DATESO'2002 are also available at Web site http://www.cs.vsb.cz/dateso.

We are also thankful to the Organization Committee.

March 2002 Václav Snášel

Program Committee

Jaroslav Pokorný, *chair*	Charles University, Prague
Karel Richta	Czech Technical University, Prague
Václav Sklenář	Palacky University, Olomouc
Václav Snášel	VŠB - Technical University of Ostrava

Organization Committee

Yveta Geletičová	VŠB - Technical University of Ostrava
Michal Krátký	VŠB - Technical University of Ostrava
Tomáš Skopal	VŠB - Technical University of Ostrava

Table of Contents

Move to front coding based on splay trees

Jiří Dvorský, Václav Snášel

Department of Computer Science, Technical University of Ostrava,
17. listopadu 15, Ostrava - Poruba, Czech Republic

e-mail: jiri.dvorsky|vaclav.snasel@vsb.cz

Abstract. In 1994 Burrows and Wheeler presented a new algorithm for loss less data compression. The compression ratio that can be achieved using their algorithm is comparable with the best-known algorithms, whilst its complexity is relatively small. In this paper we shortly explain the principles of this algorithm and its word- based version. Word-based version is useful for fulltext systems for storing of textual databases. Main goal of this paper is fast Splay Tree Move to Front coding which appears as the second step of compression algorithm. Experimental results are also presented.

1 Introduction

In 1994 Burrows and Wheeler [Bur94] presented a data compression algorithm based on the Burrows-Wheeler Transform (BWT). Its compression ratios were comparable with the ones obtained using the best statistical methods.

In the first part of this paper we define the compression algorithm and the BWT, which is a base for it. We further shortly discuss the word-based version of this algorithm. The second step of the compression algorithm is Move To Front coding. For small alphabets it is easy to implement this type of coding. But for word-based alphabet or other large alphabet it is problem. Straightforward implementation has $O(n^2)$ time complexity. We propose to adopt Splay Trees [Sle83] to achieve $O(\log n)$ complexity. In the end we present experimental results.

2 Burrows-Wheeler Compression Algorithm

2.1 Definitions

In this section we introduce some definitions and notations, which are necessary to precisely describe the Burrows-Wheeler Transform (BWT).

Let us assume that $x = x_1 x_2 \dots x_n$ is a *sequence*. The sequence *length* denoted by n is a number of elements in x, and x_i denotes the i-th element of x. We define the *reverse sequence*, x^{-1} as $x^{-1} = x_n x_{n-1} \dots x_1$. The element x_i belongs to a finite

ordered set $\mathcal{A} = \{a_0, a_1, \ldots, a_{k-1}\}$ that is called *alphabet*. The number of elements in $\mathcal{A}$ is the size of the alphabet and is denoted by k. The elements of the alphabet are called *symbols* or *characters*.

2.2 Compression Algorithm

In this section we provide an insight description of the steps performed by the Burrows-Wheeler compression algorithms [Bur94]. Let the input data of the compression algorithms be a sequence of x of length n. First we have to compute the BWT. To achieve this we create n sequences by rotating x by one symbol. The n sequences created in this way are put into $n \times n$ matrix $M(x)$:

$$M(x) = \begin{bmatrix} x_1 & x_2 & \ldots & x_{n-1} & x_n \\ x_2 & x_3 & \ldots & x_n & x_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n-1} & x_n & \ldots & x_{n-3} & x_{n-2} \\ x_n & x_1 & \ldots & x_{n-2} & x_{n-1} \end{bmatrix}$$

Then matrix $M(x)$ is transformed into $\widetilde{M(x)}$ by sorting its rows in lexicographic order. Let $R(x)$ denote the row of sequence x in $\widetilde{M(x)}$. The results of the BWT are: $R(x)$ and the last column of matrix $\widetilde{M(x)}$ that we denote by x^{bwt}.

Once the BWT is completed, x^{bwt} is encoded using Move To Front (MTF)[Bent86] transform. The coding proceeds as follows. First the list $L = (a_0, a_1, \ldots, a_{k-1})$ consisting of the symbols of the alphabet $\mathcal{A}$ is created. Then to each symbol of x_i^{bwt}, $i = 1, 2, \ldots, n$, a number p_i is assigned such that x_i^{bwt} is equal to the p_i-th element of L, and the p_i-th element is moved to the beginning of list L. As the result we obtain a sequence x^{mtf} over the alphabet $\mathcal{A}_{mtf}$ consisting of integer number from range $[0, k-1]$.

In last step, sequnce x^{mtf} is compressed using a universal entropy coder, which could be the Huffman or arithmetic coder. The number $R(x)$ is also coded as binary number of $\lceil \log_2 n \rceil$ bits.

2.3 Decompression algorithm

The reverse Burrows-Wheeler Transform is based on the observation that sequence x^{bwt} is a permutation of sequence x and its sorting gives the first column of matrix $\widetilde{M(x)}$ that is the first character of a context by which the sequence x^{bwt} is sorted. Therefore given a symbol x_i^{bwt}, a symbol s located in the first column and i-th row of matrix $\widetilde{M(x)}$ can be found. Knowing that this is the j-th occurrence of s in the first column of matrix $\widetilde{M(x)}$ we find its j-th occurrence in the last column. Moreover the symbol s precedes x_i^{bwt} in sequence x. Thus if we know $R(x)$ we also know the last character of the sequence x, i.e. $x_{R(x)}^{bwt}$. Starting from this character we can iterate in similar manner to restore the original sequence x in time $O(n)$.

N	$R(N)$	$F(N)$
1	1	11
2	10	011
3	100	0011
4	101	1011
5	1000	00011
6	1001	10011
7	1010	01011
8	10000	000011
16	100100	0010011
32	1010100	00101011

Table 1: Example of Fibonacci codes

3 Fibonacci Codes

Fibonacci codes are based on Fibonacci numbers. These numbers are defined recursively as

$$\begin{aligned} F_0 &= F_{-1} = 1 \\ F_n &= F_{n-1} + F_{n-2} \text{ for n} \geq 1. \end{aligned}$$

Theorem 1. A ny positive integer number N can be expressed as

$$R(N) = \sum_{i=0}^{k} b_i F_i,$$

where $b_i \in \{0, 1\}$ i.e. binary digit, $k \leq N$, and F_i for $0 \leq i \leq k$ are Fibonacci numbers.

This representation has interesting property that the string $b_0 b_1 \ldots b_k$ does not contain any adjacent 1's. Fibonacci code for integer N is defined as

$$F(N) = b_0 b_1 b_2 \ldots b_k 1$$

In other words Fibonacci code is reversed binary representation $R(N)$ and one additional 1 at the end of binary string. Resulting binary words forms prefix code. Examples of some small codes are given in table 1.

4 WBW method

Origin of compression method, which is called WBW (Word based Burrows-Wheeler), was inspired by Huffword [Wit94] and WLZW [Dvo99], which apply traditional compression algorithms on words. The WBW method arises from concept that words occurs in text in some manner, which is not random, but it follows certain rules. The rules are given by used language and also by contents of text itself.

4.1 Definition of words and nonwords

The alphabet of the WBW consists of words and nonwords. We called them *tokens* together. A *word* is defined as maximal string of alphanumeric characters (letters and digits) and *nonword* is defined as maximal string of other characters (punctuations and spaces).

4.2 Basic Principle

The algorithm we have implemented uses two passes because of the archival nature of textual databases. The text of a document is stored only once and is then read multiple times. This makes it possible to choose a better compression ratio, even at the price of two passes and lengthier compression. Another consideration is that computers used for constructing collections of documents are usually more powerful than client computers accessing the finished collections.

The compression consists of four phases:

1. lexical analysis – input text is divided into words and nonwords. Alphabet for BW algorithm is now created,
2. storing of the alphabet
3. BW transformation,
4. Move To Front coding,
5. storing of the MTF results by appropriate method.

Decompression is inverse process to compression.

5 Move to Front coding

The last phase of the compression (see section 2.2) is Move To Front coding. It turned out that classical MTF is unusable (very slow) for large sized alphabet. Therefore the new method should be developed.

6 Splay Tree

A Splay tree ([Sle83, Sle86]) is a Binary Search Tree on which splaying operations are performed. A splaying operation is one that causes a node in the tree to move up the tree and become the root of the tree. Splaying of a node is done by rotating the nodes a more detailed description of the rotations will be given bellow. Whenever a node is accessed via standard tree operations (Find, Insert, Remove, etc.), it is splayed, it is splayed thus making it the root.

Because of moving the accessed node to the root via rotation, the height of the tree may be shortened or lengthened. However, while splaying the accessed node to the root the height of all other nodes on the access path are just about halved. Even though the total height may be infact lengthened the height of the nodes off the access path will be increased by no less then two. Splay trees guarantee amortized complexity $O(m \log n)$.

Searching (operation Find) in splay tree is important for our purposes. Splaying is done every time a Find operation is done. First a regular binary search tree algorithm is executed. If the node is found tree is splayed. If no node is found, then the inorder successor on the search path is splayed in accord with the splay rules described above. The node splayed is dependent on the structure of the tree, as trees that have the same values may have a different inorder structure.

We propose using of Splay tree to perform Move To Front coding of BWT results. Initially the whole alphabet is inserted into the tree. MTF coding is a sequence of searching on the tree. Therefore all codes from BWT can be founded in the tree. The first step of Find operation in the splay tree is standard binary search tree algorithm. A path from the root of the tree to the searched node is considered as code (binary string), that identifies the node. Unfortunately this code isn't prefix code or block code, so it cannot be unambiguously decoded. For example 0 is code for the first left child of the root but 00 is code for the leftmost child of the root in the second level. And node in the root of the tree has code equal to empty string. Due this reasons one 1 is added to beginning of all codes (e.g. 10 - the first left child of the root, 100 the leftmost "grandson" of the root). This numbers are encoded as Fibonacci codes.

The splay tree moves frequently used node near to root of the tree and therefore path from root to particular node is relative short. By this way compression is achieved. But sometimes less frequent node must be searched in the tree. The binary string produced by encoding algorithm will be very long. It is usually longer than $\log_2 n$, where n is size of alphabet, bits that are necessary to encode token number of encoded token. It doesn't represent any compression. The binary string should be transformed into Fibonacci code in the second step. The transformation of such binary number can be done with large-scale integer arithmetic. Performance of this type of arithmetic is low when it is compared with speed of hardware integer arithmetic in processor. If we think matter over, it isn't necessary to encode the large number by Fibonacci code and cope with problem of large-scale arithmetic. Problem can be solved by escape code mechanism. If length of binary string produce by splay tree is greater than given threshold then escape code is written into output at first and then token number at second. Node is splayed to the root of the tree of course. In other case binary string is transformed into Fibonacci code and then it is written into output. The threshold could be set up to the size of processor word (e.g. 16, 32 or 64 bits). Then length of binary string doesn't cross size of hardware arithmetic and could be easily transformed in both directions. He escape code must be different from all possible codes produced by splay tree. Splay tree produces binary codes (strings) from 1 to $1^k \in \{0,1\}^k, k \in N$. The smallest code for the threshold is number with binary representation 10^k, i.e. the largest code plus one.

Example 1. W e usually have 32-bit processor, then the largest code has binary string that contains 31 ones i.e. $2^{31} - 1$. Therefore threshold is equal to 2^{31}.

	bible
Size of files [bytes]	4047392
Number of tokens	1535711
Number of unique tokens	13509
Number of words	767855
Number of unique words	13456
Number of nonwords	767856
Number of unique nonwords	53

Table 2: Characterization of experimental file

7 Experimental Results

7.1 Data for experiments

We perform our test on file *bible*. The file is part of Canterbury Compression Corpus (see [Arn97]) intended for testing compression algorithms.

7.2 Experiments

In our test we observe impact of threshold in Splay tree Move to front coding scheme on size of compressed text. A number of escape codes and average length of code (it was measured in bits per token) was displayed too. For experimental see table 3 and charts 1, 2 and 3. We also test how much depends results of compression on initial structure of the splay tree. It means which permutation should be inserted into the splay tree. We experiment with identical permutation, anti permutation and with random permutation. You can see that results for particular permutations are too close, so the size of compressed text – our main object of interest - doesn't depend on initial permutation.

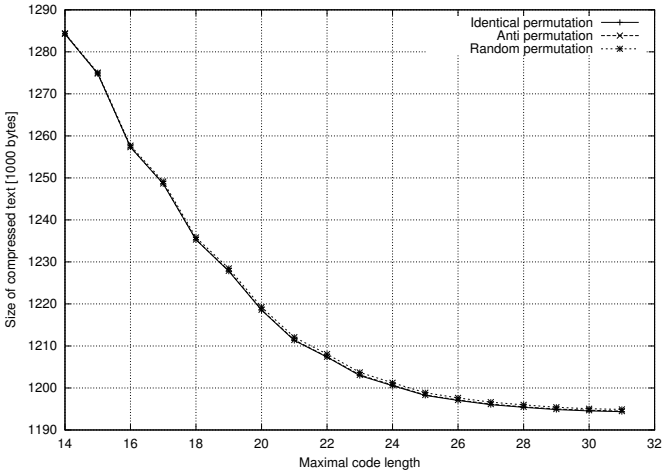

Fig. 1: Size of compressed text with respect to threshold

Maximal Code Length	Size of compressed text			Escape codes			Bits per token		
	Type of permutation			Type of permutation			Type of permutation		
	identical	anti	random	identical	anti	random	identical	anti	random
14	1284226	1284287	1284406	89955	89977	90029	6.690	6.690	6.691
15	1274818	1274829	1275061	72757	72749	72865	6.641	6.641	6.642
16	1257328	1257321	1257671	58063	58045	58214	6.550	6.550	6.552
17	1248584	1248666	1249133	45660	45693	45913	6.504	6.505	6.507
18	1235293	1235326	1235825	35186	35191	35410	6.435	6.435	6.438
19	1227782	1227886	1228464	26609	26649	26883	6.396	6.396	6.399
20	1218521	1218620	1219235	19692	19727	19953	6.348	6.348	6.351
21	1211285	1211432	1212080	14315	14372	14589	6.310	6.311	6.314
22	1207331	1207437	1208062	10243	10271	10463	6.289	6.290	6.293
23	1202948	1203127	1203701	7111	7178	7316	6.267	6.267	6.270
24	1200543	1200670	1201235	4873	4903	5026	6.254	6.255	6.258
25	1198213	1198331	1198872	3253	3275	3376	6.242	6.242	6.245
26	1197033	1197152	1197626	2197	2217	2284	6.236	6.236	6.239
27	1196026	1196152	1196634	1492	1514	1556	6.230	6.231	6.234
28	1195392	1195505	1195971	968	980	1002	6.227	6.228	6.230
29	1194850	1194997	1195415	604	634	621	6.224	6.225	6.227
30	1194529	1194686	1195045	385	418	390	6.223	6.223	6.225
31	1194362	1194501	1194886	243	262	238	6.222	6.223	6.225

Table 3: Experimental results

8 Conclusion

Adaptation of Burrows-Wheeler compression algorithm for word-based compression was presented. More detailed experimental results are presented in paper [Dvo01]. Another application of splay trees to data compression can be found in [Jon88]. Compression ratio is about 30 percent, it may be considered as good.

Rereferences

[Arn97] R. Arnold and T. Bell. A corpus for evaluation of lossless compression algorithms. In Proceedings Data Compression Conference 1997, 1997. `http://corpus.canterbury.ac.nz`.

[Bent86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. Communications of the ACM, 29(4):320–330, 1986.

[Bur94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center Research Report 124, 1994.

[Dvo01] J. Dvorský and V. Snášel. Modifications in burrows-wheeler compression algorithm. In Proceedings of ISM 2001, 2001.

[Dvo99] J. Dvorský, V. Snášel, and J. Pokorný. Word-based compression methods

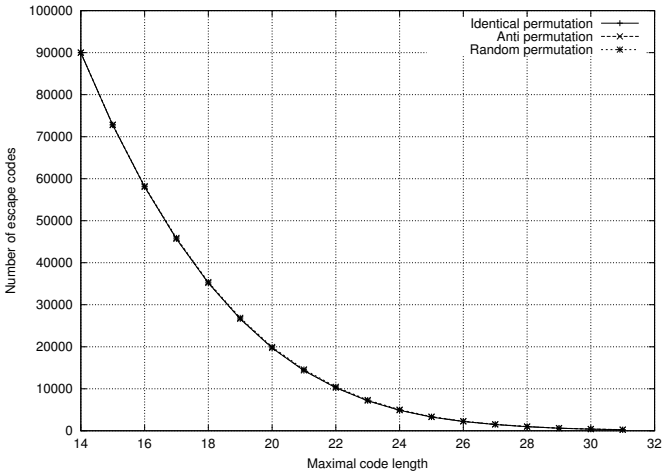

Fig. 2: Number of escape codes with respect to threshold

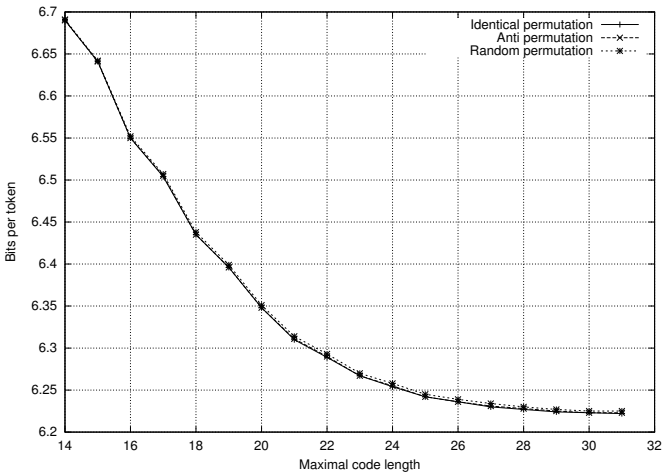

Fig. 3: Bits per token with respect to threshold

and indexing for text retrieval systems. In Lecture Notes on Computer Science 1691. Springer-Verlag Berlin, 1999.

[Jon88] D. W. Jones. Application of splay trees to data compression. Communication of the ACM, 31(8):996–1007, 1988.

[Sle83] D. D. Sleator and R. E. Tarjan. Self-adjusting binary tree. Proceedings of the ACM SIGACT, pages 235–245, 1983.

[Sle86] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search tree. Journal of the ACM, 32(3):652–686, 1985.

[Wit94] I. Witten, A. Moffat, and T. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. Van Nostrand Reinhold, 1994.

Multi-Agent Reactive Scheduling in SDL[1]

Petr Olmer

Department of Theoretical Computer Science and Mathematical Logic
Charles University, Prague

e-mail: petr.olmer@mff.cuni.cz

Abstract. Reactive algorithms expect their input data to be added and/or changed. Computational systems working on this base are known as reactive agents. SDL is the programming language that can specify and describe a reactive multi-agent system, where relations between agents, their behaviour and communication ways are well defined. We describe how to get the benefit of SDL in area of scheduling. We present three SDL scheduling systems that show a development from the traditional (static) scheduling to a distributive reactive system without any central unit.

Key words: Multi-agent systems, reactive agents, reactive scheduling, SDL.

1 Introduction

This paper presents an application of SDL (Specification and Description Language) in area of multi-agent systems for scheduling. Our research is aimed at using SDL for modelling and specifying multi-agent system in general. We are not concerned in the high-level development of scheduling algorithms, nor in the research in static scheduling. The work is in progress; we present a partial solution and ideas in the paper. The work is a part of author's PhD. research on multi-agent systems.

We define a scheduling problem as follows: A system consists of machines processing jobs. Each job is defined by its name, a machine that can process this job, a time that the machine spend on this job, and by a set of before-jobs, i.e. jobs that must be done before the given job can be processed. Each machine can process only one job at time. We say that a job is scheduled, if the time when the job starts is known. We search for a scheduling of all given jobs. There is often a request of being optimal in defined criteria. This paper is not the case. We only demand scheduling of all jobs, if possible (i.e. no dead-locks).

We talk about a static algorithm, if all input data are known before the algorithm starts. An on-line algorithm allows a part of input data to be known even during running of the algorithm. At any time, an on-line algorithm assumes that the given

[1] This work was done while the author worked in Agit AB.

input data are complete. A reactive algorithm describes how reactive agent behaves: it expects the world round (i.e. input data) to be changed, and it reacts in sense of changing its mind with respect to these changes.

Agent is an over-used term. In the context of this paper, we understand agents as long-lived computational systems, which have goals and decide autonomously which actions to take in the current situation to maximize progress toward its goals. As we have done above, we can talk about agents mind, about the world around, about agents feelings and intentions. Multi-agent system (MAS) is an environment where agents live, communicate and collaborate.

The programming language SDL is an object-oriented, formal, and graphical language for specification and description of complex event-driven (reactive) systems [Olm1]. Since 1976, SDL is an ITU-T standard; the last revision is called SDL-2000 [Z.100]. The language has broad possibilities of use [Olm2], from traditional ones like communication protocols (GSM, GPRS, UMTS, Bluetooth) or telecommunication software (Nokias firmware) even to workflow modelling [OV]. To our knowledge, SDL has not been used to develop reactive scheduling algorithms.

In section 2, a part of SDL standard is presented. In section 3, we describe three SDL multi-agent scheduling systems. The first one is a naive implementation of greedy algorithm with a master agent in static scheduling. The second one is a reactive version of the previous system. It shows that reactive systems can be easily described in SDL. The last one presented system specifies a multi-agent reactive scheduling system without a master agent. Machine agents communicate among themselves in the system. Section 4 concludes and gives ideas for further research.

2 SDL

SDL system consists of agents, i.e. processes running in parallel. Behaviour of each process is specified by an extended state machine. The syntax entities related to the paper are described in the figure 1. Agents communicate among themselves and with system environment via concept of signals. Agents are objects. An object-oriented approach can be used.

In SDL, there is no global data. This approach requires that information between processes, or between processes and the environment, must be sent with signals and optional signal parameters. Signals are sent asynchronously, that is, the sending process continues executing without waiting for an acknowledgment from the receiving process.

Processes in SDL can be created at system start, or can be created and terminated dynamically at runtime. More than one instance of a process can exist. Each instance has a unique process identifier (PId). This makes it possible to send signals to individual instances of a process.

In the following SDL systems, we use a built-in data type called powerset. The only literal for a powerset sort is empty, which represents a powerset containing no elements. The following operators are available for a powerset data type: *in* (tests

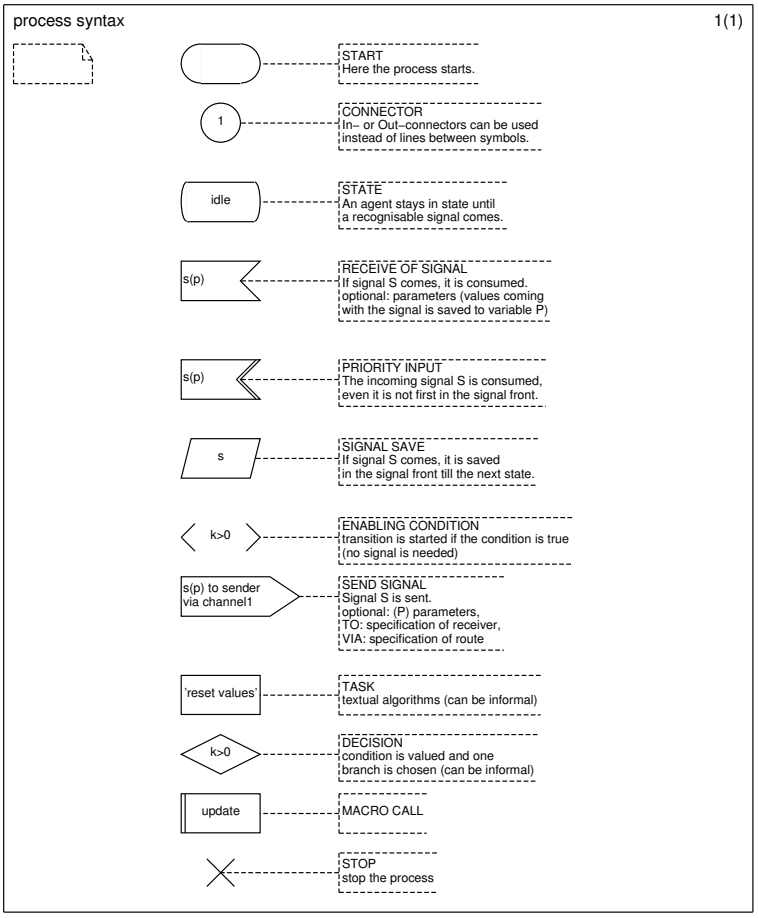

Fig. 1: SDL syntax entities.

if a certain value is member of the powerset or not), *incl* (includes a value in the powerset), *del* (deletes a member in a powerset), *length* (the number of elements in the powerset), *take* (returns one of the elements in the powerset, it can be specified which one they are implicitly numbered).

3 SDL scheduling systems

In this section we describe three SDL scheduling systems. In all three systems, each machine has its machine agent. The agent schedules the given jobs. During processing a job, no other actions are taken by the agent. The algorithm of scheduling is the simplest one in the real time: the first possible job is chosen. We can call the algorithm *greedy*.

There is no negotiating between machine agents. The time horizon is zero; it means that jobs are not scheduled into the future.

In the first two systems, we make use of a concept of master agent. The master agent does not schedule any jobs. It serves as a public blackboard for machine agents. There is no direct communication channel between machine agents; all communication channels are between a machine agent and the master agent.

3.1 Static scheduling with the master agent

The system is on figures 2 and 3.

A machine agent starts in the state *warm_up*. It waits for all input data, i.e. all jobs to be processed (*jobs*). When the signal *input_jobs* comes, the machine agent start scheduling. For each given job, it asks the master agent, whether its before-jobs are done (signal *ask*). The response can be *yes* or *no* for each before-job. If all answers were *yes*, the job is processed. To process a job means to delete the job from jobs, to set a timer, and, after the timer expiration, to announce to the master agent that the job is finished (signal *done*).

If any of the answers was *no*, the next job is tested. If all jobs were processed, the machine agent ends. If any jobs cannot be processed (because of its before-jobs), the machine agent waits for one time unit, and then tests all the jobs again.

The master agent starts in the state *idle*. It waits for signals *ask* and *done*. Signal *ask* requests a query to the set of finished jobs (*donejobs*). If the given job is finished, the signal *yes* is sent to the sender, otherwise the signal *no* is sent to the sender. Signal *done* updates *donejobs*.

We see the disadvantages of this system in two areas. First, the system is inefficient, because it traverses a set of jobs to be done again and again. If a job can start at time n, its machine agent will ask the master agent at least n times, whether the job can start. Second, behaviour of machine agents is not clear, because it supplies work of the master agent when asking before-jobs one by one. Both disadvantages are solved in the next system. In addition, the second system is even reactive.

3.2 Reactive scheduling with the master agent

The system is on figures 4 and 5.

A machine agent starts in the state *idle*. It waits for signals *new_job*, and *job_done*. Signal *new_job* (from the environment) announces a new job *nj* to be processed. Names of its before-jobs (*bj*) are sent to the master agent. The response is *all_done* or *wait*. The signal *all_done* means that all given before-jobs have already been done, and the job can be processed. The signal *wait* means that some before-jobs (*new_bj*) were not processed yet. The machine agent will be announced via *job_done* when any of these jobs is done. The set of before-jobs is updated (jobs already done are deleted), and the job is saved to *jobs* (the set of jobs to be done).

Signal *job_done* from the master agent announces that another machine has finished a job *dj*. The signal is sent to all machine agents known by the master agent. All jobs from the set *jobs* are tested in the same way: if the finished job was the last member of before-jobs of any job *cj*, then the job *cj* is processed. To process a job means to delete the job from jobs, to set a timer, and, after the timer expiration, to announce to the master agent that the job is finished (signal *job_done*).

The master agents starts in the state *idle*. It waits for signals *test_done*, and *job_done*, both from machine agents. Signal *test_done* requests a test of set of jobs *js*, whether they are already done or not. First, the PId of sender (a machine agent) is

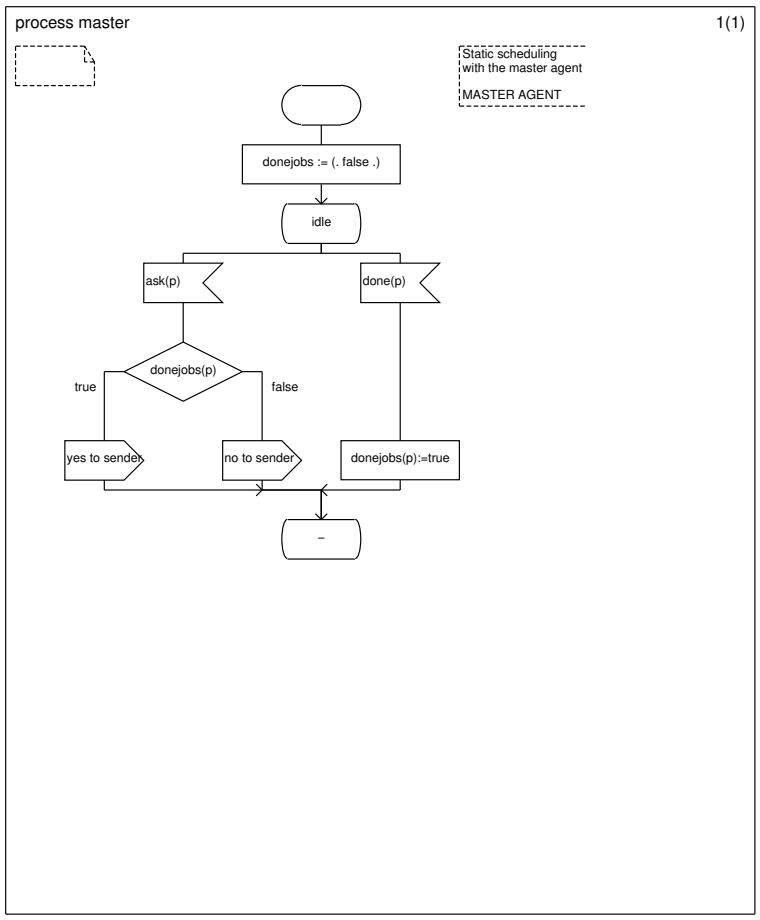

Fig. 2: Static scheduling with the master agent: MACHINE.

saved to *known* (set of known machine agents). Second, all finished jobs *fj* are deleted from the given set of jobs. If the set *js* remains empty, the signal *all_done* is sent to the sender, otherwise the signal *wait* with the set of remained jobs is sent to the sender.

Signal *job_done* from a machine agent announces that this machine has finished a job *j*. The job *j* is added to the set of finished jobs *fj*. Then, the signal *job_done* is resent to all known machine agents.

3.3 Reactive scheduling without a master agent

The system is on figure 6.

The master agent in the previous scheduling system maintains a "database" of finished jobs and a "database" of machine agents in the system. When a job was finished, an announcement was sent to all machine agents. In the system without a master agent, each machine agent maintains its own databases of jobs and known machine agents (powerset *known*). Jobs already finished by any agent are saved in its powerset *f_jobs*. Jobs to be done by the agent are saved in *w_jobs*. Before-jobs to be done by another known agent are saved in *r_jobs*. Before-jobs to be done by an unknown agent are saved in *u_jobs*.

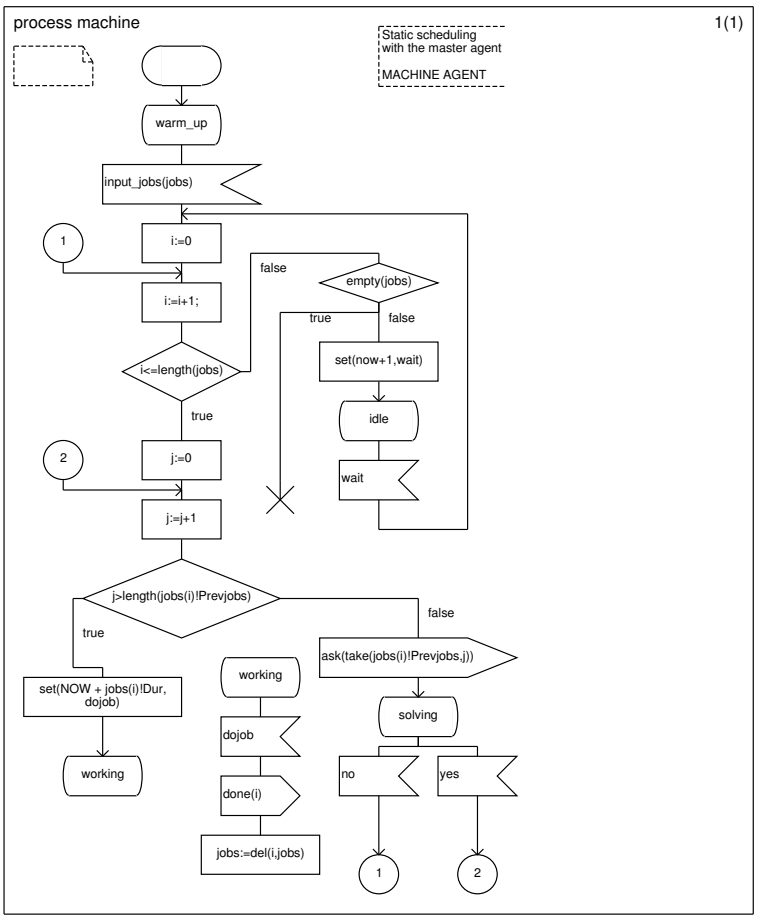

Fig. 3: Static scheduling with the master agent: MASTER.

An agent in this system can say about a given job one of these three statements: the job is mine and it is done (it is in its *f_jobs*), the job is mine and it is not done yet (it is in its *w_jobs*), and the job is not mine.

When an agent starts, it sends signal *hello* to a random agent. For the first agent in the system, the timer *init_timer* expires. If the signal reaches another agent, it replies with the signal *welcome* and saves its PId. The new agent also saves the PId of the replier (macro *update_kn* updates *known*). By this, the new agent is classed to the communicating system. Now it can ask other known agents about their meaning of given jobs. Each machine agent has only partial information about the world (agents, jobs) round (at the very start, it knows only one other agent).

The main agent's state is called *idle*. In the state, these signals are expected: *hello*, *new_job*, *ask*, *done*, and *wait*. The signal *hello* comes from a new agent; the *welcome* message is sent back.

The signal *new_job* comes from the environment. Macro *save_job* saves a given job *nj* to *w_jobs* and classifies its before-jobs to *f_jobs*, *r_jobs*, and *u_jobs*. If all before-jobs are finished, the given job can be done too.

The signal *ask* comes from another agent with three parameters: *job* (job's name), *who* (PId of agent requesting the job), and *not_to* (set of PIds where this signal already was). If an agent receiving *ask* has the job in *f_jobs*, it replies with the signal *done* (no

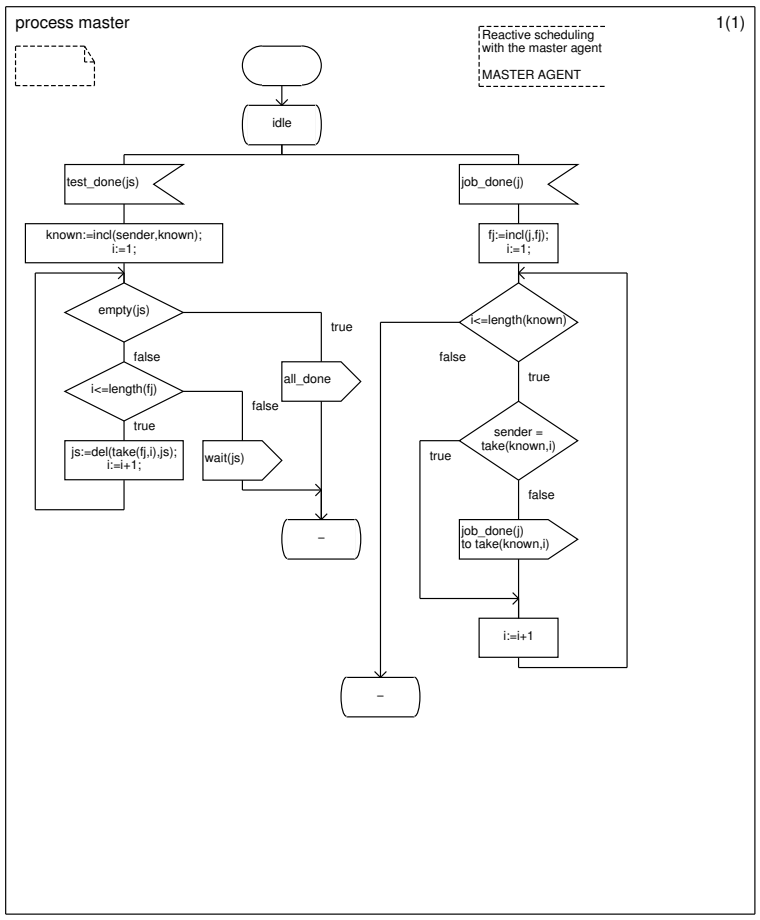

Fig. 4: Reactive scheduling with the master agent: MACHINE.

matter who has finished the job). If the job is in *w_jobs*, the agent replies with the signal *wait* that means that it will let the agent *who* know when the job is finished. In both cases, the reply is sent to the agent *who*, not to the sender that can be only distributing the signal *ask* among its known agents. Additionally, the agent *who* is added to the set *known*.

The job can be also in *r_jobs*. It means that the agent is also waiting for this job to be finished. In the case, the signal *ask* is immediately resend to the agent *w* responsible for the job. The last case is that the agent does not know anything about the job; the macro *distrib_ask* is called. The agent added its PId to the set *not_to* and resent the signal *ask* to agents that are in *known* but not in *not_to*.

The signal *done* announces that a job from $^\circ$_jobs is finished. In macro *try_to_work* the agent tries to find a job in *w_jobs* with all its before-jobs finished and schedule the job (see the signal *job_done* in reactive scheduling with the master agent). The sets *f_jobs*, *r_jobs* and *w_jobs* are updated.

If the signal *wait* comes, a job from *u_jobs* has found its agent. The sets *r_jobs* and *u_jobs* are updated.

The last transition from the state *idle* is by an enabling condition: If *u_jobs* is not empty, the agent tries to locate jobs in this set by sending the signal *ask* (*who* is equal

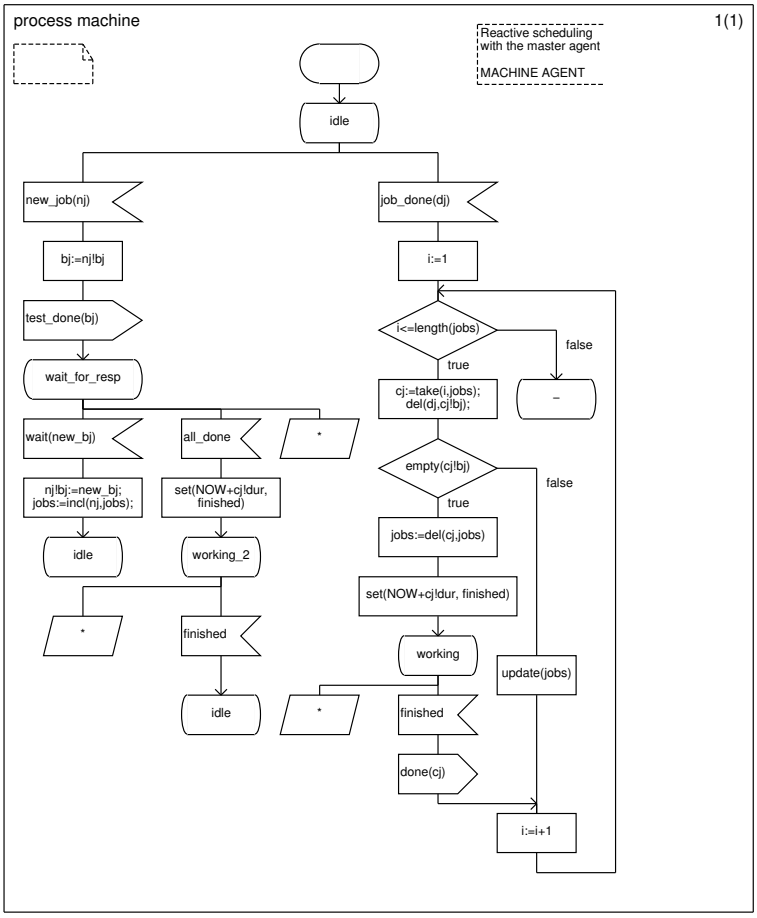

Fig. 5: Reactive scheduling with the master agent: MASTER.

to its PId) to its known agents. Possible responses are *done*, *wait*, or no response, if the relevant agent was not found.

4 Conclusion

We have presented a way in which SDL can be used in area of scheduling. The language SDL is suitable for specification of reactive scheduling system. There were three systems proposed in the paper: the first one was static (non-reactive) scheduling system, where all the input data were known before scheduling. The advantages of SDL were demonstrated in next two reactive systems. Requests of new jobs are received during a scheduling process in these systems. The second system includes the master agent that collects and distributes public information. The reactive multi-agent system can also work without any master agent; we have shown it in our third system. There are only machine agents in this system, and they collaborate among themselves and extend their local knowledge about jobs and other agents.

Further research should be aimed at negotiating between machine agents with respect to optimise their local knowledge. Adding of a time horizon in scheduling is also a relevant feature of these reactive systems [Smi]. The proposed multi-agent

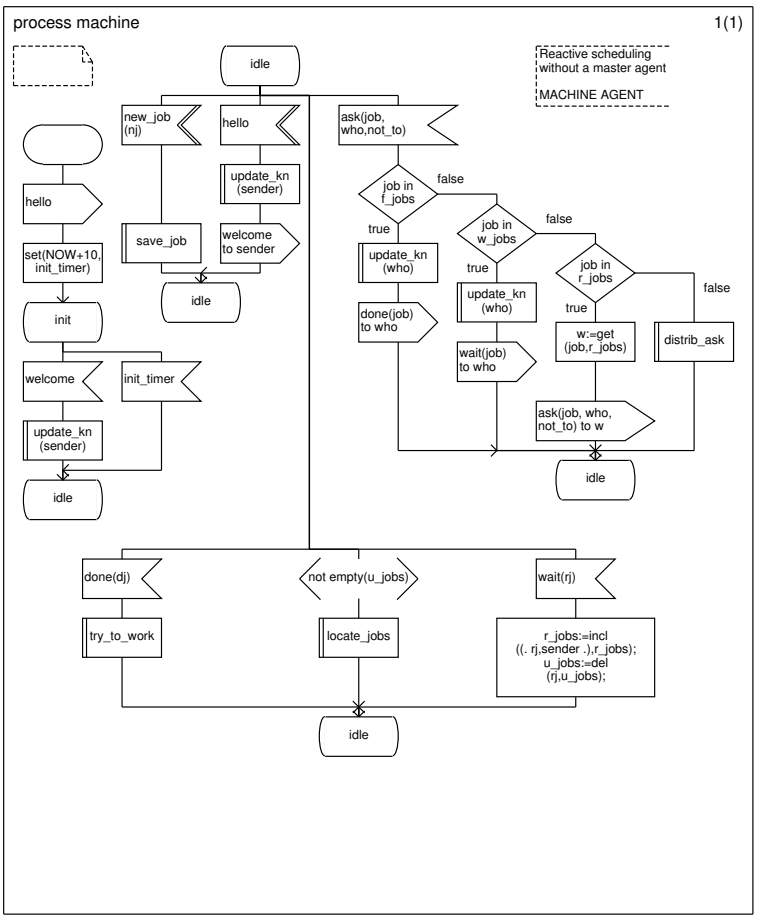

Fig. 6: Reactive scheduling without a master agent.

systems can be simulated and tested. Following-up their reactions in different worlds (the number of agents, their load) should be also an interesting issue.

Rereferences

[Olm1] Olmer P., *SDL-2000: Programovani rizene udalostmi*, In Proceedings of Objekty 2001, ISBN 80-213-0829-X.

[Z.100] *Z.100 Specification and Description Language (SDL)*, ITU-T, 2000.

[Olm2] Olmer P., *SDL: Chybejici clanek retezce?*, In Valenta (ed.), Proceedings of Workshop on Databases, Texts, Specifications, and Objects, DATESO '01, CVUT Praha, 2001, ISBN 80-01-02376-X.

[OV] Olmer P., Vrana C. *Workflow Application Development Using SDL and UML*, In G. Harindranath et al (eds.), New Perspectives on Information Systems Development: Theory, Methods and Practice, Proceedings of the 10th International Conference On Information Systems Development, ISD 2001, Kluwer Academic/Plenum Publishers 2002.

[FW] Fiat A., Woeginger G. J. (eds.), *Online algorithms, The State of the Art*, Lecture Notes in Computer Science, Vol. 1442, SPringer 1998, ISBN 3-540-64917-4.

[Smi] Smith S. F., *Reactive Scheduling Systems*, In D. Brown and W. Scherer (eds.), Intelligent Scheduling Systems, Kluwer Publishing 1994.

Application of Java Proxy Object to Handle Context Information

Roman Szturc

Dept. of Computer Science, VŠB-Technical University of Ostrava
tř. 17. listopadu, 708 33 Ostrava-Poruba, Czech Republic

e-mail: roman.szturc@vsb.cz

Abstract. Proxy objects are well-known from area of remote method invocation. Their application in "local computing" was inhibited mainly by lack of a good support for proxies in standard libraries. This paper presents application of Java proxy object in area of handling of context information.

Key words: Object-Oriented Programming, Proxy Object, Java

1 Introduction

Development of principles described in this paper was inspired by the following problem:

> There is a lot of systems which elements and connections form vertices and edges form an ordered graph. Systems traverse the graph and manipulates data held by vertices. A way how the data are manipulated may depend on many factors forming *context*. The context must be carefully handled by the system, often in thread-safe way.

1.1 Motivation

Experience have shown that although handling of context is usually groovy job, it is source of frequent and hard to detect errors. After several sleepless nights spent on searching errors emerging from incorrect context state, the author decided to rapidly change method how context is handled by programmers. Main goal was to find out a method how to handle context information in a *thread-safe*, *programmer-friendly* and *efficient* way.

1.2 Organization

This paper exemplifies handling of context information on well-understandable context factor—path the system went through to reach certain vertex. Since the system forms ordered graph, not a tree, there may be several paths leading to the same vertex. Then, a result of the same operation performed upon the vertex may differ according to actual path. In addition the vertex can be manipulated by several threads of execution simultaneously.

2 Actual State

Problem of handling of context information is not new. Two approaches are used in most cases:

- passing context information as a method argument,
- storing context information in a data pool of thread of execution.

2.1 Passing Context as an Argument of a Methods

The most strightforward way, how to handle context in a thread-safe way is to pass context in form of method argument. This approach is easy to implement but has big disadvantage. Nearly each method has to have an argument representing context, which makes source code dusty and makes programmer to handle context even if he or she is not interested with its content. Listing 1 outlines simple example of passing context as an argument.

Listing 1: Method `doSomething` with context argument.

```
public void doSomething(Context context) {
   // Select a node to operate upon it.
   int index = ...
   Node node = getSuccessor(index);
   // Adjust context, i.e., set correct path leading to node.
   String path = context.getPath();
   context.setPath(path + "/" + node.getName());
   node.doSomethingElse(context);
   context.setPath(path);
}
```

■

2.2 Passing Context in Thread's Data Pool

In order to avoid passing context as an argument of method it is possible to take advantage of storing information into data pool of thread of execution. This approach does not require context argument, because context is carried by actual thread of

execution and can be accessed at any time. It enables programmer to take care about context only in places, where context information is really necessary.

Application of this approach presupposes existence of a thread capable to carry context information. Since there is `Thread` class representing thread of execution in Java, such a class can be easily defined.

Listing 2: Definition of class `ContextThread`.

```
public class ContextThread extends Thread {
   private Context context;
   ...
   public Context getContext() {
      return context;
   }
}
```

■

Supposing that current thread of execution is kind of `ContextThread`, then method `doSomething` from listing 1 can be rewritten into the one presented in listing 3.

Listing 3: Method `doSomething` without context argument.

```
public void doSomething() {
   // Select a node to operate upon it.
   int index = ...
   Node node = getSuccessor(index);
   // Obtain context from current thread of execution.
   ContextThread thread = (ContextThread)Thread.currentThread();
   Context context = thread.getContext();
   // Adjust context, i.e., set correct path leading to node.
   String path = context.getPath();
   context.setPath(path + "/" + node.getName());
   // Perform doSomethingElse operation upon node adjusted context.
   node.doSomethingElse();
   context.setPath(path);
}
```

■

2.3 Robustness

Implementations listed in listings 1 and 3 are not robust enough, because in case of an unpredictable situation, for instance in case of occurence of an exception in method `doSomethingElse`, context is inconsistent and may cause whole system to become unstable. In order to ensure robustness, context must be restored to correct state regardless of process of invocation of method `doSomethingElse`.

The Java `try-finally` statement can be used to do that. Unfortunately, this solution makes source code even more hard to read, as exemplified in listing 4.

Listing 4: Ensuring context restoration.

```
public void doSomething() {
   // Select a node to operate upon it.
   int index = ...
   Node node = getSuccessor(index);
   // Obtain context from current thread of execution.
   ContextThread thread = (ContextThread)Thread.currentThread();
   Context context = thread.getContext();
   // Adjust context, i.e., set correct path leading to node.
   String path = context.getPath();
   try {
      context.setPath(path + "/" + node.getName());
      // Perform doSomethingElse operation upon node adjusted
      // context.
      node.doSomethingElse();
   }
   finally {
      context.setPath(path);
   }
}
```

■

This approach is fully functional and robust, however, has some disadvantages. Programmer has to ensure:

- modification of current context,
- invocation of a method in modified context and
- restoration of original context (after the method invocation).

The first item is often source of errors. Programmers usually forget to modify current context before invocation of a method. The method then may behave incorrectly, because it is executed in incorrect context. Similar problem concerns the third item. Programmers either restore context incorrectly or forget to restore the context at all.

2.4 Desired Solution

The goal is to ensure *automatic context adjustment* and relieve programmer of groovy context handling. Desired solution should allow programmer concentrate only on problem area and manipulate with context only when it is really necessary. Reduced code from listing 4 should look like the one given by listing 5.

Listing 5: Desired method content.

```
public void doSomething() {
   // Select a node to operate upon it.
   int index = ...
   Node node = getSuccessor(index);
   node.doSomethingElse();
}
```

■

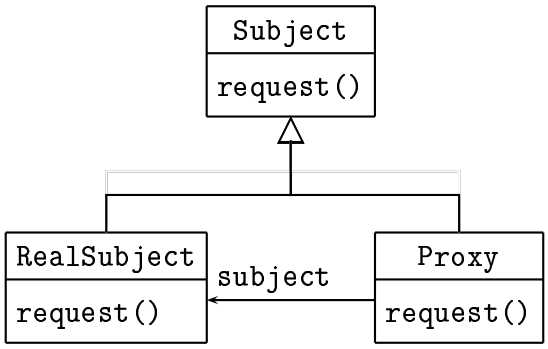

Fig. 1: Proxy design pattern.

3 Proxy

An elegant way how to solve problems described in section 2 is application of *proxy* object.

3.1 Proxy Pattern

Behavior of proxy object can be described by well-known *proxy design pattern* [1]. It consists of two classes `Proxy` and `RealSubject` having common interface[1] `Subject`, see figure 1. The common interface enables sender of `request` message communicate with instances of `Proxy` and `RealSubject` classes in the same way. So, passing reference to `Proxy` instance, instead of reference to `Subject` instance, enables implementator of the `Proxy` to monitor and possibly modify messages issued by a sender.

3.2 Java Dynamic Proxy

Java implement proxy design pattern in its standard libraries since version JDK1.3 [2]. Java uses slight modification of traditional proxy design pattern, see figure 2. The key items of Java solution are class `java.lang.reflect.Proxy` (simply `Proxy`) and interface `java.lang.reflect.InvocationHandler` (simply `InvocationHandler`).

3.2.1 Dynamic Proxy Class

A *dynamic proxy class* is a class that extends class `Proxy` and implements a list of interfaces specified *at runtime* when the class is created. Invocation through one of the interfaces on an instance of `DynamicProxy` class is encoded and dispatched to another object through a uniform interface given by `InvocationHandler`.

It means that each instance of `DynamicProxy` is associated with an *invocation handler*—an instance implementing interface `InvocationHandler`. Invocations on an

[1] Note, that `Subject` do not have to be necessarily implemented in form of an interface in sense of programming language.

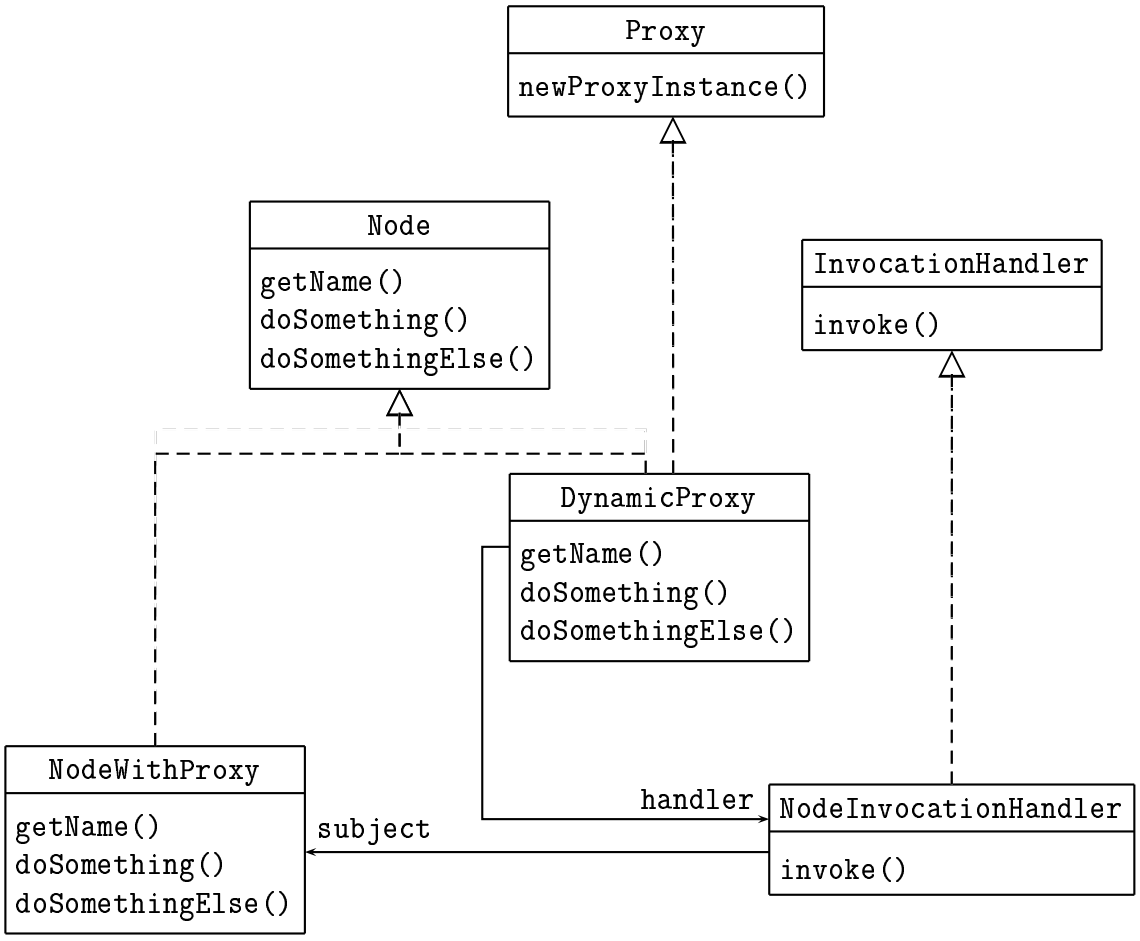

Fig. 2: Java Dynamic proxy implementation.

instance of dynamic proxy class are dispatched to `invoke` method of the invocation handler. In order to enable dispatching of invocation of arbitrary method to the single method, the `invoke` method is declared as follows:

```
public Object invoke(Object proxy, Method method, Object[] args);
```

When `invoke` method of an invocation handler is invoked, the handler can detect what `method` of what `proxy` has been invoked. Parameters passed to the `proxy`'s method are represented in form of array of objects `args`.

3.3 Handling Context via Proxy

The fact that each invocation on an instance of dynamic proxy class is dispatched to `invoke` method of corresponding invocation handler is used to handle content of context. Because the same code ensuring context manipulation appears in each method, it can be moved from the methods and centralized in `invoke` method.

For instance invocation handler `NodeInvocationHandler` of a `Node` can be defined as depicted in listing 6. The `invoke` method ensures correct context handling.

When a method declared in interface `Node` is invoked on `DynamicProxy` instance, the call is dispatched to `invoke` method of corresponding `NodeInvocationHandler` instance. Behavior of the `invoke` method can be described by the following steps:

1. retrieve context from current thread of execution;
2. if method should be invoked on different than actual node, initialize `path` to actual path stored in context;
3. if `path` has non-null value, set context's path to actual path appended with separator and name of `node`;
4. invoke the `method` on given `node`;
5. finally restore context's path to original value.

Listing 6: Definition of class `NodeInvocationHandler`.

```
public class NodeInvocationHandler implements InvocationHandler {
   Node node;
   public NodeInvocationHandler(Node node) {
      this.node = node;
   }
   public Object invoke(Object proxy, Method method, Object[] args)
         throws Throwable {
      ContextThread thread = (ContextThread)Thread.currentThread();
      Context context = thread.getContext();
      String path = (context.getNode() != node) ?
         context.getPath() : null;
      try {
         if (path != null)
            context.setPath(path + "/" + node.getName());
         return method.invoke(node, args);
      }
      finally {
         if (path != null)
            context.setPath(path);
      }
   }
}
```

■

Instances of dynamic proxies and corresponding invocation handlers are created during execution of method `getSuccessor()` of class `NodeWithProxy`, see listing 7. First, `NodeInvocationHandler` instance is created and initialized with a node which proxy will be created. Next, proxy object itself is instantiated. It is done by `newProxyInstance` method of class `Proxy`. The method works as a factory producing instances of a class[2] implementing set of interfaces given by the second argument. The newly created instance is associated with `handler`.

[2]Dynamically generated classes are typically named `$Proxy0`, `$Proxy1`, ...

Listing 7: The `getSuccessor` method of class `NodeWithProxy`.

```
public Node getSuccessor(int index) {
   Node node = successors.get(index);
   InvocationHandler handler = new NodeInvocationHandler(node);
   ClassLoader loader = Node.class.getClassLoader();
   Node proxy = (Node)Proxy.newProxyInstance(loader,
      new Class[] {Node.class}, handler);

   return proxy;
}
```

■

Using proxy object and appropriately designed invocation handler programmer do not need to worry about handling of context. The context is handled correctly on "background".

4 Performance Issues

Comfort bearing on automated handling of context information is not freebee and is compensated by:

- supplementary time which is required to invoke a method on proxy object and distribute the invocation to context handler,
- bigger consumption of memory caused by allocation of proxy objects and invocation handlers.

According to performed experiments, handling of context information via proxy object takes 2.6 times more time in comparison with traditional approaches discussed in sections 2.1 and 2.2. The growth of time is given mainly by transformation of call of a proxy method into form of a `Method` object, which is acceptable by `invoke` method of invocation handler, see section 3.2.

Another factor bearing on performance is memory consumption. It is evident that application of proxies enlarges consumed memory. Fortunately, size of instances of proxies and invocation handlers are typically several decades smaller than size of referred objects. In addition, life time of proxies and invocation handlers is rather very short and does not affect memory requirement significantly. According to experimental measures, one can estimate amount of of allocated memory per invocation of proxy method in range 100 – 120 bytes. This amount is really an estimation, because real amount of memory is hard to measure due work of garbage collector.

5 Conclusion

Dynamic proxy class associated with invocation handler gives programmer opportunities to implement new functionality without modification of existing API. Invocation

handler enables to monitor invocation of all methods invoked on corresponding proxy. Of course, there are some overheads connected with this approach, but it is programmer's (or developer's) responsibility to decide if the overheads are acceptable.

Rereferences

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, 1994, Addison-Wesley, ISBN 0-201-63361-2

[2] Sun Microsystems, Inc., *Dynamic Proxy Classes*, 1999, http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

ACB Compression Method and Query Preprocessing in Text Retrieval Systems

Tomáš Skopal

VŠB-Technical University of Ostrava, Computer Science Dept.

e-mail: Tomas.Skopal@vsb.cz

Abstract. This article presents ACB – high efficient text compression method, its word-based modification and finally discuss relating benefits by creating auxiliary query subsystem for usage in text retrieval systems.

Key words: ACB, compression, context item, index, query preprocessing

1 Compression and Indexing in TRS

Data compression is widely used in text (or full-text) databases to save storage space and network bandwidth. In typical full-text database, various auxiliary structures are provided in addition to the main compressed text. They include at least a text index, a lexicon, and disk mappings [BELL93].

Many compression methods can be used to reduce size of all these data. Relatively more advanced methods are based on coding words as basic unit. We could cite word-based adaptive Huffman coding and HuffWord [HC92, WIT94], another experiments have been done with word-based LZW by Horspool and Cormack [HC92, SAL98].

However, word-based compression methods may serve moreover for other purposes, in addition to data compression itself. One of these purposes have been presented by Dvorský, Pokorný and Snášel in [ADBIS99]. Their modification of LZW (WLZW) simultaneously provides an indexing of source text (documents). In addition to the compressed data stream, a token index file is produced. This file maintains for every token an inverted entry, which contains a list of all documents where the token occurs.

Thus, index file is particular example of interconnection between data compression and TRS. Another approach of such interconnection is introduced in the following section.

2 Query preprocessing

In TR environment we often don't know how to query for intended data. User knows only few (and/or very general) terms, which can be found in very many documents.

On the other side, user may express the conceptual content of the required information with query terms which do not match the terms appearing in the relevant documents.

This vocabulary problem, discussed, for example, by Furnas et al. [FUR87], is more severe when the user queries are short or when the database to be searched is large, as in the case of Web-based retrieval.

A survey of this topic is thoroughly presented in [ACM01] where several query-advancing techniques are discussed, leading to relevant documents response.

Back to the data compression. The word-based modification of ACB compression method is suitable for production (besides compressed stream and text index) of structure (i.e. context index) which can be used as a standalone TR tool for query advancing. This structure allows us to query upon data we don't exactly know, using a simple term-matching query. As a response of that query, we will obtain a set of word-based substrings contained within indexed/compressed documents, in which the original query term appears. These substrings we call **data contexts**. Furthermore, in the data context we can browse for any other terms lexicographically/semantically connected with the original term. Such sequences of iterative querying-obtaining will produce more terms, even term strings, which can be issued to the standard full-text searching systems (through vector/boolean queries).

Thus, idea of data contexts can be utilized for advanced query construction. Following section will introduce the fundamental structures of ACB method, respectively structures for building context-based TR subsystem. Detailed description of plain ACB method can be found in [SAL98].

3 Basic structures

In following definitions we will use general strings – sequences of terms (characters or tokens) – and the common string operations. Representation of terms can vary from bits through ASCII characters to general tokens.

3.1 Context, content, context item

Every term (character/token) t_i in the given text can be represented as a position in the text. Let any substring α in the left vicinity from this position is called **context** and any substring β in the right vicinity (including t_i) $\beta = t_i.\beta_{ext}$ is called **content** of term t_i. Let the pair $I_i = (\alpha, \beta)$ be **context item** of term t_i. Let γ_j be the substring (term string) of the text, $\beta = \gamma_j.\beta_{ext}$, then pair $J_j = (\alpha, \beta)$ is context item of string γ_j. If context or content of context item is limited by a number of terms, then the context item is bound.

Example 1.

a) Terms are ASCII characters
text: `swiss␣miss␣is␣missing`
4-bound context item for the 4^{th} 's' is '`s␣mi`**`s`**`s␣i`'

b) Terms are bit characters
text: `10011101011`
unbound context item for the 3^{rd} 0 is '`100111`**`0`**`1011`'

c) Terms are words (identifiers)
text: `'swiss''miss''is''missing'`
2-bound context item for the 'is' is '`'swiss''miss'`**`'is'`**`'missing'`'

Any subset of all possible k-bound context items of the text form k-bound **context item collection**.

3.2 Context dictionary

Structure of **context dictionary** adds a complete order to the context item collection. The order can be described by following:

1. Context item I is smaller then context item J if α_I is smaller using **reverse comparison** then α_J. Reverse comparison means right-to-left. Comparison of individual terms is interpretation-dependent.
   Formally holds: $\alpha_I < \alpha_J \rightarrow I < J$

2. If both contexts are equal, contents are examined using **regular comparison** (left-to-right).
   Formally holds: $(\alpha_I = \alpha_J) \& (\beta_I < \beta_J) \rightarrow I < J$

3. If both contexts and both contents are equal then context items are equal.

Duplicate items are not allowed.

```
1 ...swiss␣|m.....
2 ......swi|ss␣m..
3 ........s|wiss␣m
4 .....swis|s␣m...
5 ....swiss|␣m....
6 .......sw|iss␣m.
```

Fig. 1: Context dictionary, terms are ASCII chars (vertical separates context and content)

4 Compression method ACB

This compression method[1] uses idem structures for highly efficient text compression. Compression algorithm passes the text term-by-term obtaining context items by usual fashion. At the beginning of encoding/decoding, the runtime[2] context dictionary is empty.

4.1 Encoding algorithm

1. Current context item C (made up from text already encoded (context) and the rest of the text (content)) is examined against the actual state of context dictionary. Position i of the nearest context item N (most similar to C) in the dictionary is found using the defined order, but comparing only the contexts. Formally holds: $(\alpha_C \leq \alpha_N) \& \forall K (\alpha_K < \alpha_N \rightarrow \alpha_K < \alpha_C)$

2. From this position in dictionary we incrementally search (in both directions – the search radius is naturally limited) a context item M at a position k, where β_M best match β_C. Best match is determined by n – maximal count of equal terms.

3. To the output goes a triplet $(k-i,n,t)$ – where t is the first non-matching term in β_C (from step 2)

4. The context dictionary is updated with $n+1$ new context items. Each new added context item is the previous one incremented by 1 (i.e. the context is longer by 1 and the content is shorter by 1).

Example 2. (encoding step) :

Text being encoded is string from example 1a. Dictionary's actual state is shown at the figure 1. Current context item is `'swiss␣m|iss␣is␣missing'`.

1. Appropriate item in context dictionary is found. Item at position 2.

2. Then search in the dictionary from this position for item with best content match. Item at position 6. 4 chars match (`'iss␣'`), first non-matching char in the current item content is `'i'`.

3. To the output goes triplet (4,4,`'i'`).

4. Context dictionary is updated with 5 new items (see figure 2).

[1] Associative Coder of Buyanovsky by George Buyanovsky
[2] means in-memory; items are positions of terms and are unbound

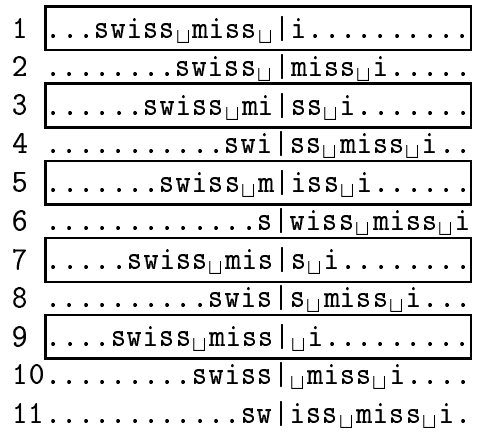

Fig. 2: Context dictionary after update (items in frame are the new ones)

4.2 Decoding algorithm

The decoding is exactly inverse. Runtime context dictionary grows the same way as by encoding. Text is reconstructed applying triplets on the actual state of dictionary.

Example 3. (decoding step) :

Initial current item and state of dictionary are the same as in previous example. Input triplet (4,4,'i') is processed by following:

1. Find in dictionary the appropriate item for current item. Same way as by encoding. Item 2 was found.
2. Take the first member of triplet (here 4) and add it to the found position. Get item on position 6. Read from the items content first n chars specified by the second triplet member. You get 'iss␣'.
3. To the output goes 'iss␣' + the third member 'i' → 'iss␣i'
4. Dictionary is updated with 5 new items (see figure 2)

4.3 Benefits of ACB brought into TRS

- ACB uses for each file (data stream) new dictionary, which serves as a compression tool. This data-locality allows represent context by numbers (positions to source text).
- For purposes of TRS we can persist produced dictionary as a valuable source of semantic relations. For a growing collection of documents there could evolve big dictionary over time. In this phase we should speak rather about **context index** than about persistent dictionary.

- ACB algorithm itself can provide (as a side effect) heuristic techniques to improve context index creation and analysis.

5 Building context-based TR subsystem

General context index conception must now become more clear. Because of context index is persistent and standalone structure, context items cannot be represented as some positions in source text. This feature considerably differs from runtime context dictionary which can be unbound. Context index items are represented as term identifier strings. Identifiers and terms are associated in text index which serves as a lexicon as well.

Thus, context index must be bound and length of context items significantly determines storage costs. Fortunately, lengths up to 10 terms seem to be sufficient.

5.1 Simple query

Example 4. (one item in 4-bound context index) :

```
                    :
                    :
steam engines | work reliably
                    :
                    :
```

When querying on 'engines', string "'steam' 'engines' 'work' 'reliably'" will return as a response. This answer offers possibility that the word 'engines' relates *somehow* to the other words. The user have obtained more information and is able to formulate better query.

5.2 Context item reduction

It is obvious that even 10-bound context index might produce big storage overhead. Linguistically, only few words are possible to be semantically interconnected with words in distance greater than 3. However, this is a question for additional testing in real TR environment.

5.3 Alphabet of terms

Next goal in the process of context index reduction is the alphabet reduction. This can be achieved by following restrictions:

- Word and non-word consideration as presented in [ADBIS99]. Words are semantic terms (words) and non-words are blank terms (spaces, separators, tabs,

etc). Words and non-words strictly alternate (word is immediately followed by non-word).

Context index contains only words (i.e. their identifiers). This feature considerably improves response relevancy.

- Reduction achieved by *lemmatizers*[3]. Lemma is the lexical root of its various derivates (for example 'go' is lemma for 'goes','went','going',etc). Instead of that derivates, the lemma will be used in context index. However, this loss of information may worsen the response relevancy.

- Ignoring stop-words consideration. Stop-word is a word with minimal semantic meaning. For example 'the', 'a', 'it', 'that'.

Reduction of alphabet causes reduction of context index as well. There will be high probability (even higher with low-bound items), that required item addition will produce duplicate items in the index – that is not allowed – thus the item will be ignored.

6 Future work

- In the future we are going to examine **transitive queries**, i.e. complex queries combining particular context-content match.

  Example 5.

```
steam engine | is an ancestor of jet propulsion
                                 jet propulsion | uses liquid fuel
```

  Request on 'steam engine' might produce 'jet propulsion' or even 'liquid fuel'.

- Nowadays, all the presented stuff is under software development and extensive testing.

Rereferences

[ADBIS99] J. Dvorský, J. Pokorný, V. Snášel: *Word-based Compression Methods and Indexing for Text Retrieval Systems*, Proc. ADBIS'99, Springer Verlag, 1999, pp. 75-84

[ACM01] C. Carpineto, R. de Mori, G. Romano, B. Bigi: *An Information-Theoretic Approach to Automatic Query Expansion*, ACM Transactions and Information Systems, Vol. 19, No. 1, January 2001

[FUR87] G.W. Furnas, T.K. Landauer, L.M. Gomez, S.T. Dumais: *The vocabulary problem in human-system communication.* Commun. ACM 30, 11 (Nov.) 1987, 964971.

[3] stemming techniques

[BELL93] T.C. Bell et al: *Data Compression in Full-Text Retrieval Systems*, Journal of the American Society for Information Science. 44(9), 1993, pp.508-531

[HC92] R.N. Horspool, G.V. Cormack: *Construction Word-based Text Compression Algorithms*, Proc. 2nd IEEE Data Compression Conference, Snowbird, 1992

[SAL98] D. Salomon: *Data Compression*, Springer Verlag, 1998

[WIT94] I.H. Witten, A. Moffat, T.C. Bell: *Managing Gigabytes: Compressing and Indexing Documents and Images.*, Van Nostrand Reinhold, 1994

Methods of employing metadata for managing large systems

Jan Vrana

CVUT Faculty of Electrical Engineering, Dept. of Computer Science

e-mail: vrana@komix.cz

Abstract. Metadata is a term, which is mentioned very often. Various theoretical aspects of metadata utilization are discussed most frequently. In this paper we shall focus on usage of metadata for visualizing and querying very large and complex system of objects. We will also describe a universal query processing system as a module of an application server. It can be used for querying of almost arbitrary data that can be described in terms of Entity and Relationship which gives the Universal query processing system opportunity to be used in a wide range of application domains.

Key words: Storing Data, Metadata, Query Processing

1 Topic

Consider a system consisting of a large number of objects (say) which may be organized into higher-order units. Every object may have a large number of attributes. Attribute values may determine relationships of the given object to other objects. There may be a number of types of relationships and each object may have different number of relationships to other objects. Further consider, that values of individual attributes may change in time and that it is important to store all historical values and their changes.

System with considered properties can be illustrated. A distributed control system or a complex technology network or other complex network consisting of individual elements (each of them having its own internal structure and own large sets of configuration parameters) can be a good illustration of the considered system. Values of individual attributes of elements determine a behavior of a whole system. The goal is to manage (to assign values of individual attributes) in such a way, that the whole system has required behaviour. It might be a very complicated task due to a large number of elements in the system and their high interconnectivity.

In order to be able to meet this goal (to tune-up or reconfigure such a system), the staff needs to be able to view and change current configuration of individual elements, their attributes and relationships and to check the overall setup consistency.

Mentioned tasks are very difficult and time consuming issue when performed without any computer supported intelligent visualisation, automation and verification. It is difficult even to create the computer support due to the system complexity. This paper will discus, how metadata can help with a formal description of such type of systems and how to build a controlling application based on the formal description. We will also show, that a large set o application domains comply to our assumptions, which means that main concepts and principles described in this paper can be utilized there.

2 Example application

We shall use one particular application solving one particular problem of assumed type to illustrate solution of the general problem. We shall mention only those parts of application that are dedicated to storing and managing the processed data. Figure 1 is a schematic chart of the application architecture which helps to identify the application modules.

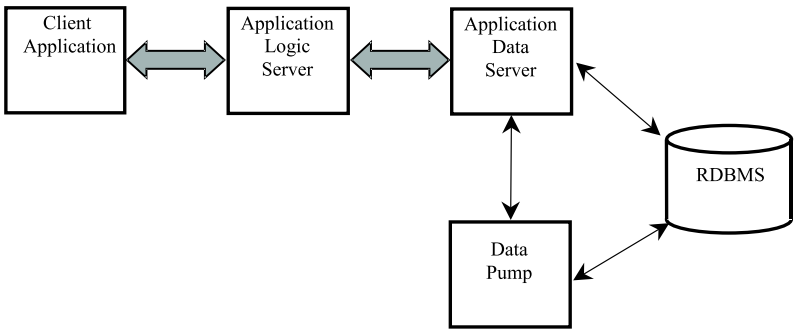

Fig. 1: Example Application Architecture

We shall discuss the structure and contents of data stored in a database and an architecture of the Application Data Server. It is based on the principles of how it stores and controls data that it provides to overlaying application layers, taking the role of an object data server. We will not discuss the Data Pump, although it also manipulates and controls the data, because the Data Pump is based on the same principles as the Application Data Server.

3 The structure of data

At the beginning, we mentioned that metadata may be used to solve this problem. What are the symptoms and reasons that lead to metadata employment? The most important reason in the case of the described problem was its size and complexity and implied diversity that must be implemented in algorithms. The diversity is so high that it is impossible to implement every single alternative by a "fixed" algorithm. Although fixed algorithms can achieve the lowest computational complexity (= highest computation effectivity). The development and maintainance of software

with described characteristics is not only complicated but also expensive. With the help and utilization of metadata it is possible to build the software in such a way, that it will be universal and it's behaviour will be controlled by the metadata, which will make the software independent on the particular structure of the processed data. The functional diversity will be achieved by diversity of metadata instead of a diversity of (men written) algorithms.

It is common that every universality is paid for. In this case it is paid mainly by a higher cumputational complexity (lower effectivity) of universal algorithms and by requirements for storing a large volume of metadata. The computing and the storage requirements may be important according to assumed system complexity and it must be taken into account when designing the structure of metadata (the way of storing the system description). So, let us consider also a compromise method. Let us call it the Dynamic Relational Data Storing Method. It makes a compromise between the Fixed Structure Data Storing Method and "fixed" algorithms on one hand and using a general graph data structure to store the data (here called the General Data Storing Method) on the other hand. The main feature of the Dynamic Relational Data Storing Method is a separate storing of metadata from controlled data, which are managed with a maximum utilization of relational characteristics of a host relational DBMS. A more detailed explanation of the terms Dynamic Relational Data Storing Method and General Data Storing Method, description of their main features and their comparison follows.

3.1 Data storing method

We do not consider that the Fixed Structure Data Storing Method is a good solution to our problem due to mentioned system complexity, so we will not discuss it further.

One reason to use the Dynamic Relational Data Storing Method may be the fact that a size and a structure of a maintained system model is not known (eventually, if the system model is assumed to be further developed and extended). Let us consider the Dynamic Relational Data Storing Method, that stores its metadata separately in fixed-structure database tables, and then it creates a dynamic structure of tables for storing data. To clarify this procedure, we will breefly describe the General Data Storing Method and its main features. We will use E-R diagrams to describe a database structure. Diagrams will be in such a form, that every single entity will correspond to a single database table. Arrows in the middle of relationship lines indicate a direction of foreign keys propagation.

To compare the two data storing methods mentioned abowe, let us consider one representative example of a system model, that consists of interconnected units, which have their attributes and the attributes may carry different values. Let us describe how this model can be stored with a given data storing method and what features and requirements will the stored model have. For the example, assume the system model consisting of 100 higher-order units. Each higher-order unit consists in average of 100 lower-order units. Every lower-order unit has a character of a relational parameter table and has 10 relationships to other lower units and it has 1000 attributes

(individual cells of parameter table). Let us further assume that every attribute may have 3 historical values.

3.1.1 General Data Storing Method

Figure 2 shows the main idea of what we call the General Data Storing Method.

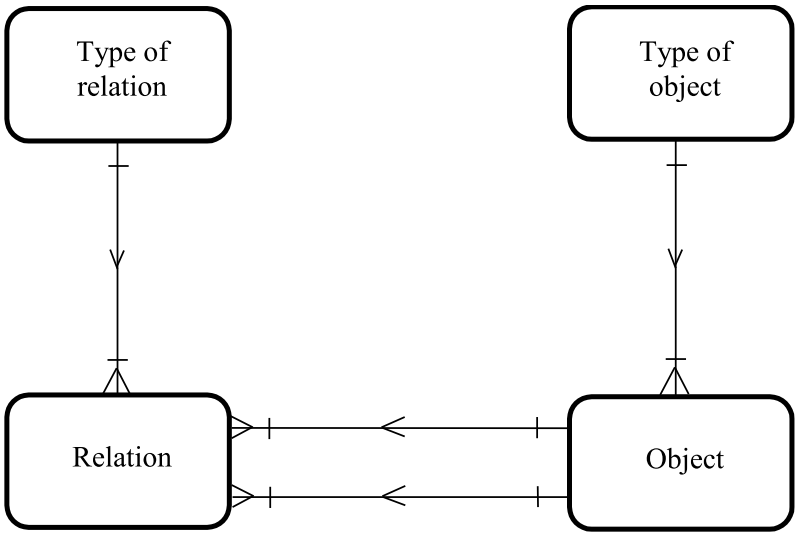

Fig. 2: The General Data Storing Method

The two related entities at the bottom of the diagram are capable to store an arbitrary graph-type data structure. Let us look how to describe some fundamental constructions, which are needed for building of a system model when using this method:

- **A higher order unit consists of lower order units:** here one object A of type "unit" is *connected by a relationship* of type "consists of" with other object B of type "unit" (two records with different identifiers exist in the database table Object with the value of foreign key from the table Type of objects corresponding to the term "unit". We shall further denote it as the "type of object" and, by analogy, as the "type of relationship". There exists a record in the table Record, which has: identifiers of both mentioned records from the Object table as values of respective foreign keys and the type of relationship corresponding to the term "consists of").
- **One unit has a (logical) relationship to other unit:** one object of type "unit" is connected by a relationship of type "is incident" with an object of type "relationship", which is further connected by another relationship of type "is incident" with other object of type "unit".
- **A unit has attributes:** an object of type "unit" is connected by a relationship of type "unit attribute" with other object of type "attribute".
- **Attribute has values:** an object of type "unit" is connected by a relationship of type "attribute value" with other object of type "value".

Any other aspect of the model can be described in a similar way. It is no doubt that the advantage of this method is its universality. The universality ensures that almost any change of the model can be covered by a change of metadata without a necessity to change programs.

What are the costs of this method? What should we pay for the universality? What are its disadvantages?

We will avoid all unnecessary details. All values may be derived from described system model characteristics and from the basic constructions. The following table shows the number of records that have to be inserted into respective tables to store the system model when using this method:

table Object		**table Record**	
Type of Record	**records**	**Type of Record**	**records**
Unit	10^4	**Consists of**	10^4
Relation	$9 \cdot 10^4$	**Is incident**	$2 \cdot 10^5$
Attribute	10^7	**Unit Attribute**	10^7
Value	$3 \cdot 10^7$	**Attribute Value**	$3 \cdot 10^7$
	$4 \cdot 10^7$		$4 \cdot 10^7$

Most of records are required for storing description of attributes, their values and for assigning values to attributes.

It requires a lot of system resources just to store this number of records. Querying this data structure (processing queries) is also very difficult and complex task. Processing of one query retrieving attributes implies either multiple joining of large tables or gradual partial retrieving of the result data. In both cases it leads to unacceptable response times. This was only a case of a query retrieving a value of an individual attribute. The response times would be even much longer when processing a query that tries to retrieve a whole parameter table consisting of thousands of attributes or tries to find some hidden relationships between objects or in a case of massive data-insert operations.

Another type of complication that this method implies comes from the fact that metadata is stored in the same place with data. It has at least two unpleasant consequences. The first consequence manifests mainly at runtime. It is the metadata access complexity. When accessing data, many subqueries retrieving metadata must be processed to obtain the requested data. Although metadata occupy only relatively small part of data space and although it is a "static" part, it must be "mined" from very large tables. It is obvious that such mining implies a high time demands. The application development is influenced by the second consequence of storing metadata together with data. Namely metadata development. Metadata can be understood as a program for a universal automat that handles the data according to "instructions" given by metadata. It is well known that every computer program has to undertake some development process in which it is repeatedly tested, debugged and upgraded (an "old version" is repeatedly replaced by a "new version"). The same holds for metadata with a difference that the real (and possibly valuable) data should be preserved during the upgrade process. Due to a very tight cohesion of metadata and data the upgrade process is usually very complicated.

3.1.2 Dynamic Relational Data Storing Method

The Dynamic Relational Data Storing Method is based on such an assumption that individual units of modelled system have character of relational attribute tables. Then it can be stored very efficiently taking advantage of a native relational environment of a host (relational) database engine. Every modelled system unit data (or all units of the same type) can be stored in one dynamically created database table, separately from the description metadata. Figure 3 describes the main principles:

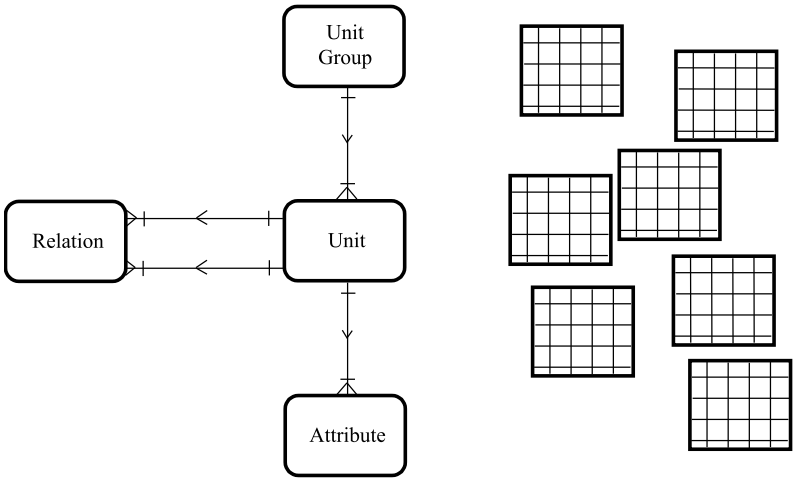

Fig. 3: The Dynamic Relational Data Storing Method

The figure clarifies separation of metadata (description of system model structure - of units, their attributes, relationships and unit groups) on the left side and data (attribute values in a dynamically created tables) on the right side. The construct Unit-Relationship is similar to the construct Object-Relationship in the General Data Storing Method. In this method, the assumed relational characteristics of stored data are utilized so that there are special tables dedicated for some of the fundamental modelling constructs. The same fundamental modelling constructs as in the case of the General Data Storing Method can be expressed in the following way:

- **A higher order unit consists of lower order units:** units (records in the Unit table) correspond to the term "lower order unit" and the value of a foreign key is an identifier of a record from the Unit Group table corresponding to the term "higher order unit".
- **One unit has a (logical) relatioinship to other unit:** one unit is connected by some relationship with another unit.1
- **A unit has attributes:** the value of a foreign key of attributes (records of the Attribute table) is an identifier of a unit (a record in the Unit table). Only "columns" - attributes of one row - are stored according to assumed columnal structure of units.
- **Attribute has values:** values of the corresponding column in the dynamically created table corresponding to the given unit.

The Dynamic Relational Data Storing Method avoids the main weaknesses of the General Data Storing Method being still general enough with respect to structures of assumed system models. The following table presents the costs of the Dynamic Relational Data Storing Method to store metadata, so that it can be compared to the "costs" of the General Data Storing Method.

Table	**records**
Structure	10^2
Unit	10^4
Relation	$9 \cdot 10^4$
Attribute	$5 \cdot 10^4$
	$2 \cdot 10^5$

Still speaking about the example, dynamically created data tables will have in average 50 columns and will share $6 \cdot 10^5$ records in total. The actual number of data tables created and their sizes depend on their logical structure. One extreme is to create $1 \cdot 10^4$ data tables. Such a number is unacceptable but this extreme case can be easily transformed in such a way that the number of tables considerably decreases. This transformation will imply a growth of individual table sizes but it is not critical since the estimated total number of rows is $6 \cdot 10^5$.

The General Data Storing Method suffers from two main weaknesses: firstly from high redundancy and large volume of stored data implying high computational complexity and secondly from high metadata-data cohesion implying difficult metadata manipulation and upgrade.

The first mentioned weakness of the General Data Storing Method is high redundancy and large volume of stored data. The Dynamic Relational Data Storing Method will produce much smaller metadata tables resulting in adequate decrease of the metadata querying overhead. The most essential complexity reduction is in accessing data. Dynamically created data tables (suppose 1000) will have in average (according to previous assumptions) 50 columns and 3000 rows. Due to this data storing architecture it is possible to retrieve data from different related units using single SQL query that joins adequate data tables. Joining data tables of mentioned sizes is not critical. The total processing time of an elementary or a complex query may be much shorter than in the case of the General Data Storing Method. The same holds for massive inserting operations during data importing.

The second mentioned weakness of the General Data Storing Method is a difficult metadata manipulation and upgrade. This task is also much easier in case of the Dynamic Relational Data Storing Method although it is a general weakness of all applications that are based on using metadata.

4 Application Data Server

The Application Data Server plays a role of an object data server. It provides data and communication to overlaying application components on a structural (logical)

level keeping these components independent on the actual data storing method. We will briefly describe the architecture of the Application Data Server mentioning the Query Processing System more in detail. The Query Processing System is one of the Application Data Server modules and it has a high availability potential.

By the term of "Application Server" we do not mean just a framework that needs to be still filled by required functionality as it is very often supposed when speaking about middleware "Application Servers" like, for example, the IBM WebSphere, BEA, iPlanet, etc. By the "Application Server" we mean a fully featured application that communicates with its neighbourhood through a network communication protocol or other inter-process communication method.

In the example application, the overlaying components (the Application Logic Server) communicate with the Application Data Server with requests like: "Get structure of unit XY", "Get data from unit XY satisfying condition ZZ", "Change the value of attribute AA in unit XY" or "Reread data of unit XY", etc., The Application Logic Server does not care about how these data are actually stored and how to retrieve them.

The architecture of the Application Data Server is modular and multilayer, as shown in the following picture. This architecture highlightes the Query Processing System and also the part which is usually understood as a middleware "Application Server" (top and left modules) (see fig. 4).

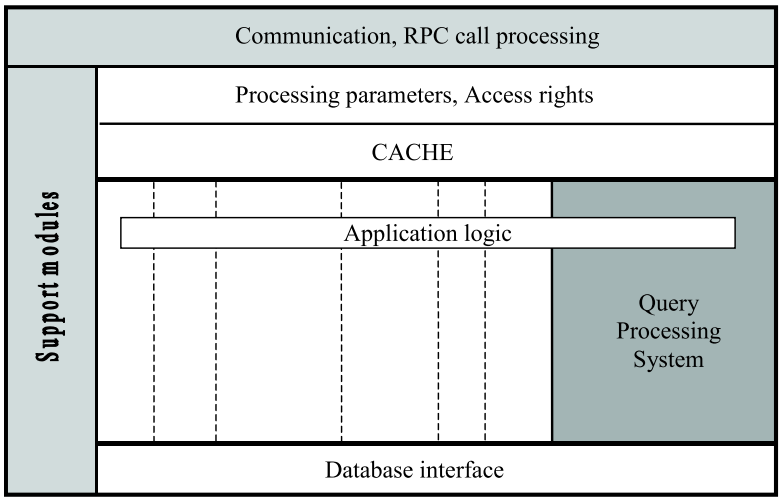

Fig. 4: The Application Data Server

This architecture is based on our experience with building large datawarehouses. Let us just mention the lowermost layer called the Database Interface. This layer helps the whole application to be more easily portable to various RDBMS platforms and it should be as thin as possible. Now, let us describe the other highlighted module, the Query Processing System.

4.1 Query Processing System

Currently there is a frequent need to interactively process and query data gathered by primary information systems. To find and process complex logical relationships in this data. The example application, which is focused on processing configuration parameters of a complex technology network, can serve here as an example of an application, which is based on such a need. To bring another example, let us imagine a bank, an insurance company or a business company and an application giving it's managers, salesmen, dealers, etc. an chance to interactively query and view the core data of the company and to explore some complex relationships implied by this data.

Such needs are usually satisfied by specialized tailor-made applications, which are completely or partially dependent on the task and application domain. These specialized systems are usually based on several more-or-less fixed, in advance constructed queries with some modifiable parts, typically filters and a projection.

The application domains of individual applications are usually wery differend from each other but there is one common aspect . All these applications process and query data interconnected by relationships - a data from a "world" that may be described by objects or entities of various types and relationships of various types interconnecting objects. If we succeed to find some general formal description of these "worlds" we may be able to build a more universal query processing system based on this description and if the query processing system takes the advantage of universality of the formal description it may be capable of processing even AD-HOC queries without any touching implementing software.

We mentioned here two of methods for description of ER (entity-relationship) based "world". Let us focus on the Dynamic Relational Data Storing Method. This method was built in such a manner that it first creates a description of a "data world" = metadata and then, based on it, it created the "data world" itself = data. Due to a separation of metadata from data, this method may be also used in the reverse manner. It may preserve and accept the existing "data world" and just formally describe it by separated metadata. The resulting configurations and their features will be very similar.

Suppose that the Query Processing System processes data that is stored using the Dynamic Relational Data Storing Method. In this case we can take advantage of a rich metadata desctription of data and reach a large flexibility and generality. The flexibility may come so far that the Query Processing System is capable of processing arbitrary AD-HOC queries, graphically defined by user. Due to its philosofhy, the Query Processing System may be also used in an application having much weaker metadata support or no metadata support at all. Figure 5 shows an internal architecture of the Query Processing System and it's link to a host application.

The above picture shows that the Query Processing System is composed from two parts: 1) the general part, which is application domain independent and generally usable and 2) from the application dependent part that is completely dependent on the application domain and on the data storing method. The Application dependent part of the Query Processing System has to be revised in every implementation. Two arrows on the above picture correspond to individual steps of a two-step processing of a query.

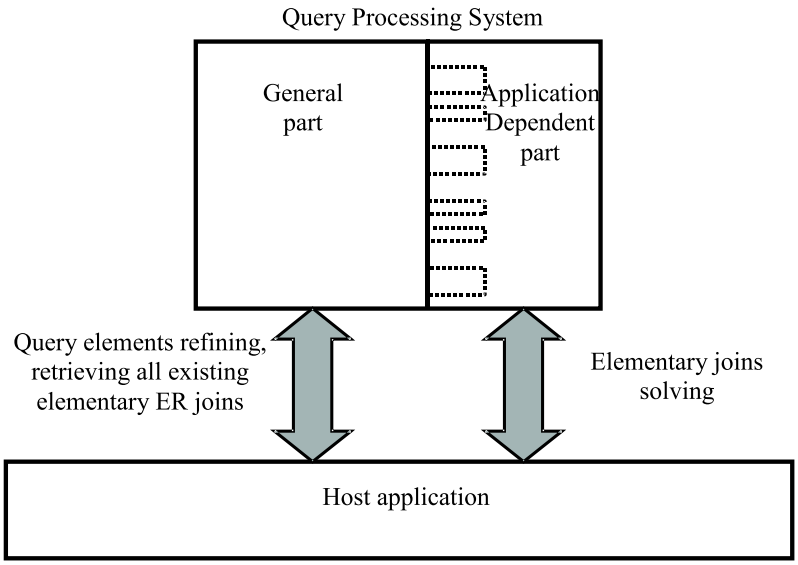

Fig. 5: The Query Processing System

A common feature of both parts of the Query Processing System is the fact, that constructive elements of processed queries may be described in terms of entity and of relationship, which are also constructive elements of the described system models, according to the initial assumptions. All queries to both: the general and the application dependent parts are in fact strings of basic elements of type **E** (entity) and **R** (relationship). The first and the last elements of every string have to be elements of type E. Strings, which begin or end with an R-element are unacceptable. Furthermore, every two neighbouring E-elements in any string have to be interlaced by an R-element and every R-element have to be surrounded by two E-elements. The Query Strings have the following form:

$$E_1 - R_1 - E_2 - R_2 ... R_x - E_y$$

4.1.1 General Part

The General Part of the Query Processing System deals with a *structure* of processed data. It's task is to search for all potentially existing relationships between existing objects (entities) in a given data structure, which are solutions of the given query. A query to a general part has a tree structure. It describes all requested object relationships. Every single path from the root to a leaf in such tree presents an above mentioned string of elements. Figure 6 shows an example of an "ER-world":

Symbols on the above picture represent objects. Shapes of symbols correspond to types (classes) of involved objects. For simplicity, let us denote the object types by letters A to F. Objects of the same type are distinguished by an "instance" number. Relationships have similar features. There are relationships of two types on the above picture. Relationship types are again denoted by letters, here X and Y. Relationships of a same type are again distinguished by a number.

Suppose the following query: find all routes starting in the object of a type B, following then a relationship of any type to an object of type C or D, coming along

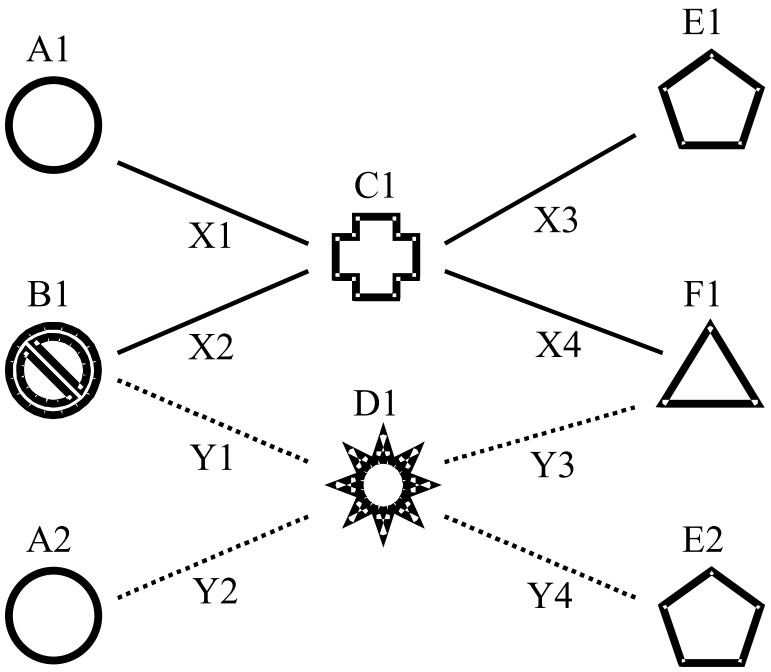

Fig. 6: "World" of objects

relationship of any type to an object of type E. This query to the general part of the Query Processing System is represented by the following string (brackets represent elements of requested type, a letter prescribes a requested type, a question mark "?" substitutes any type and a pipe symbol "|" means enumeration of requested types): $[B][?][C|D][?][E]$.

Two routes are solutions of the given query: route $[B1][X2][C1][X3][E1]$ and route $[B1][Y1][D1][Y4][E2]$. These two routes are the result of the general part of the Query Processing System.

With this mechanism it is possible to build an arbitrarily complex specification of required routes. The specification in the above example has only a trivial linear structure. Generally, the specification may have a form of a tree or of a forest.

The General part of the Query Processing System is independent on an application domain and on a method used to store data. It even does not define particular representation of entities and relationships. In order to be able to manipulate particular data, it uses several precisely defined elementary services from the application dependent part. Let us call these services the Connection Points. These connection points are used to adapt the general part of the Query Processing System to particular application domain. They have to be implemented by the host application. It is fully the host application's responsibility how it implements these connection points and consequently how information concerning entities and their relationships is retrieved. Whether "comfortably" with a full metadata support as in the case of the Dynamic Relational Data Storing Method, or less "comfortably" with partial metadata support or even with no metadata support at all. By redefining functionality of connection points it is possible to redefine or optimise the process of searching existing routes, because it can be a time and computation-complex task in some cases.

4.2 Application Dependent Part

An Application Dependent Part is a second layer of the Query Processing System. It's goal is to process and solve the found links of entities and relationships with possible additional filters to individual entities and relationships. It is completely an application dependent task. In case of the Dynamic Relational Data Storing Method, all necessary information (like a name of particular database table, names of columns involved in a table join, names of columns that are subject to some filter) can be obtained from metadata and used for dynamical building of a complex SQL statement. If such a full metadata support is not available, it is possible to solve given problem by selecting a pre-built SQL statement fragments and combining them together to build the resulting SQL statement or by only selecting one statement from a set of the pre-built final SQL statements. A combination of mentioned approaches may also be used for example to optimise critical queries if they have too high computational complexity when dynamically built. Such critical queries may be implemented by optimized fixed queries while others may be generated dynamically.

Described features allow the Query Processing System to be really universal and application domain and a data storing method independent so that it can be widely usable.

5 Conclusion

In this paper we discussed the problem of manipulating and querying a large and complex system of interconnected objects. We mentioned several approaches for solving this problem and we described two alternative methods for storing data of such a system in a relational DBMS: 1) the General Data Storing Method and 2) the Dynamic Relational Data Storing Method. We found the Dynamic Relational Data Storing Method to be very appropriate and convenient for description and management of complex relational data.

We described basic ideas of the Query Processing System that may become (especially when combined with the Relational Data Storing Method) a powerful tool for comfortable querying of complex relational data structures and systems. The architecture of the Query Processing System makes it an application domain and a data storing method independent. We also described, that the Dynamic Relational Data Storing Method is not limited to its original purpose - to describe and manage stored data. It may also be used partially - only for describing data managed by other system. In this case the Query Processing System may be used for comfortable querying data that is managed by a foreign system.

The target application domain of the described method is a management of large and complex networks of active elements. Together with computer networks, this category also covers other telecommunication networks and for instance also various networks transporting electrical energy, water, gas and other transporting networks. This method may also be used in those domains where there is a need for an easy changing of behavior of a distributed system. For instance an international transportation, logistic, and market chains.

Although this method is targeted to managing large and complex systems, it may also be used in much smaller scale tasks, for instance in cases where primary information systems manage big number of interrelated events as it is in banks, insurance companies, etc. Here this method may not only simplify the development of an application for a secondary management of the core data but it mainly provides extended features of the management itself and consequently, it enables to achieve expected behaviour of the managed system.

The Query Processing System may be used for querying almost any data with a relational character. This allows its wide utilisation.

Organizational Structures Modeling and Analyze

Ivo Vondrák, Václav Snášel, David Ježek

Department of Computer Science, VŠB-Technical University of Ostrava,
tř. 17. listopadu 15, 708 33 Ostrava-Poruba

e-mail: ivo.vondrak@vsb.cz, vaclav.snasel@vsb.cz, david.jezek@vsb.cz

Abstract. Business process re-engineering is based on changes of the structure of business processes with the respect to obtain their higher efficiency. As a result of this business process re-engineering new organizational structure has to be defined to reflect changes in business processes. The organizational structure is usually defined using the best experience and there is a minimum of formal approach involved. This paper shows the possibilities of the theory of concept analysis that can help to understand organizational structure based on solid mathematical foundations.

Key words: business processes, re-engineering, organizational structure, concept analysis, concept lattices.

1 Introduction

Business processes represent the core of the company behavior. There are many possibilities how these processes can be defined. Usually all modeling tools are focused on various kinds of business process aspects based on what abstraction is considered as the main. From this point of view there are three basic approaches that can be employed [Chri95] for the business process specification:

- *Functional View.* The functional view is focused on activities as well as on entities that flow into and out of these activities. This view is often expressed by Data Flow Diagrams [Mar79].
- *Behavioral View.* The behavioral view is focused on *when and/or under what conditions* activities are performed. This aspect of the process model is often based on various kinds of *State Diagrams or Interaction Diagrams.* More sophisticated approaches based on the theory of *Petri Nets* are convenient for systems that may exhibit asynchronous and concurrent activities [Pet77]. The behavioral view captures the control aspect of the process model. It means that the direction of the process is defined on current state of the system and event that occurs.

- Structural View.The structural view is focused on the static aspect of the process. It captures objects that are manipulated and used by a process as well as the relationships that exist among them. These models are often based on the *Entity-Relation Diagrams* or any of the *Object Diagrams* that are used by the various kinds of *Object Oriented Methods*.

Unfortunately, none of these views captures organization structure of roles implemented by human resources participating in processes being modeled. For example, BPM (Business Process Modeling) method [VonSK99] involves roles in a process specification but there is no option how the organizational structure implied by such models can be analyzed and evaluated.

The next chapters will show how the theory of concepts might remove the gap between process models and organizational structure.

2 A Motivating Example

Lets start with a toy example to demonstrate how the business process models serve as a source of the organizational structure specification. Let's assume that we have a car sale company with a showroom that employs four people: *manager*, *salesman*, *technician* and *accountant*. Let's assume that we have only two business processes enacted: *car sale* and *car fleer purchase*. The first one reflects the situation when a customer wants to buy a car; the second one is performed by the showroom when a fleet of cars has to be purchased for demonstration and for immediate sale purposes.

Car sale process starts with the activity of *offering* a car to a customer. Activity of *ordering* the chosen car from a manufacturer follows if the car is not available in the showroom. Employees of the showroom try to help the customer with *financing* afterwards and finally the payment from the customer is *checked*and the car is *handed over*. *Fleet purchase* process is started with the *selection* of the appropriate fleet, and then the selected cars are *ordered*, *paid* and *taken over* by the showroom.

Simple flow chars of the following structure can model these processes as Fig 1.

It is obvious that the next logical step is to assign roles responsible for the specified activities. Based on that assignment it is possible to derive so called table of responsibilities that can be used for the purposes of the organization structure specification. Let's assume that in the car sale process for *offering* activity the *salesman* or *technician* is responsible. The showroom *manager* or *salesman* can realize the *ordering* activity while the *accountant* or *manager* takes care of the financial operations like help with *financing* and *checking* the payment. Finally for the activity car *hand over* the *technician* or *salesman* is responsible. The process of fleet purchase has to be assigned with its roles, too. Resulting tables of responsibilities are Table 1 and Table 2.

It is obvious that our showroom would have to implement some additional processes with more complex structure in a real life situation but for our purposes that are to demonstrate the potential of the theory of concepts this simplified example should be sufficient.

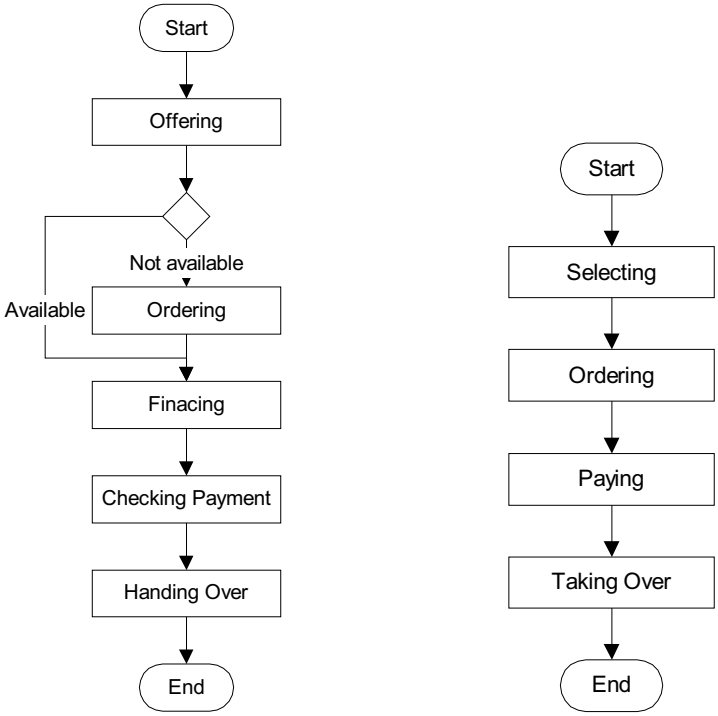

Fig. 1: Processes flow chars

	Offering	*Ordering*	*Financing*	*Checking P.*	*Handing Over*
Manager		×	×	×	
Salesman	×	×			×
Technician	×				×
Accountant			×	×	

Table 1: Role Assignment for Car Sale Process.

3 Formal Concept Analysis(FCA)

The idea of formalization of terms of context and concept by means of theory of lattices is not quite new. Formerly isolated experiments at application so-called Galois lattices already existed, especially in the area of Information retrieval, but systematically built up theory did not arose until the work of R.Wille in about 1980 at the TH Darmstadt, Germany and his group. FCA is a mathematical approach to data analysis based on the lattice theory of Garret Birkhoff [Bir93].

The claim of this contribution is not a detailed description of the whole problems but rather motivating acquaintance with the given problems.

For the formalization of term concept and context a theory of ordered sets and the theory of lattices are applied. The terms from these classic disciplines can be found in [GantW99].

	Selling	*Ordering*	*Paying*	*Taking Over*
Manager	×	×		
Salesman		×		
Technician	×			×
Accountant			×	

Table 2: Role Assignment for Fleet Purchase Process.

Definition 1. A *context* is a triple $(\mathcal{O}, \mathcal{A}, T)$, where $\mathcal{O}$ and $\mathcal{A}$ are sets and $T \subseteq \mathcal{O} \times \mathcal{A}$. The elements of $\mathcal{O}$ are called objects and the elements of $\mathcal{A}$ are attributes.

Concept analysis theory can be used for grouping of *objects* that have common *attributes*. Concept analysis begins with a binary relation, or boolean table, T between a set of objects $\mathcal{O}$ and set of attributes $\mathcal{A}$.

For any set of objects $O \subseteq \mathcal{O}$, their set of common attributes is defined as $\sigma(O) = \{a \in \mathcal{A} \mid \forall o \in O : (o, a) \in T\}$. For any set of attributes $A \in \mathcal{A}$, their set of common objects is $\tau(A) = \{o \in O \mid \forall a \in A : (o, a) \in T\}$.

Definition 2. A pair (O, A) is called a *concept* if $A = \sigma(O)$ and in the same time $O = \tau(A)$.

Definition 3. Let $(\mathcal{O}, \mathcal{A}, T)$ be a context. For couple (O, A) where $O \subseteq \mathcal{O}$, $A \subseteq \mathcal{A}$, $A = \sigma(O)$ and $O = \tau(A)$ we say O is extent and the set A is intent of concept (O, A). The set of all concepts is called $\mathcal{C}(\mathcal{O}, \mathcal{A}, T)$.

This property says that all objects of the concept carry all its attributes and that there is no other object in O carrying all attributes of the concept. When looking at the cross table this property can be seen if rectangles totally covered with crosses can be identified.

Looking at the definition of a formal concept one can easily see that for all $O \subseteq \mathcal{O}$, the pair $(\tau(\sigma(O)), \sigma(O))$, is a formal concept. The dual holds for all $A \subseteq \mathcal{A}$, i.e. $(\tau(a), \sigma(\tau(A)))$ is always a formal concept, too. Yet, the sets of concepts achieved in this way are equal and contain exactly the concepts existing in the given context.

The very important property is that all concepts of a given table form a *partial order* via $(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2 \Leftrightarrow A_1 \supseteq A_2$. It was proven that such set of concepts constitutes a complete lattice called *concept lattice* $L(T)$. For two elements (O_1, A_1) and (O_2, A_2) in the concept lattice, their *meet* $(O_1, A_1) \wedge (O_2, A_2)$ is defined as $(O_1 \cap O_1, \sigma(O_1 \cap O_1))$ and their *join* $(O_1, A_1) \vee (O_2, A_2)$ as $(\tau(A_1 \cap A_1), A_1 \cap A_1)$. A concept $c = (O, A)$ has *extent* $e(c) = O$ and *intent* $i(c) = A$.

Theorem 1. *The Fundamental Theorem on Concept Lattice.* Let $(\mathcal{O}, \mathcal{A}, T)$ be a context. Then $\mathcal{C}(\mathcal{O}, \mathcal{A}, T)$ is a *complete lattice* and infimum and supremum are defined as follows:

$$\begin{aligned} \bigwedge_{t\in T}(O_t, A_t) &= \Big(\bigcap_{t\in T} O_t, \sigma\big(\tau(\bigcup_{t\in T} A_t)\big)\Big) \\ \bigvee_{t\in T}(O_t, A_t) &= \Big(\tau\big(\sigma(\bigcup_{t\in T} O_t)\big), \bigcap_{t\in T} A_t\Big) \end{aligned} \tag{1}$$

More about concept analysis can be found in [GantW99], [SnelT97].

Concept lattice can be depicted by the usual as lattice diagram. It would however be too messy to label each concept by its extent and its intent. A much simpler *reduced labeling* is achieved if each object and each attribute is entered only once in the diagram. The name of object O is attached to the lower half of the corresponding object concept $c = (\tau(\sigma(O)), \sigma(O))$, while the name of attribute A is located at the upper half of the attribute concept $c = (\tau(A), \sigma(\tau(A)))$.

4 Organizational Structure Modeling

The tables of responsibilities specified in the previous chapter correspond with boolean tables described in concept analysis where objects of the relation are substituted by roles and attributes of objects are substituted by activities that the roles are responsible for.

Before we construct the conceptual lattice describing roles and their responsibilities from our showroom example we have to join two tables of responsibilities defined for each process separately. The reason is that we want to have one organizational structure for the showroom as a whole not for each of the defined processes. The result table is Table 3.

	Off.	*Ord.*	*Fin.*	*Check.*	*Hand.*	*Sel.*	*Pay.*	*Tak.*
Manager		×	×	×		×		
Salesman	×	×			×			
Technician	×				×	×		×
Accountant			×	×			×	

Table 3: Role Assignment for All Process

The set of concepts that can be derived from the joined table of responsibilities is in Table 4.

Concept lattice (Fig. 2) can be constructed from the set of described concepts using following rules defining a structure of the graph:

- Graph nodes represent concepts and arcs their ordering.
- The top-most node is a concept with the biggest number of roles in its extent (c_{MSTA} in our case).

C_{MSTA}	=({ Man., Sal., Tech., Acc. },	{ })
C_{MS}	=({ Man., Sal. },	{ Ord. })
C_{MT}	=({ Man., Tech. },	{ Sel. })
C_{MA}	=({ Man., Acc. },	{ Fin., Check. })
C_{ST}	=({ Sal., Tech. },	{ Off., Hand. })
C_{M}	=({ Man. },	{ Ord., Fin., Check., Sel. })
C_{S}	=({ Sal. },	{ Off., Ord., Hand. })
C_{T}	=({ Tech. },	{ Off., Hand., Sel., Tak. })
C_{A}	=({ Acc. },	{ Fin., Check., Pay., })
$C_{\emptyset}$	=({ },	{ Off., Ord., Fin., Check., Hand., Sel., Pay., Tak. })

Table 4: Set of concepts.

- Concept node is labeled with an activity if it is the largest concept with this activity in its intent.

- Concept node is labeled with role if it is the smallest concept with this role in its extent (reduced labeling).

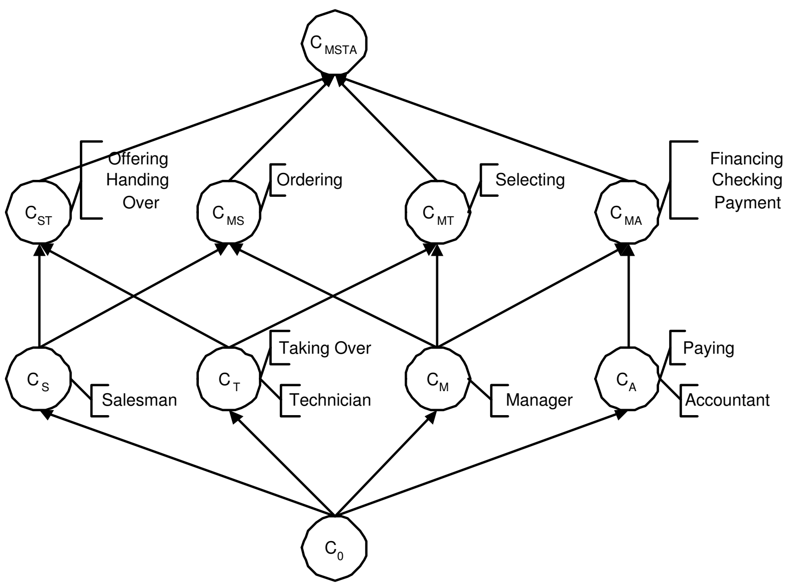

Fig. 2: Concept Lattice of the Organizational Structure

Resulting graph provides alternate views on the information contained in the above-described table. In other words, the concept lattice enables to visualize the structure "hidden" in the binary relation. In our example we can see that the technician is the only one who can take over delivered cars but he/she can also select a fleet of cars as well as the manager or to offer and hand over car like the salesman. Obviously, the more complex is the table of responsibilities the more difficult is to understand who is responsible for what.

5 Re-engineering

Visualization of the organizational structures opens new possibilities to its re-engineering. The concept lattice described in previous chapter Fig. 2 shows that the accountant resp. the technician are responsible for paying resp. taking over activities and thus cannot be substituted by anybody else. On the other hand, the technician in case of offering a car and handing over activity can substitute the salesman as well as the manager can substitute the salesman in the ordering activity. It means that the salesman can be removed. The manager has five activities that he/she is responsible for. If we remove his/her responsibility for checking payment activity then we obtain simplified organizational structure with a new graph of the structure in Fig. 3.

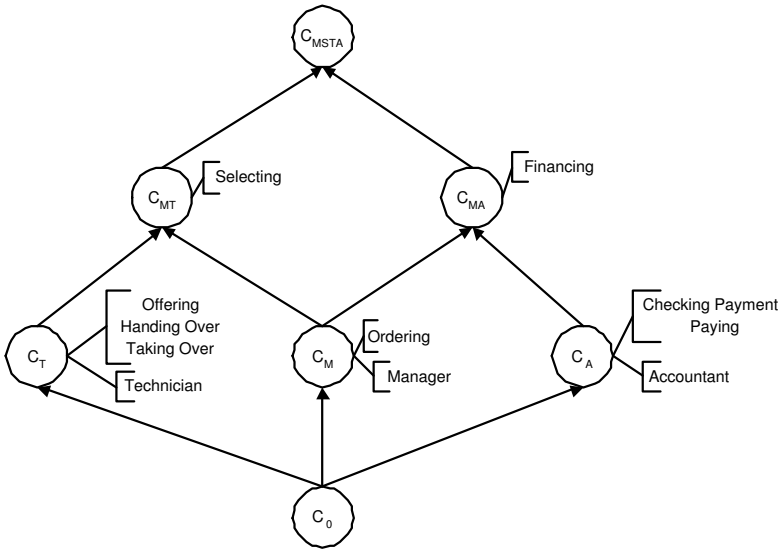

Fig. 3: New Organizational Structure

It looks that such kind of organizational structure is better balanced than the previous one because all roles have responsibility only for three activities except the technician that has four of them.

Since the lattice and table can be reconstructed from each other we are able to define new version of table of responsibilities for the given organizational structure.

	Off.	*Ord.*	*Fin.*	*Check.*	*Hand.*	*Sel.*	*Pay.*	*Tak.*
Manager		×	×			×		
Technician	×				×	×		×
Accountant			×	×			×	

Table 5: New Assignment of Roles.

6 Software description

We developed software to analyze small examples. This software constructs and draws conceptual lattice from a table. The software can label concepts with roles and activities. Each concept has a tool tip text with description. The description contains a full list of activities and roles, which represents the concept. All concepts with the same number of roles are drawn in the same horizontal line. This line represents one layer. Users can edit the table or the concept lattice, and the software keeps consistence of data in both views.

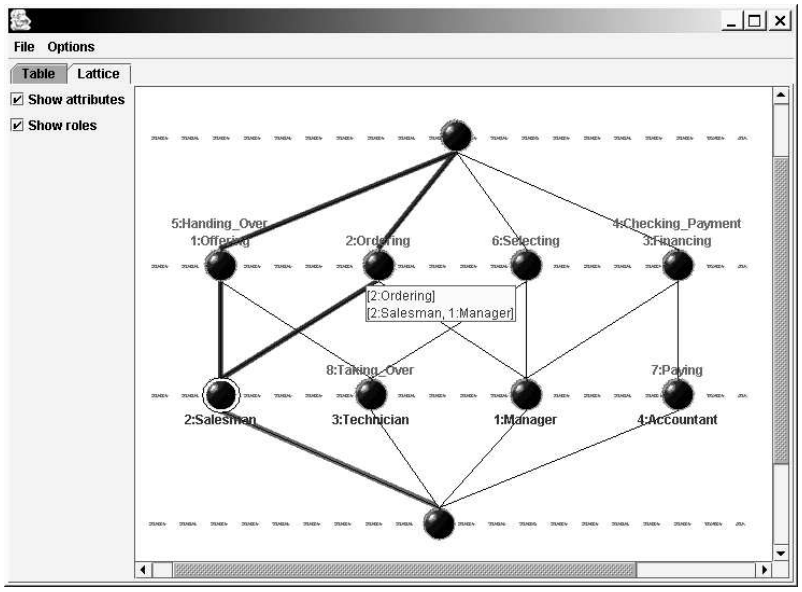

Fig. 4: Screenshot from our software

There is a problem: how to sort concepts in single layers to produce a *nice* concept lattice. In *nice* concept lattice the arcs intersect in minimum places. That is why users can drag concepts and move them on any position in their layer.

Users can select one concept and highlight concepts which contain all roles from this concept, or concepts witch contains all activities from this concept. In Fig. 4 first concept is selected from second layer. This is concept C_S, from the example described above. Selected arcs above concept C_S show concepts which contain all roles contained in concept C_S. All roles from concept C_S (Salesman) are in relation to all activities in concepts accessible by red arcs. Selected arcs under concept C_S show concepts which contain all activities contained in concept C_S. All activities from concept C_S (*Offering*, *Ordering*, *Handing Over*) are in relation to all roles in concepts accessible by green arcs.

This software allows to work with small tables, but lattice for a big table is more complicated and cannot be drawn this way. We must use another way to draw big lattice, or find a way how group concept in order to minimize their quantity.

7 Conclusions

Presented method of concept analysis provides exact and formally well defined way how the organizational structure can be analyzed and re-designed. The example used in our paper was simplified but it demonstrated sufficiently the potential of concept lattices and the way how they can be adopted for purposes of re-engineering. The future research is focused on building appropriate software tools that will enable to deal with much larger examples than the presented one and to verify the method in real-life situations.

Rereferences

[Chri95] Christie A., *Software Process Automation*, Springer-Verlag 1995

[Bir93] Birkhoff G., *Lattice Theory. 3rd edition*, American Mathematical Society, Providence, RI., 1993

[Mar79] DeMarco T., *Structured Analysis and System Specification*, Prentice-Hall, Englewood Cliffs, New Jersey 1979

[GantW99] Ganter B., Wille R., *Formal Concept Analysis, Mathematical Foundation*, Springer-Verlag 1999

[Pet77] Peterson J.L., "Petri Nets." *ACM Computing Surveys, vol.9, no.3 (Sept): 223-251*, 1977

[SnelT97] Snelting G., Tip F., *"Reengineering Class Hierarchies Using Concept Analysis" Research Report RC 21164(94592)24APR97*, IBM Research Division, USA (Apr), 1997

[VonSK99] Vondrak I., Szturc R., Kruzel M., "Company Driven by Process Models." *In Proceedings of European Concurrent Engineering Conference ECEC '99* (Erlangen-Nuremberg, Germany), SCS, Ghent, Belgium,. 188-193, 1999

Using XSLT for IS Simulation[1]

Karel Richta, Phạm Kim Long

Dept. of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Prague 2, Karlovo nám. 13, 121 35

e-mail: richta@fel.cvut.cz

Abstract. This paper presents a method of using XML-XSLT to implement term rewriting used in information systems prototyping. The description of an information system is converted into XML-formatted documents. The syntax for these XML documents is defined via DTD or XML-schema. Information system services are translated from a specification into a XSLT-code. This code simulates services behaviour via rewriting an input request into an output document.

Key words: XML,XSLT,prototyping

1 Introduction

Algebraic specifications with its clear syntax and semantics can be used for system specification and prototyping. An algebraic specification consists of two parts - a signature and a set of axioms. The signature serves as a definition of the syntax. The axioms specify the semantics. Any algebraic structure satisfying axioms is a so-called model of a specification. The variety of all models of a specification always contains the subclass of so called initial models. The meaning of a specification is the isomorphic class of all initial models - it means, the class of models, whose are exchangeable [Bergstra89].

The specification can be prototyped - the problem is to find out automatically any member of the initial class. One possible way is to use a unique symbolic model constructed from Herbrand's universe of all well-formed terms by the smallest congruence relation generated by axioms of a specification. This symbolic model always exists, and if we are able to construct it, the result depicts so-called decision procedure for

[1]This work has been partially supported by the research program no. MSM 212300014 "Research in the Area of Information Technologies and Communications" of the Czech Technical University in Prague (sponsored by the Ministry of Education, Youth and Sports of the Czech Republic), and it also has been partially supported by the grant No. CTU 300109713 "Using metadata in information transfer" of the Czech Technical University in Prague.

the equality problem in the defined class. A decision procedure is usually modeled by term rewriting systems [DJ89].

The problem how to construct an appropriate term rewriting system for a given algebraic specification is solved by the well-known Knuth-Bendix completing procedure [DJ89], which completes the appropriate term rewriting system for a given specification. If the procedure succeeded, we receive a term rewriting system that solves the equality problem for the specification and we can use it as a base for the construction of a prototype.

This paper discusses the construction of a prototype from a term rewriting system. The key problem in prototyping is to produce a prototype in a short time and at a sufficient level of efficiency for testing. A very simple method is to transform the term rewriting system to Prolog or another similar high-level logical programming language [Bergstra89]. In such a case the process of construction is simple, but the resulting product is directly dependent on the efficiency of a Prolog compiler.

Another possibility is to translate the term rewriting system into a functional language [Prívara88]. This process is not so straightforward as the above conversion to Prolog, but the resulting product should be more efficient due to an essentially simpler execution. The basic difference is that there is no need for general unification in prototyping (for parameter passing). The sufficient concept is matching of parameters with function definition's equations. Unfortunately, the efficiency is still dependent on the functional language compiler.

Another possible optimization consists of developing an abstract machine dedicated for this purpose. Generally it could be Warren's abstract machine [Warren83] designed for Prolog and therefore it can also consequently be used for rewriting. But such a choice is not the most efficient one because the code of Warren's machine is designed for unification. However, the matching machine can be made much simpler, see [RN91].

The paper [Richta01] initiates a method of using XML and XSL for term rewriting. Input algebraic specifications are converted into XML-format and expressed as XML documents. Such XML documents belong to a document class defined by a DTD (Document Type Definition). This DTD defines the syntax for writing valid algebraic specifications in XML-format. The semantic part of a specification is converted into XSL-code, which serves as a rewriting system. Prototyped expressions are expressed as XML-formatted terms, which are to be rewritten with the help of XSL into a canonical form - their meaning.

This paper will develop further the idea in [Richta01] emphasizing on the use of XSL transformations for prototyping. The significant achievement is we build generic XSLT programs, which compile the specification in XML format into the XSL specific code that will serve as the prototyping tool.

A typical algebraic specification expressed in XML/XSL consists of following parts:

- An XML document specifying the signature as well as rewriting rules.
- An XSLT document which serves as the rewriting engine for prototyped expressions.

- Prototyped expressions are expressed as XML-formatted terms, which are to be rewritten by applying the XSLT engine to them.

The syntax for writing signatures, rewriting rules and terms are defined by a generic DTD. It's very interesting to note that the XSLT rewriting engine can be generated from the specification itself with the help of a generic XSLT document.

The first part will review the syntax proposed in [Richta01] for expressing algebraic specifications in XML format. In the second part, we briefly mention implementation issues for transforming XML term documents. The third part will present a technique of compiling to the signature to produce the desired XSLT document used for term rewriting.

2 Algebraic specification in XML format

In the following we suppose that an *algebraic specification* **S** is the couple formed by a *signature* Σ and a finite set of (possibly conditional) *equations*. The signature Σ of the specification introduces all symbols permissible to denote objects in the defined world. The signature consists of the definition of all *sorts* of data and declarations of all admissible *operations* with their arities.

An *equation* is a pair of well-formed terms L, R over signature Σ, written as $L = R$. Let t_1 and t_2 be well-formed terms of the same sort d over Σ. The expression $t_1 == t_2$ is the *atomic well-formed* condition where the symbol $==$ denotes the identity of the sort d. Let $P1$ and $P2$ be two *well-formed conditions* over Σ, then the following expressions are also well-formed conditions over Σ:

```
not(P1), (P1 and P2), (P1 or P2).
```

The *conditional equations* are equations extended by a well-formed condition P over Σ and written as $L = R$ *if* P. All equations are supposed to be universally quantified over each variable occurring in it. We will use the OBJ-like notation [Bergstra89] for expressing specifications. For example, a stack abstract type can be described by the following specification:

obj Stack **is**
  sorts Nat, Stack
  opns emptyStack: Stack
    push: Nat Stack → Stack
    pop: Stack → Stack
  var S: Stack
  var N: Nat
  eqns pop(push(N,S)) = S
endo

Let us suppose that R is a term rewriting system completed via Knuth-Bendix procedure for an algebraic specification S. R consists of a set of rewriting rules. The

rewriting rule is a pair of well-formed terms L, R over signature Σ, written as $L \to R$. The conditional rewriting rule is a rewriting rule extended by a well-formed condition P over Σ and written as $L \to R \; if \; P$. The rewriting rule can be viewed as an oriented equation. The stack abstract type can be described by the following rewriting system:

trs Stack **is**
  sorts Nat, Stack
  opns emptyStack: Stack
    push: Nat Stack → Stack
    pop: Stack → Stack
  var S: Stack
  var N: Nat
  rules pop(push(N,S)) → S
endtrs

This term rewriting system is the canonical rewriting system for Stack-terms. It means that any constant well-formed stack-term can be rewritten into the canonical normal form of it that serves as a meaning of it. For example:

$pop(pop(push(2, push(1, emptyStack))))$
$\to pop(push(1, emptyStack)) \to emptyStack$

The term rewriting system can be expressed in XML format. One possible DTD can be:

```
<!ELEMENT trs (sorts, opns, rules)+>
<!ATTLIST trs
        names CDATA #IMPLIED
>
<!ELEMENT ident EMPTY>
<!ATTLIST ident
        name CDATA #REQUIRED
        type (sort | const | var) #REQUIRED
>
<!ELEMENT sorts (sort)+>
<!ELEMENT sort (ident)>
<!ELEMENT opns (op)+>
<!ELEMENT op (ident, args, result)>
<!ELEMENT args (arg)*>
<!ELEMENT arg (sort)>
<!ELEMENT result (sort)>
<!ELEMENT const (ident)>
<!ELEMENT var (ident, sort)>
<!ELEMENT term (const | var | apply)>
<!ELEMENT apply (term)+>
<!ATTLIST apply
        functor CDATA #REQUIRED
>
<!ELEMENT rules (rule)+>
```

```
<!ELEMENT rule (term, term)>
<!ELEMENT termdoc (term)+>
<!--
<!ELEMENT head (const)>
<!ELEMENT tail (term)*>
```

The example of stack specification is expressed in XML format as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE trs SYSTEM "algespec.dtd">
<trs>
<sorts>
   <sort><ident name="Nat" type="sort"></ident></sort>
   <sort><ident name="Stack" type="sort"></ident></sort>
</sorts>

<opns>
   <op>
      <ident name="emptyStack" type="const"></ident>
      <args/>
      <result>
         <sort><ident name="Stack" type="sort"></ident></sort>
      </result>
   </op>
   <op>
      <ident name="push" type="const"></ident>
      <args>
         <arg><sort><ident name="Nat" type="sort"></ident></sort></arg>
         <arg><sort><ident name="Stack" type="sort"></ident></sort></arg>
      </args>
      <result>
         <sort><ident name="Stack" type="sort"></ident></sort>
      </result>
   </op>
   <op>
      <ident name="pop" type="sort"></ident>
      <args>
         <arg><sort>
            <ident name="Stack" type="sort"></ident></sort></arg>
      </args>
      <result>
         <sort><ident name="Stack" type="sort"></ident></sort>
      </result>
   </op>
</opns>

<rules>
   <!- rule: pop(push(X,S)) ==> S -->
   <rule>
      <!-- left term -->
```

```
      <term><apply functor="pop">
         <term><apply functor="push">
            <term><var>
               <ident name="X" type="const"/>
               <sort>
                  <ident name="StackElement" type="const"/>
               </sort>
            </var></term>
            <term><var>
               <ident name="S" type="const"/>
               <sort>
                  <ident name="Stack" type="const"/>
               </sort>
            </var></term>
         </apply></term>
      </apply></term>

      <!-- right term -->
      <term><var>
         <ident name="S" type="const"/>
         <sort>
            <ident name="Stack" type="const"/>
         </sort>
      </var></term>

   </rule>
</rules>
</trs>
```

Following is the term push(2,pop(push(3,emptyStack))) expressed in XML format.

```
<?xml version="1.0"?>
<!DOCTYPE termdoc SYSTEM "algespec.dtd">
<termdoc>

<term><apply functor="push">
   <term><const>
      <ident name="2" type="const"/>
   </const></term>

   <term><apply functor="pop">
      <term><apply functor="push">

         <term><const>
            <ident name="3" type="const"/>
         </const></term>

         <term><const>
            <ident name="emptyStack" type="const"/>
         </const></term>
```

```
      </apply></term>
   </apply></term>
</apply></term>

</termdoc>
```

3 XSLT and rewriting issues

XSLT is a language using XML syntax, which allows transformation of XML documents. The structure of the result document tree can be completely different from the structure of the source tree. XSLT programs are stored in stylesheets.

An XSL style sheet consists of a set of template rules. A template rule has two parts: a pattern that is matched against nodes in the source tree; and a template that will be instantiated in the context of the matching nodes to form part of the result tree.

XSLT engine takes one XML and one style sheet as its inputs and produces a XML document, which is the result of applying the input style sheet on the input XML document. XSLT is widely used in web applications. One of the advantages of XSLT is its concise but powerful primitives.

In our term-rewriting systems, once the terms are written as XML documents, it can be easily rewritten with the help of XSLT.

3.1 General structure of XSLT stylesheets for term-rewriting

An XSL rewrite style sheet can be constructed from a rewrite system as follows:

- For each rewrite rule of the rewrite system, there's is a corresponding rewrite template. A rewrite template specifies a condition that term must match to be rewritten. The body of a rewrite template specifies how a matching term will be rewritten. More precisely, for each rewrite rule, its left hand specifies the matching condition of the rewrite template, meanwhile its right-hand term defines the content of the rewrite template, which will be instantiated in the context that the left hand term is matched.

- There are some fixed default templates, which will be applied to terms that do not match any rewrite template. These templates simply copy the term in context to the destination and continue the transformation for sub terms (if any) of the context term. In our rewrite stylesheet these default templates have a priority of 0 that is lower that that of rewrite templates.

Let us first describe the default templates mentioned above. For constant and variable terms, we just copy the terms through to the output.

```
<xsl:template match="term[var or const]" priority="0">
   <xsl:copy-of select="."/>
</xsl:template>
```

For function terms, which contain subterms, the default template copy the function term (parent term) through and proceed with the subterms so that these subterms can be rewritten.

```
<xsl:template match="term[apply]" priority="0">
   <xsl:copy>
      <!-- copy attributes -->
      <xsl:for-each select="@*">
         <xsl:copy/>
      </xsl:for-each>

      <!-- apply transformation child terms -->
      <apply>
         <xsl:for-each select="apply/@*">
            <xsl:copy/>
         </xsl:for-each><xsl:apply-templates select="apply/term"/>
      </apply>
   </xsl:copy>
</xsl:template>
```

In our example, the rule $pop(push(N, S)) \rightarrow S$ is translated to the following rewrite template.

```
<xsl:template priority="0.5"
   <!-- pattern matching -->
   match="term[apply/@functor='pop' and
               apply/term[1]/apply/@functor='push' and
               apply/term[1]/apply/term[1]/* and
               apply/term[1]/apply/term[2]/*]">

   <!-- right-hand term -->
   <xsl:apply-templates select="apply/term[1]/apply/term[2]"/>

</xsl:template>
```

The full rewrite style sheet can be found at the [PhạmKimLong02].

4 One Experiment

Problem Statement: We need to generate certifications of students for different organisations (they require same data but in different formats).

One possible solution is to store students' data into XML-document (described by POS1.DTD).

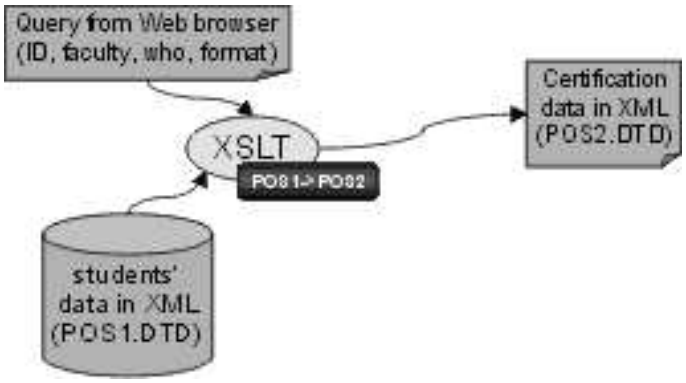

Fig. 1: Extraction of required data

```
<?xml version="1.0" encoding="iso-8859-2"?>
<!DOCTYPE Fakulty SYSTEM "pos1.dtd">
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="xsl/2pos2.xsl" type="text/xsl"?>

<Fakulty obdobi="2000/2001" datum="15. 4. 2001">

<Fakulta idFakulty="F4" celyNazev="informatiky a statistiky">

<Student uzivatel="xaaaa10" typStudia="Ing">
<Jmeno>John Smith</Jmeno>
<DatumNarozeni>1.1.1980</DatumNarozeni>
<Bydliste>Praha</Bydliste>
<Semestr>3</Semestr>
</Student>

</Fakulta>
</Fakulty>
```

Required data of the student (given by faculty and ID) are retrieved from the database and stored again as XML-document (with the structure defined by POS2.DTD)

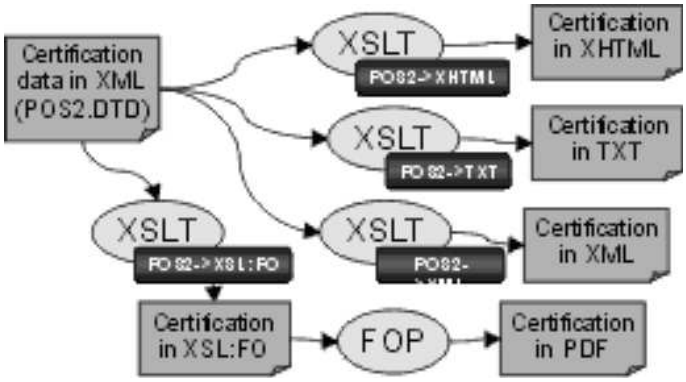

Fig. 2: Formatting report

```
<?xml version="1.0" encoding="iso-8859-2"?>
<!-- 2pos2.xsl - transformace dokumentu XML ze schematu pos1.dtd
     -> pos2.dtd
-->
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:param name="id"/>
 <xsl:param name="who"/>
 <xsl:param name="faculty"/>
 <xsl:param name="format"/>

 <xsl:template match="Fakulta">
   <xsl:if test="@idFakulty = $faculty">
     <xsl:apply-templates/>
   </xsl:if>
 </xsl:template>

 <xsl:template match="Student">
   <xsl:if test="@uzivatel = $id">
      <!-- transformation -->
   </xsl:if>
 </xsl:template>

</xsl:stylesheet>
```

Resulting intermmediate document can look like follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<?cocoon-process type="xslt"?>
<?xml-stylesheet href="pos2-HTML.xsl" type="text/xsl"?>

<certificate>
  <header>
    <Faculty>Electrical Engineering</Faculty>
    <date>15.8.2001</date>
  </header>
  <paragraph>
    <row>
      <text>The Dean of The Faculty of </text>
      <Faculty>Electrical Engineering</Faculty>
      <text> certifies that </text>
    </row>
    <row>
      <text> Mr. </text>
      <Name>John Smithaa10</Name>
```

```
      <text>, date of the birth </text>
      <BirthDate>1.1.1980</BirthDate>
    </row>
    <row>
      <text> address: </text>
      <Address>Praha</Address>
      <text> is the </text>
      <GradType>master</GradType>
      <text> student of our faculty.</text>
    </row>
  </paragraph>
  <paragraph>
    <row>
      <text>He/she is in his/her </text>
      <Year>3</Year><text>-rd year of his/her study</text>
      <text> in the period </text>
      <Period>2000/2001</Period>
      <text>.</text>
    </row>
    <row>
      <text> This certificate is for: </text>
      <Who>insurance company</Who>
      <text>.</text>
    </row>
  </paragraph>
</certificate>
```

The intermmediate document can be converted into different output formats.

4.1 The Results of Experiments

In [Kudibal01] there are pressented results of this experiment done on AMD K5 300 MHz, 128 MB RAM, Windows NT 4.0 SP3, XSL-processor Xalan, XML-processor Xerces:

#students	#elements	response time (s)
1000	5002	3
4000	20002	8
7000	35002	19
10000	50002	20
13000	65002	30

5 Conclusions

The paper proposed a method of using XML and XSL in the information system prototyping. Although more works need to be done to evaluate our method, the achieved results show that using XML and XSLT in specification prototyping could be a viable and promising method. Thanks to the power of XSLT as a query and pattern matching language, the rewrite rule compiler written is quite concise, and thus easy to maintain and to develop further. Although XSLT is not a procedural language and limited in some areas, hopefully simple extensions to XSLT can be added to solve possible problems.

Rereferences

[Bergstra89] Bergstra, J.A. - Heering, J. - Klint, P.: *Algebraic Specification.* ACM Press, ISBN0-201-41635-2, Addison-Wesley 1989.

[BR00] Bisová, V. - Richta, K.: *Transformation of UML Models into XML.* In: Proceedings of Challenges 2000 ADBIS-DASFAA. Praha: MATFYZPRESS UK ISBN 80-85863-56-1, pp. 33-45. Praha 2000.

[DJ89] Dershowitz, N. - Jouannaud, J.P.: *Rewrite systems.* Handbook of Theoretical Computer Science, North-Holland 1989.

[DL90] Dershowitz, N. - Lindenstrauss, N.: *An Abstract Concurrent Machine for Rewriting.* Proc. of Algebraic and Logic Programming, LNCS vol. 463, (Kirchner, H. and Wechler, W. ed., Springer Berlin 1991), pp. 318-331, October 1990.

[Kudibal01] Kudibal, I.: *Using XML Technology at University of Economics in Prague.* Diploma thesis, Dept. of IT UE Prague, (in Czech) 2001.

[NR90] Nešvera, Š. - Richta, K.: *The efficient implementation of rewriting.* Proc. SOFSEM'90, vol.2., Masaryk University of Brno (in Czech), Janské Lázně, December 1990.

[PhạmKimLong02] Long,P.K.: *Dissertation Thesis.* In preparation. 2002

[Prívara88] Prívara, I. - Šatura, F.: *From An Algebraic Specification to A Functional Program.* Technical report VUSEI-AR-OPS-3/88, Dept. of Programming Systems, Institute of Socio-Economic Information and Automation in Management, Bratislava 1988.

[PR00] Pokorný, J. - Richta, K.: *XML a semistrukturovan data.* In: Proceedings of DATASEM 2000. Brno: Masaryk University - ISBN 80-210-2428-3, pp. 47-63. Brno 2000.

[Richta80] Richta, K.: *Abstract Data Types and Their Implementation.* PhD thesis, Department of Computer Science, Czech Technical University of Prague, (in Czech) Prague 1980.

[Richta91] Richta, K.: *An Algebraic Specification Prototyping*. Thesis, Dept.of Computer Science, Czech Technical University of Prague, Praha 1991.

[RN91] Richta, K. - Nešvera, Š.: *The Abstract Rewriting Machine*. Proc. of the 3-rd Logical Programming Winter School and Seminar (LOP'91), Ruprechtov, pp. 179-185. Masaryk University of Brno, January 1991.

[Richta00] Richta, K.: *Formáty XML a XSL*. In Proc. of Moderní Databáze 2000, pp. 1-23. Mělník 2000.

[Richta01] Richta, K.: *Using XSL in IS Development*. In: Proc.of ISD 2001, Royal Holloway, Egham, 2001.

[Warren83] Warren, D.H.D: *An Abstract Prolog Instruction Set*. Technical Note 309, Artificial Intelligence Centre, SRI International 1983.

[XML] World Wide Web Consortium Recommendation. *Extensible Markup Language (XML) 1.0*. February, 1998. `http://www.w3.org/TR/REC-xml`.

[XSLT] World Wide Web Consortium recommendation. *XSL Transformation (XSLT) Version 1.0*. November, 1999. `http://www.w3.org/TR/xslt`.

[XPath] World Wide Web Consortium recommendation. *XML Path Language (XPath) Version 1.0*. November, 1999. `http://www.w3.org/TR/xpath`.

Navigation through Query Result Using Concept Lattice

Václav Snášel, Daniela Ďuráková, Michal Krátký

Department of Computer Science, VŠB-Technical University of Ostrava
17. listopadu 15, 708 33 Ostrava-Poruba, Czech Republic

e-mail: {vaclav.snasel, daniela.durakova, michal.kratky}@vsb.cz

Abstract. In this article we describe using the concept lattice for the navigation through the query result. The user often gets a very large result set. We used the concept lattice for the ordering the query result in the hierarchical structure. It makes possible to navigate through the query result and this way we can choose the optimal object from result set.

Key words: concept lattice, hierarchical structure, navigation through the query result

1 Introduction

The built up lately information systems have a gigantic amount of data stored inside. The software market offers various methods of processing the stored data. For the users it is not sufficient to select data complying with the given demands only, but they expect selected information to be helpful for them in decisions of everyday work. In situation, when selections from database of ones own data are made or public databases are used, in many cases the answer is returned as the extensive set of acceptable (proper) data.

In the area of geographic information systems (GIS) two kinds of access to the above mentioned problem are apparent. In this contribution a method is proposed to facilitate hierarchical order of results of query in GIS. For this hierarchical ordering a theory of concept lattices is applied (chapter 2). In the example of selection a ski centre an application of concept for hierarchical order of query results from the geographic information systems is presented.

2 Context and concept

The idea of formalisation terms of context and concept by means of theory of lattices is not quite new. Formerly isolated experiments at application so-called Galois lattices had already existed, especially in the area of information retrieval, but systematically built up theory did not arise until the work of R.Wille and his group [3].

The claim of this contribution is only brief description of the application of the formal context and conceptual lattices. The terms from classic disciplines of a theory of ordered sets and the theory of lattices can be found in [1].

Definition 1. *A context* is a triple (G, M, I), where G and M are sets and $I \subseteq G \times M$. The elements of G are called *objects* and the elements of M are *attributes*.

We write gIm or $(g, m) \in I$ and say "the object g has the attribute m". The relation I is called incidence relation of context.

Definition 2. For $A \subseteq G$ and $B \subseteq M$, define $A' = \{m \in M |\ gIm\ \forall g \in A\}$, $B' = \{g \in G |\ gIm\ \forall m \in B\}$ where A' is the set of attributes common to all objects in A and B' is the set of objects possessing the attributes in B.

Definition 3. Let (G, M, I) be a context. For a couple (A, B) where $A \subseteq G$, $B \subseteq M$, $A' = B$ and $B' = A$ we say A is the *extent* and the set B is the *intent* of concept (A, B). The set of all concepts is called $\mathcal{K}(G, M, I)$.

Definition 4. For concepts (A_1, B_1) and (A_2, B_2) in $\mathcal{K}(G, M, I)$ we say (A_1, B_1) is a *subconcept* of (A_2, B_2) if $A_1 \subseteq A_2$ (it is the same as $B_2 \subseteq B_1$) and we write $(A_1, B_1) \leq (A_2, B_2)$ and that (A_2, B_2) is a *superconcept* of (A_1, B_1).

The relation $\leq$ is the relation of order on the set of all concepts $\mathcal{K}(G, M, I)$. This relation is even lattice order on this set, that means that there exists supremum and infimum with regard to $\leq$ for every two elements from $\mathcal{K}(G, M, I)$. The proof and further details can be found in [1].

This relation on the concepts introduces a hierarchical structure, with the requirements for lattice order which can be described by a mathematic apparatus.

Further the fundamental theorem on concept lattice will be mentioned. The theorem contains two predications — concept lattice is complete and performs a model for calculation of supremum and infimum. Supremum will be indicated by symbol $\bigvee$ and infimum by symbol $\bigwedge$.

The Fundamental Theorem on Concept Lattice. Let (G, M, I) be a context. Then $(\mathcal{K}(G, M, I); \leq)$ is a *complete lattice* and infimum and supremum are defined as follows:

$$\begin{aligned} \bigwedge_{t \in T} (A_t, B_t) &= \Big(\bigcap_{t \in T} A_t, (\bigcup_{t \in T} B_t)'' \Big) \\ \bigvee_{t \in T} (A_t, B_t) &= \Big((\bigcup_{t \in T} A_t)'', \bigcap_{t \in T} B_t \Big). \end{aligned} \tag{2}$$

The proof of this theorem can be found in [1]. The using of these terms will be presented in the following chapter.

3 The creation of a concept lattice

Let us imagine the situation in the travel agency where the staff execute the clients demands for ski centre of various character. Often the situation comes that they do

their best to comply with the most different clients requirements which can be very specific. Let us attempt to follow the process in our example.

Example 1. Consider ski centres into Austria and Italy. There are the 18 ski centres into our database (see Figure 1 and Table 1). Attributes of the ski centre are a distance from Prague (d), a price of ski-pass (s), an elevation (e), a length of pistes (l) and number of tows (t).

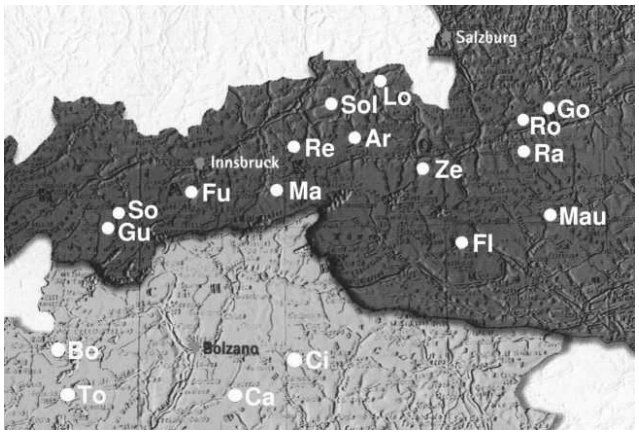

Fig. 1: Map of Austria and Italy ski centres.

Ski Centre	Abbreviation	d	s	e	l	t
Mayrhofen	Ma	483	5276	3250	451	161
Sölden	So	576	4866	3260	108	36
Gurgl	Gu	591	4866	3080	110	22
Fulpmes	Fu	520	4658	3150	62	37
Reith	Re	455	3373	2120	47	19
Arena Kitzbühel	Ar	465	4741	2000	158	64
Flattach	Fl	490	4411	3125	50	8
Söll	Sol	460	3664	1835	269	91
Lofer	Lo	422	2855	1747	47	14
Zell am See	Ze	482	4632	3029	130	55
Radstadt	Ra	450	4625	2130	345	136
Mauterndorf	Mau	501	4119	2360	128	31
Gosau	Go	390	3774	1600	65	33
Rohrmoos	Ro	426	4565	2700	167	87
Cavalesa	Ca	730	2158	2400	100	50
Ciapela	Ci	780	2311	3340	59	32
Tonale	To	800	1781	3017	80	29
Bormio	Bo	720	2072	3012	195	70

Table 1: The data about the ski centres.

Example 2. The client wants to choose ski centres whose distance from Prague is under 810 km, the price of ski-pass is under 4000 Kč, the elevation is above 2500m, the length of pistes is above 99 km and number of tows is above 30.

First, we evaluate a query with these conditions. The query result must be changed to the context (Definition 1). Every numeral value is reduced to a boolean value. For example, if we will want to know the ski centres with the elevation above 2500 m, the centre Mayrhofen has got this property while the centre Gosau does not have got the property (see Table 1). In this way we obtain the context table (Table 2).

Now we use an algorithm [2] applying the fundamental theorem on concept lattice [4] and we construct the concept lattice for above mentioned requirements. The result of the algorithm is presented in Table 3. Then we can draw the Hasse diagram of the concept lattice (Figure 2).

	d	*s*	*e*	*l*	*t*
Ma	x		x	x	x
So	x		x	x	x
Gu	x		x	x	
Fu	x		x		x
Re	x	x			
Ar	x			x	x
Fl	x		x		
Sol	x	x		x	x
Lo	x	x			
Ze	x		x	x	x
Ra	x			x	x
Mau	x			x	x
Go	x	x			x
Ro	x		x	x	x
Ca	x	x		x	x
Ci	x	x	x		x
To	x	x	x		
Bo	x	x	x	x	x

Table 2: The context table for the ski centre example.

top	({Ma, So, Gu, Fu, Re, Ar, Fl, Sol, Lo, Ze, Ra, Mau, Go, Ro, Ca, Ci, To, Bo}, {d})
c_{11}	({Ma, So, Gu, Ar, Sol, Ze, Ra, Mau, Ro, Ca, Bo}, {d, l})
c_{10}	({Ma, So, Gu, Fu, Fl, Ze, Ro, Ci, To, Bo}, {d, e})
c_9	({Ma, So, Fu, Ar, Sol, Ze, Ra, Mau, Go, Ro, Ca, Ci, Bo}, {d, t})
c_8	({Re, Sol, Lo, Go, Ca, Ci, To, Bo}, {d, s})
c_7	({Ma, So, Gu, Ze, Ro, Bo}, {d, e, l})
c_6	({Ma, So, Ar, Sol, Ze, Ra, Mau, Ro, Ca, Bo}, {d, l, t})
c_5	({Ma, So, Fu, Ze, Ro, Ci, Bo}, {d, e, t})
c_4	({Sol, Go, Ca, Ci, Bo}, {d, s, t})
c_3	({Ci, To, Bo}, {d, s, e})
c_2	({Ma, So, Ze, Ro, Bo}, {d, e, l, t})
c_1	({Sol, Ca, Bo}, {d, s, l, t})
c_0	({Ci, Bo}, {d, s, e, t})
bot	({Bo}, {d, s, e, l, t})

Table 3: The concepts for the context of the ski centre example.

4 The navigation through the query result

We show the navigation principle through the created concept lattice and the presentation of this in the GIS background. We see the top of concept lattice to contain every ski centre with desired distance. We need to know client rank of the next requirement to help him to select of the best ski centre.

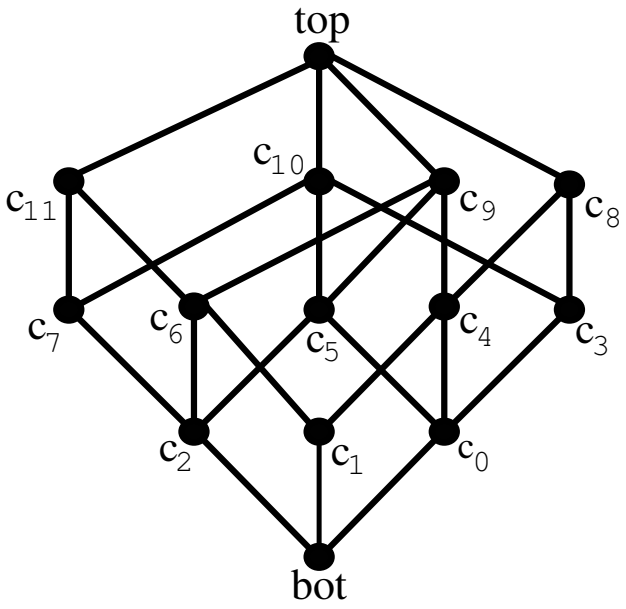

Fig. 2: The Hasse diagram of a concept lattice for the context of ski centre example.

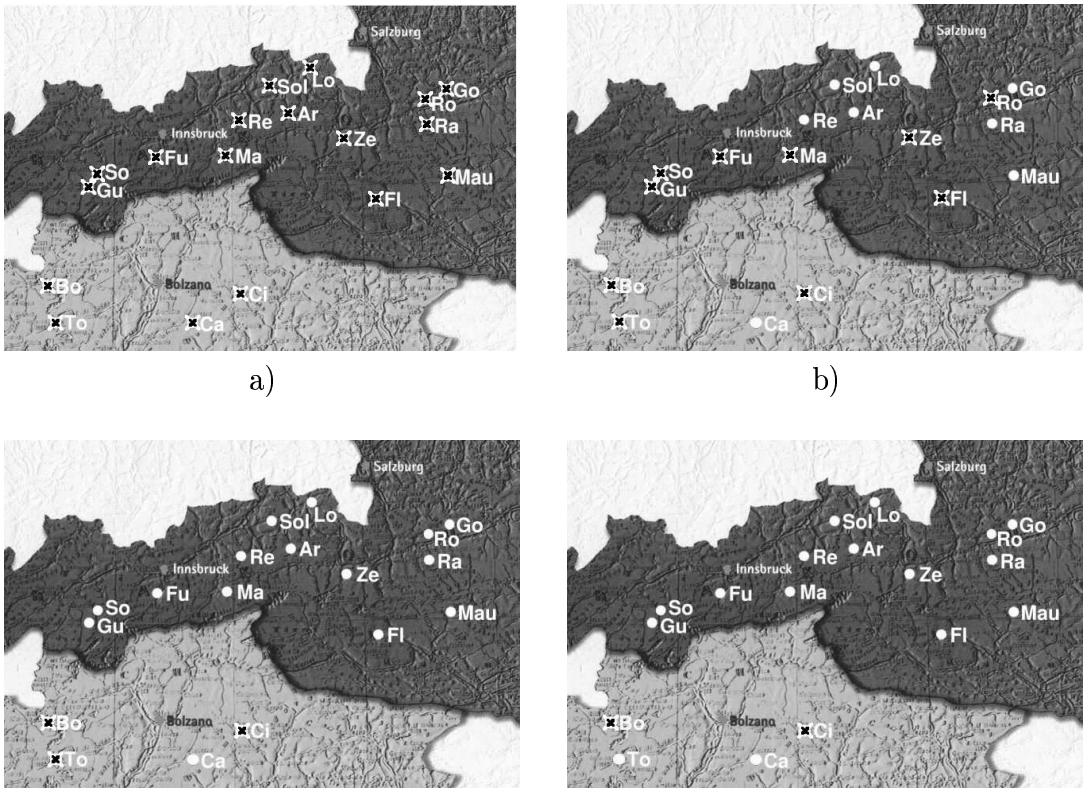

Figure 4: The narrowing of the objects from the example of the navigation (see Figure 3). The chosen objects are marked by ⊠ .

Assume the elevation is the most important requirement because snow is held above 2500 meters above sea level for a long time. We will leave the top concept and go to the concept with attributes d and e (see Figure 3 change the node 1 to

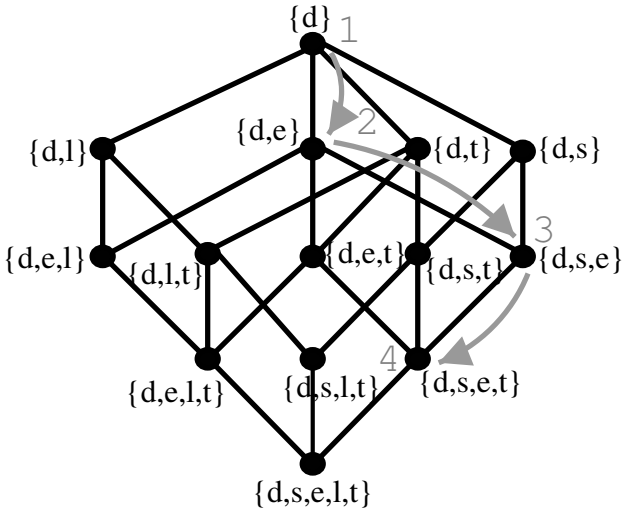

Fig. 3: The Hasse diagram of a concept lattice for the context of the ski centre example with displayed intents of the concepts and an example of the navigation in the concept lattice.

the node 2). The node 2 corresponds with a concept c_{10} (see Figure 2). We know the objects for the concept c_{10} from Table 3. The chosen objects are marked in Figure 4b.

If we could narrow down the number of ski centres we consider next property — a price of ski-pass. We change the node 2 to the node 3 in the diagram of Figure 3 (it contains attributes *d, s* and *e*) but we have not got only one object yet, see Figure 4c. Only if we add next parameter t – number of tows – we obtain marked node 4 of Figure 3. It is a concept c_0 with two objects and with a set of attributes *d, s, e* and *t*.

In this example it could continue to last node with all attributes but it depends on the client requirements.

5 Conclusion

In this article we have shown the possibility of application of the theory of concept lattices. As the most interesting can be considered:

- The creation of hierarchical structure of the query result.
- The instruction for decomposing the structure to single parts and for reducing the complicated hierarchical structure.
- The possibility of navigation in sets of objects with different level parameters of query requirements.

Rereferences

[1] B. Ganter, R. Wille. *Formal Concept Analysis, Mathematical Foundation.* Springer Verlag 1999.

[2] G. Snelting. *Reengineering of Configurations Based on Mathematical Concept Analysis.* ACM Transactions on Software Engineering and Methodology 5(2):146-186, April 1996.

[3] `http://www.mathematik.tu-darmstadt.de/ags/ag1/Literatur/liste/intern/en.html`

[4] R. Wille. *Restructuring lattice theory: an approach based on hierarchies of concepts.* In: I.Rival (ed.): Ordered sets. Reidel, Dordrecht-Boston, 445-470. 1982

Design Patterns in Functional Programming

Jan Hric

Dept. of Theoretical Computer Science
Charles University, Faculty of Mathematics and Physics
Malostranské náměstí 25
118 00 Praha 1

e-mail: jan.hric@mff.cuni.cz

Abstract. Design patterns are well-known technique used in a development of object-oriented systems for reusing solutions of typical problems. In the paper we analyse design patterns in the new context of functional programming. We compare the patterns to development techniques used in functional programming and we transfer some patterns to the new context.

Key words: design patterns, functional programming

1 Introduction

Design patterns [GHJ], [BMR] are used as standard solutions of typical problems of an object-oriented design. Some problems are language independent and so they are relevant also in a different context of functional programming. We took problems and their corresponding patterns from literature and look for corresponding patterns in functional programming.

Declarative programming provides support which is not available in object-oriented languages. Polymorphic functions and data structures and functional parameters are basic examples of such a support in both functional and logic languages. We chose the functional language Haskell for this paper because it has some other features compared to the logic language Mercury. We suppose that patterns can be transferred also to logic programming.

The paper concentrates on transferring patterns. Knowledge of particular design patterns and functional programming is an advantage during reading.

As noted in [Pr], the classification of patterns is of a little help for program developers. They need solutions for their problems. Thus we describe patterns in a new context and do not analyse their relations.

1.1 Level of patterns

High-level architecture of a program is independent on an implementation language. Thus high-level architectural patterns [BMR] can be used analogically in different languages (the patterns Blackboard, Microkernel). We are mostly interested in a lower level of patterns.

Low-level patterns, called programming idioms, are usually language specific. Therefore they can not be used as a source for a transfer. Moreover, some patterns in one language disappear in another language due to different possibilities of languages.

1.2 Comparison of object-oriented and functional programming

An object is the core entity in OOP. An object has a state and a composed interface and it associates data and functions. In functional programming there is no such a universal entity. So various means are used to describe design patterns, especially data structures, higher-order functions, type classes and modules. There is also a difference in a granularity of objects and data structures. One data structure usually corresponds to many interconnected objects.

The nonexistence of a state means that many patterns devoted to a processing or synchronization of states of one or more objects are not usable. The architecture of such programs is different and a problem formulated in the context of OOP disappears or must be reformulated for other entities than objects. One basic characteristic of pure functional languages is a referential transparency. Thus each function must get all data which are needed for computing of an output value. Therefore a (representation of a) state must be given in input data.

A direct reformulation of patterns in a functional programming sometimes gives too specific solutions. Such solutions can be generalised for other data structures or types.

1.3 Separation of hook and template

The idea behind many patterns is a decoupling. A hook part which should be flexible is hidden from the rest of the system and is called only through a template part. Possibilities of an actual implementation are described in the following subsection.

1.4 Means in functional programming

We do not have objects and their virtual methods in functional programming as an universal way of late binding. Patterns must be implemented using other low-level principles.

First possibility is to use parametric polymorphism. Data structures and functions

can be polymorphic and thus independent on a particular type of a parameter. Second possibility is to use functional parameters in higher-order functions. An appropriate code for a hook is explicitly given as a parameter. Third possibility is to use type classes and allow to select the particular operations during a (re)compilation. The last possibility is to use modules and abstract data types. Cooperating functions of a pattern are grouped together in the last two cases. Also some special features as extensible records can be suitable for a pattern description.

More OOP patterns will be reduced to the same or similar FP pattern. This is possible, as we can look at some patterns from different points of view.

The same program can be written using different programming styles. There are for instance continuation passing style, monadic style or compositional programming using combinators in functional programming. Such styles can use specific low-level patterns, which are not analysed here. Styles correspond to frameworks in some sense. There are special features and usual ways of combining parts in both cases.

Patterns can be aimed also at special domains. Hot spot identification combined with essential construction principles is suggested for a development of domain-specific patterns [Pr]. Combinators for specific domain are such (low-lever) patterns in functional programming.

2 Patterns

We take patterns from [GHJ] and look for corresponding ideas in a functional programming. Patterns described there are more general and less object-oriented in comparison with [BMR]. Some patterns solve problems too specific for object-oriented programming, especially questions of a state manipulation and synchronization in a wide sense.

The first three subsections describe structural, behavioral and creational patterns according to [GHJ]. For each pattern we describe an original central idea [HDP] in an object-oriented programming and then we start to analyse its functional twins.

2.1 Structural patterns

2.1.1 Adapter

The adapter pattern converts an interface of a class into another interface expected by a client.

This idea can be used for functions and for data structures. In the first case the interface of a function is its type. Each use of the pattern means to write an adapter function, which transforms the original adaptee function to a new one. The functions $flip$, $curry$ and $uncurry$ are examples of the pattern. Instead of the original incompatible function f we call the compatible function $(adapter\ f)$ in the same context.

```
flip           :: (a -> b -> c) -> b -> a -> c
flip f x y      = f y x
curry          :: ((a,b) -> c) -> (a -> b -> c)
curry f x y     = f (x,y)
uncurry        :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p     = f (fst p) (snd p)
```

In the second case of data structures, the adapter is a function which converts a data structure to another structure.

2.1.2 Composite

The composite pattern composes objects into tree structures to represent part-whole hierarchies. The pattern corresponds to a definition of a new type constructor $Tree$. Composite structures use a type $Tree\ a$ instead of a. Trees can be binary, n-ary etc.

2.1.3 Decorator

A decorator enable to attach additional responsibilities to an object dynamically.

A possible reformulation in a functional programming is that we want to extend the behaviour of a function for a given data structure. A simple approach is to give a function higher-order parameter f, which describes how the data structure should be processed. This solution has a disadvantage that the parameter describes the whole processing but is not extensible. Using the idea of continuations, the extensible solution is to use the parameter f with a hole – another functional parameter g. The latter function g describes only the additional processing and is substituted to the identity function id when nothing new is needed.

One note concerning a type of results. We suppose the same type of results for calls with an additional functionality and without it. So the type of results must be extensible and we must understand the semantics of results if we want to use them. Extensible structures in this sense are data structures as lists, trees etc. The semantics of old and new functionality can be captured in a lookup list. Each new functionality adds one (or more) key - value pair to the result. Other means as extensible records (TREX) in Hugs implementation [Hs] are available.

We need not understand the results when they are not processed or are processed uniformly. The results given directly to output are an example of the former case.

2.1.4 Proxy

The proxy patterns provide a surrogate or placeholder for another object to control access to it. There is a more specific pattern concerning data structures in functional programming. Instead of using data directly we use the name of data. For instance we can use the name of a vertex in a graph to hide an actual data about the vertex. The data represented by the name may be subject of change independently from the

names. Some look-up function must be called dynamically to get an actual data for the given name.

2.2 Behavioral patterns

2.2.1 Interpreter

If there is given a language, let's define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

In a functional programming we usually interpret structured data, so the data incorporate the used rules. From this point of view the process of parsing, i.e. building structure, can be separated. The rest is an interpretation. The general function for an interpretation of data structures of a given type is the higher-order function $fold$ for the type.

An interpretation of the constructors of the data type is given by functional parameters. Each constructor has one corresponding parameter.

The function $fold$ must be implemented for each type separetely in a typed language as Haskell. The ideas of polytypic programming [GHs] allow to write the $fold$ function once and automatically generate instances for various types. It means that the pattern can be expressed as a code in such extended language.

2.2.2 Iterator

Iterator provides a way to access elements of an aggregate object sequentially without exposing its underlaying representation. The idea can be transferred to a functional programming in two ways. They differ in understanding of the word sequentially. The first meaning is sequential data structure and the second one is sequentially in time.

In the first case we transform elements of an aggregate object to a list and then list-processing functions can be used. This is similar to the adapter pattern.

In the second case we prepare functions corresponding to an interface of an iterator. There are the functions *init*, *next*, *done* and possibly others for a given type a.

```
init    :: a -> St a
next    :: St a -> (a, St a)
done    :: St a -> Bool
```

The current state of iterator is captured in appropriate type $St\ a$ and is transferred among functions above using parameter. An implementation can use separate functions or can define type class of types equipped with an iteration.

Note that this pattern can be generalised. In both cases we are not restricted to the sequences but an element can have more following items. Such a generalised iterator can for instance implement the method "Divide et impera".

2.2.3 State

A state pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The functional counterpart to the pattern adds to functions one parameter representing a state. Depending on this parameter a function can alter its behavior. The representation of a state can be implemented using Reader monad.

2.2.4 Strategy and Template Method

The description of the strategy pattern is following. Define a family of algorithms, encapsulate each one, and make them interchangable. Strategy lets the algorithm vary independently from clients that use it.

This pattern disappears in a functional programming as a possibility to use functions as parameters enable directly parametrize functions with a strategy parameter.

The pattern Template Method is similar. Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

In this case we use more functional parameters. Each one refers to single step, which was deferred.

2.2.5 Visitor

The pattern Visitor represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

There are two types of visitor, the internal and the external. The first one performs given operation on all elements of the structure. This corresponds to the *map* function which gets the operation as a parameter. The second one needs to capture a state and an implementation is similar to the Iterator.

2.2.6 Pipes and Filters

This architectural pattern [BMR] provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data are passed through pipes between adjacent filters.

Processing (finite) lists or (infinite) streams is a standard technique in functional programming. The binding of adjacent processing steps is realised by a function composition. The *map* function process an input in one-to-one style. The $filter$ function (in functional terminology) leaves out some data. Both function have a functional parameter which describes the way of processing of an element in the first case and which data should remain in a stream in the second case. Other higher-order functions can support many-to-one or many-to-many processings.

2.3 Creational patterns

2.3.1 Builder

The Builder separates a construction of a complex object from its representation so that the same construction process can create different representations.

The data structures are built in a functional programming using constructor functions. We use the same style but instead real constructors we use virtual ones which hide the real construction process. Then we get the same effect in a functional programming.

A real implementation can use separate functions, type classes or a set of mutually recursive constructors which pass themselves to lower levels of a structure.

The described process of a construction is incremental and the real data structure can be repeatedly rebuild. So it may be more effective to give all data to the (abstract) construction process in one batch. The pattern can be also coded using the functions $fold$ and $unfold$. The first one can be used in cases when we have a structure and we want to reinterpret it. The second one enables replace constructors by given functional parameters during recursive building process.

3 Conclusion

We have shown that design patterns for many problems can be transferred to a functional programming and more generally to a declarative programming. Other problems and their patterns are too specific for an object-oriented programming, so we did not cover them in this paper. Also high-level architectural patterns and low-level patterns – programming idioms were left out.

As in OOP it is usually possible to write a template for the core of a pattern. The template and examples are important for usefullness of a pattern library. Patterns are interconnected and rules of thumb were formulated [PPR].

There is no single universal entity in a functional programming as is an object in OOP. The core idea of decoupling can be targetted to functions or to data structures and can be realized by various means. A comparison of various approaches is left for future work.

Some patterns correspond to well-known techniques in a functional programming. Other approach to analysis of correspondence can be taken. We can take such techniques and look for problems which they solve. A more general or more parametric pattern can be found using abstraction. Also an analysis of a relevance of published problems in a context of a functional (and logic) programming followed by a reformulation of the problems remains to be done.

Rereferences

[BMR] Buschmann F., Meunier R., Rohnert H., Sommerland P., Stal M., *A System of Patterns*, John Wiley, Chichester, England, 1996

[GHJ] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, USA, 1995

[GHs] `http://www.generic-haskell.org/`

[Hs] `http://www.haskell.org/`

[HDP] *Houston Design Patterns*,
`http://rampages.onramp.net/~huston/dp/patterns.html`

[PPR] *Portland Pattern Repository*,
`http://www.c2.com/cgi/wiki?PortlandPatternRepository`

[Pr] Pree W., *Object-Oriented Design*, SOFSEM'97, LNCS 1338, Springer-Verlag, Berlin, 1997

Series data preparation in KDD process

Michal Samek

Department of Computer Science, VŠB-Technical University of Ostrava
17. listopadu 15, 708 33 Ostrava-Poruba

e-mail: michal.samek@vsb.cz

Abstract. Data preparation phase is one of the most time and energy consuming phase of KDD[1] process. At the same time it is one of the most important phases of the whole process. It is obvious – if we analyze wrong data, we will achieve wrong results. This paper deals with special type of input data, namely series data. Reference model for series data preparation is presented.

Key words: knowledge discovery, data mining, data preparation, series data

1 Introduction

The general idea of knowledge discovery in databases is very attractive and intuitive. According to one of the most popular definitions, KDD is considered to be non-trivial extraction of implicit, previously unknown, and potentially useful information from data. Typically we focus our attention on analytical algorithms and methods, which provide prospective results (i.e. previously mentioned useful information). In fact there are many other tasks, which have to be performed to achieve our goals. There are for instance problem understanding, data collection, data preparation, results evaluation and utilization, and so on. Overview of the whole KDD process is presented in many papers, but one of the most cited is CRISP-DM [2, 3] (***CR****oss* ***I****ndustry* ***S****tandard* ***P****rocess model for* ***D****ata* ***M****ining*). This reference model presents global view of the life cycle of a data mining project. In this paper I concentrate on the phase of data preparation of special data type – series variables. I try to analyze in more detail main tasks and goals in that problem area and the reference model, which is special case of data preparation phase for series data, is proposed.

Paper is divided into following chapters. Chapter 2 describes briefly reference model CRISP-DM. Chapter 3 includes the definition of series data type. In chapter 4 reference model for series data preparation is presented, and the last chapter 5 concludes the whole paper.

[1] Knowledge Discovery in Databases

2 CRISP-DM reference model

This chapter is a short outline of the CRISP-DM reference model. Figure 1 presents the phases of that model, together with relationships between these phases.

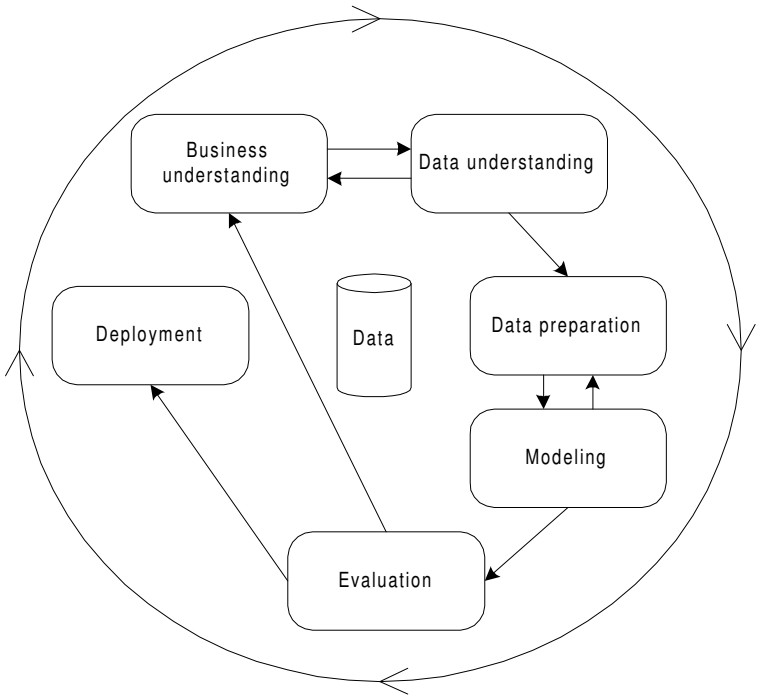

Fig. 1: Phases of the CRISP-DM reference model

The sequence of the phases is not rigid – it is determined by the outcome of each phase. In every phase a wide variety of analytical and modeling tools can be used. The outer cycle in figure 1 symbolize the iterative nature of the process itself. The process is not over once a solution is deployed – acquired results often bring new questions and the whole process is repeated (the subsequent iterations should benefit from previous results).

The rest of this chapter provides a brief description of each phase.

Business understanding
Understanding of the project requirements and objectives is the main task of this initial phase. Acquired knowledge is converted into problem definition and a preliminary plan, how to achieve the results, is designed.

Data understanding
This phase includes initial data collection and activities in order to get familiar with the data and their structure (date quality recognition, interesting subsets detection, ...). Summary statistics are often evaluated (exploratory data analysis) and visual techniques are very popular in this phase too.

Data preparation
The data preparation phase covers all activities to construct the final data set (or data sets) from the raw data. The final data set will be used in subsequent

modeling phase. This phase is the most time and energy consuming phase of the whole process.

Modeling
In this phase various analytical methods and modeling techniques are applied. Usually there are several techniques for the same data mining problem type, but they could have different requirements as for the form of the data. That is why returning to the phase of data preparation is often necessary.

Evaluation
The main task of this stage is evaluation and interpretation of the results from the modeling phase. Evaluation of the results is very important, because in this point we decide, if there are some interesting issues (from customers point of view).

Deployment
Creation of the model is generally not the end of the project. Deployment phase covers many final tasks – from report generation to information system construction (based on discovered knowledge).

In KDD process a few user types can be distinguished. Their role in the whole process is not minor, so I will mention them briefly.

Customer
Customer initializes the problem solution. Usually his expectations about results are not clear - he doesn't know exactly, what kind of results he wants. He wants the results to make his enterprise more efficient.

Domain expert
Person deputed by customer. He is well informed about problem area – his cooperation is necessary in all KDD phases.

Analyst
KDD expert. He is the leading person in the project.

End user
Consumer of the final product. Final product could be some text report, but even a software system based on discovered knowledge as well.

3 Series data type definition

Data preparation phase is sometimes underestimated. At the same time this phase absorbs significant amount of time and sources. There are many data preparation tasks for simple attribute types (real, nominal, ordinal, date, time, text) and the analyst role is to choose correct procedure. Even in the case of simple attribute types it is non-trivial task, and what about complex ones like images, series data, sounds, ...? The preparation of them is of course more difficult and that is why I decided to look in more detail at the series data preparation phase.

At first let's remind the definition of the series variable. In [6] we can find the definition of time-series, special case of series variables.

Definition 1. TIME-SERIES: data type where one attribute represents different moments of time; the records are ordered by the values of this attribute. For one value of time, other attributes store information about co-occurring properties of objects.

As we can see, in series data type we assume the implicit, independent variable (it's time for time series, but it could be for instance temperature, distance, pressure, ...) and one or more dependent variables (characteristics of given object). Since we can always construct series data type attribute with exactly one dependent variable, this case will be default for the rest of this paper. Now we can imagine each series variable as representation of function $x = f(t)$, where t is independent and x dependent variable.

Definition 2. Let $T = \{t_1, t_2, \ldots, t_n\}$, $t_i \in \mathcal{R}$, $t_i < t_j \; for \; \forall i < j$, where $\mathcal{R}$ is the real numbers set, i and $j \in \{1, 2, \ldots, n\}$. Series variable is real, ordinal or nominal values sequence $\{X(t_i), t_i \in T\}$. The length of the series variable is the cardinality of T, $card(T) = N$.

4 Series data preparation reference model

Figure 2 presents individual phases of the proposed series data preparation reference model. Sequence of the phases is not rigid (similarly to superior model) and the process is iterative again.

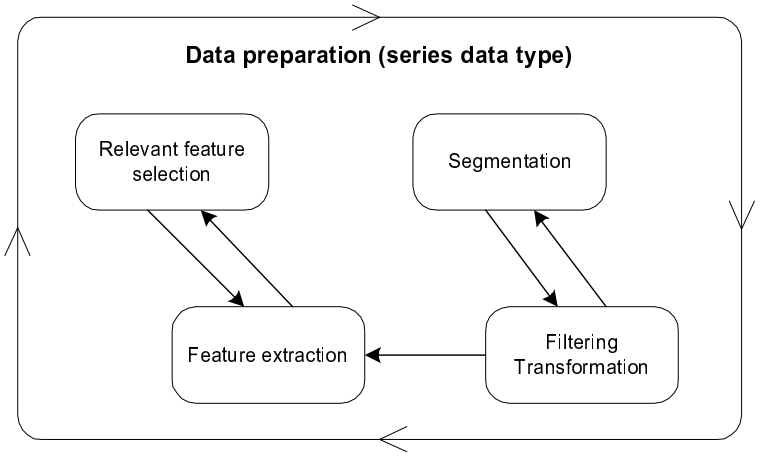

Fig. 2: Phases of the series data preparation reference model

4.1 Segmentation

Task

In some cases the whole series is not suitable for immediately processing. If a series is for example periodic, we can divide it into individual segments and use

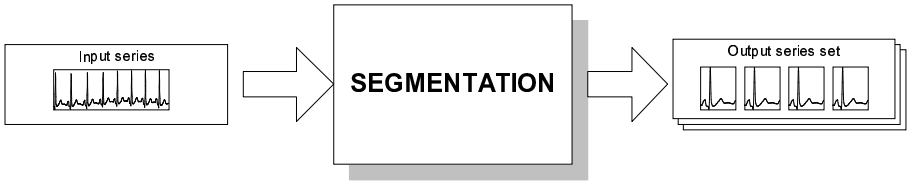

Fig. 3: Series segmentation

the result set of series for further processing (next preparation step or analyzing). This advance is often required in bio-signal processing (EKG, EEG, ...). Another reason for series segmentation could be the fact, that not all parts of series are interesting – we may want to remove some part from the entire series variable (fig. 4).

Output

Set of entire series parts (segments), every represented by new series variable. Not all of detected parts have to be included in the result set.

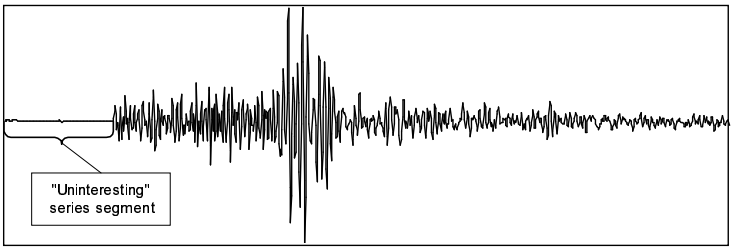

Fig. 4: Uninteresting segment removing

4.2 Filtering, transformation

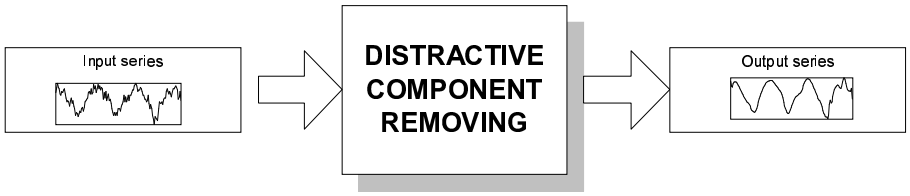

Fig. 5: Distractive component removing

Task

This phase includes a wide variety of series manipulation techniques. Generally we try in this phase to produce derived series variable, that is in some manner better than input series. What does it mean 'better'? One of the most common problems is distractive component removing (fig. 5). In every series data set we have to realize (usually after domain expert consultation), what is the distractive component in our data? In series various components can be detected

[11](trend, season, noise, ...) and we have to decide, which of the components is distractive. Usually it is noise, but it can be for instance linear trend as well.

Another technique used in this phase is normalization. There are two basic types of normalization. First of them, MIN-MAX normalization, transforms input series variable into output one, in which every dependent variable value is in the given range MIN-MAX. Second produces new series variables with common mean and standard deviation (usually zero mean and standard deviation equals to one is required).

Output

Derived series variable.

4.3 Feature extraction

Fig. 6: Series feature selection

Task

Feature extraction is the process of generating features to be used in further analysis. There are two fundamental approaches in this area: statistical and structural. Statistical approach, sometimes denoted as quantitative, includes for example simple descriptive statistics (mean, standard deviation, frequency count summarizations), Fourier transformations, wavelet transformations, ... Typical for this approach is the fact the quantitative features are organized into fixed-length feature vector. It means every series is described by the same number of features.

Quantitative approach is sometimes insufficient. There can be some morphological subpatterns in series and another approach – qualitative – is needed. Usually we try to find some structural features, called primitives, which represents subpatterns in the series. We have to store relationships among the primitives too. Feature vectors discovered by this approach contain variable number of elements, i.e. every series variable can be described by different number of features.

Both approaches may be also combined into hybrid one. It tries to suppress drawbacks of each approach, while conserving their advantages.

Output

Series feature set.

Fig. 7: Relevant feature selection

4.4 Relevant feature selection

Task

Main aim of this phase is to reduce the number of series features. We have to decide, which features are relevant for our purposes. Domain expert cooperation could be very useful in this phase, but some automated techniques for feature relevancy evaluation can be helpful as well. For example removing features with the same value for all series is desired – such features are useless in further analysis. Of course we can use more sophisticated approaches (Principal Component Analysis, feature entropy evaluation [9]).

Output

Reduced series feature set.

5 Conclusions

Data preparation phase of KDD (and series data preparation in particular) is very important, but the most time-consuming phase of the whole process. The best way, how to deal with this phase, is of course to automatize it. I don't think it is fully possible - there are too many parameters, that can be influential. Of course if we want to automatize (at least partially) some process, we have to construct its model. I propose in this paper the referential model of series data preparation – this model can guide us through the entire process.

Rereferences

[1] Vondrák,I.: *Umělá inteligence a neuronové sítě.* FEI, VŠB-TU Ostrava, 1998.

[2] Berka, P.: *Dobývání znalostí z databází.* Article in magazine Softwarové noviny nr.6/2001, p. 28-33.

[3] Pete Chapman, Julian Clinton, Thomas Khabaza, Thomas Reinartz, and Rdiger Wirth. *The CRISP-DM process model.* http://www.crisp-dm.org.

[4] John W. Sammon, Jr : *A Nonlinear Mapping for Data Structure Analysis.* IEEE Transactions on Computers, C-18(5):401-409, 1969.

[5] A. Ultsch, H.P. Siemon. *Kohonen's Self Organizing Feature Maps for Exploratory Data Analysis.* Proceedings of International Neural Network Conference (INNC'90), p. 305-308, Dordrecht, Netherlands, 1990.

[6] W. Klösgen, J. M. Żytkow: *Knowledge Discovery in Database Terminology.* Advances in Knowledge Discovery and Data Mining, p. 572-592, ISBN 0-262-56097-6, 1996.

[7] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth: *From Data Mining to Knowledge Discovery: An Overview.* Advances in Knowledge Discovery and Data Mining, p. 1-34, ISBN 0-262-56097-6, 1996.

[8] Olszewski,R.T.: *Generalized Feature Extraction for Structural Pattern Recognition in TimeSeries Data*, PhD Thesis, School of Computer Science, Carnegie Mellon, University Pittsburgh, February 2001.

[9] Y. Y. Yao, S. K. Michael Wong, Cory J. Butz: *On Information-Theoretic Measures of Attribute Importance.* PAKDD 1999: 133-137

[10] Lukasová, A., Šarmanová ,J.: *Metody shlukové analýzy.* SNTL - Nakladatelství technické literatury, Praha 1985. *On Information-Theoretic Measures of Attribute Importance.* PAKDD 1999: 133-137

[11] Pyle, D.:*Data Preparation for Data Mining.* Morgan Kaufmann Publishers, Inc., ISBN 1-55860-529-0, 1999.

DTD Representation in Lego Proof Assistant[1]

Michal Valenta

Dept. of Computer Science, CTU FEE
Karlovo náměstí 13, Praha 2, 121 35

e-mail: valenta@fel.cvut.cz

Abstract. Paper discuss representation of DTD document specifications in Lego proof development assistant. It is useful for development of mapping function (program) which transforms data from one DTD specification to another. Developed function represents also the proof of its correctness – this is the main advantage of our approach. Paper claims rather to sketch the technique than a comprehensive solution, hence it reduces the set of discussed DTD constructors. An example of DTD specification and development a mapping function is also provided here.

Key words: DTD, ECC, Lego, Type System, Document Transformation

1 Introduction

Suppose, we have two different data ontologies in the form of DTD, for example two different address book specifications. The task is to transfer data from one ontology to another. Transformation program is a function f of type $DTD_1 \longrightarrow DTD_2$. The construction of the transformation function is straightforward and clear if ontologies are simple or very similar to each other. But our "ad-hoc" constructed function becomes unclear and potentially wrong in the case of more complex ontologies. Although the mechanism of construction of transformation function is relatively easy, its checking of correctness is highly desired.

Generally, there are two different approaches how to support correctness of data transformation function:

1. Develop a transformation function and then proved that it is correct (ie. of the type $DTD_1 \longrightarrow DTD_2$.
2. Support directly development process of itself.

Lego interactive proof development assistant is a suitable tool for the second approach.The scheme of our work is very simple:

[1]This work is partialy supported by research program MSM 212300014 "Informační technologie a komunikace" CTU FEE Prague and by grant CTU 300109713

1. Construct representation of DTD_1 in Lego.
2. Construct representation of DTD_2 in Lego.
3. Start the proof development process with the goal f of type $DTD_1 \longrightarrow DTD_2$.

Successfully developed term f then represents both - the transformation function and the proof of its correctness.

We have to discuss consequently the topics of DTD representation in Lego and then the transformation function f development to meet our goal.

2 DTD Representation in Lego

Lego proof development assistant can support several different type systems (LF, CC, ECC, ...). There is not enough place to describe their specifics, rather we can refer to [Luo94] and [LuPo] for details. This paper suppose using of ECC (Extended Calculus of Constructions).

There are two different universes in ECC - *Prop* and *Type*. *Prop* is meant as a universe for representation of logic (propositions) such are reflexivity, transitivity, equality, logic operations etc. *Type* is used for (complex) data representations. Hence the *Type* universe is our basic (principal) type for DTD representation.

2.1 Supported DTD constructs

The aim of this paper is to explain possible approach to XML data transformations in Lego. It doesn't claim to supply a complete solution for all DTD constructs, rather we limit the set of supported DTD constructs here.

Suppose our DTD specifications consist only from elements without any attributes. The simplest elements are of type $PCDATA$. New elements can be derived from existing ones by sequence $(E_1, E_2, \ldots, E_n)$. We also allow to specify multiple occurrence of element $(E+, E*)$, zero or one occurrence $(E?)$ and alternative occurrence $(E_1 \mid E_2)$.

An example of considered DTD specifications for our discussion can be a DTD of address book on Figure 1.

2.2 Representation of Supported DTD Constructs in Lego

ECC provides two different universes as it was mentioned above - *Type* and *Prop*. *Type* is intended for data representation, so we can start with this universe. The simplest type of DTD element is $PCDATA$, then let *pcddata* to be a type in universe *Type*:

```
pcdata:Type
```

```
<!DOCTYPE ...
<! ELEMENT adrbook1(title*, full_name, address+, phone?, email|ICQ)>
<! ELEMENT full_name(name, nickname?, surname)>
<! ELEMENT address (street?, no, place, country, zip?>
<! ELEMENT name (#PCDATA)>
<! ELEMENT nickname (#PCDATA)>
<! ELEMENT surname (#PCDATA)>
<! ELEMENT street (#PCDATA)>
<! ELEMENT no (#PCDATA)>
<! ELEMENT place (#PCDATA)>
<! ELEMENT country (#PCDATA)>
<! ELEMENT zip (#PCDATA)>
<! ELEMENT title (#PCDATA)>
<! ELEMENT phone (#PCDATA)>
<! ELEMENT email (#PCDATA)>
<! ELEMENT ICQ (#PCDATA)>
```

Fig. 1: DTD specification of address book.

Lego provides a library with definitions of commonly used types and operations like $bool, set, list, vector, nat$ etc. Their syntax and semantics is done in a "source" lego files, so user can eventually change their behaviors if necessary.

We can used some of these ready made types for our reasons. Representation of DTD type construct is inspired by [Po00], but it is partially changed to meet advantages of ECC and Lego.

2.2.1 Alternative Occurrence of Elements – $E_1 \mid E_2$

This construction is modeled by a union type. Element $E_1 \mid E_2$ is treated as $E_1 \cup E_2$.

Lego library provides a polymorphic union type which is instantiated by the types of its components. Library provides also constructors $in1$ and $in2$ and a recursion rule for sum type construction. We'll use these rules later in construction of mapping function f from DTD_1 to DTD_2.

2.2.2 Zero or One Occurrence of Element – $E?$

There is empty type definition in Lego library. It can be used in combination with union type to model constructor $E?$. We have to define an empty element:

```
empty_el:empty
```

for further needs of reasoning over DTD definitions.

2.2.3 One or More Occurrences of Element – $E+$

This construction is modeled as a function of type $E \longrightarrow BOOL$ in [Po00]. But what we really need is a function which accepts argument of appropriate type (ie. of type E here). We'll define new type in Lego to represent this construct.

This new type should be a polymorphic type which is instantiated by the type t. It can be defined as an inductive type (named `reg`) with only one constructor (named `do_reg`) which is of type `t -> reg`. A general constructor for inductive types is provided for these reasons in Lego. It has pretty simplified syntax and it also automatically generates elimination rule and constructors for this type. Definition of type reg is as follows:

```
Inductive [reg : Type] Parameters [t ? Type]
ElimOver Type Constructors [do_reg : t -> reg];
Discharge t;
```

See [LuPo] and [Luo94] for more details about inductive type definitions. Constructor `do_reg` will be useful for reasoning over mapping function $f : DTD_1 \longrightarrow DTD_2$ which is in our focus.

We also need a reduction function for this type. It is a function which takes a type `reg t` and returns type `t`. This function is named `undo_reg`. For its definition basic elimination rule for type `reg` is used. Elimination rule is relate to inductive data types. We can refer the reader to [LuPo] and [Luo94] for more details. A little bit more to this topic can also be found in section 3.1 in this paper. The definition of function `undo_reg` is as follows:

```
[undo_reg = [t|Type](reg_elim([_:reg t]t) ([f:t]f))];
```

2.2.4 Zero, One or More Occurrences of Element – $E*$

Representation of this construct is done using above described techniques. $E*$ can also be written as $(E+) \cup \perp$, written in Lego:

```
sum (reg E) empty
```

2.2.5 Sequence of Elements – $(E_1, E_2, \ldots, E_n)$

Here we can again used a ready made Lego constructors. Used type – `Record` belongs directly to Lego, so it doesn't need any library. This type enables to create named record structure. It also automatically derives rules for appropriate constructor and reductions.

Suppose we have an element E defined as sequence of elements e_1 and e_2, which are of type E_1 and E_2 respectively. We can write in Lego:

```
Record [E : Type] Fields [e1 : E1] [e2 : E2];
```

```
Module lib_dtd;
Make lib_union;
Make lib_empty;
Inductive [reg : Type] Parameters [t ? Type]
ElimOver Type Constructors [do_reg : t -> reg];
Discharge t;
[undo_reg = [t|Type](reg_elim([_:reg t]t) ([f:t]f))];
[pcdata : Type];
[empty_el : Type];
```

Fig. 2: Lego module lib_dtd.l for DTD specification

Lego defines new type `E`, its constructor (named `make_E`) and two reduction rules `e1` and `e2` of types `E->E1` and `E->E2` respectively. Again all these rules are necessary for reasoning over DTD definitions.

2.2.6 Review of Representation of DTD specification in Lego

Let us now shortly review our representation of DTD documents in Lego. Remember, we are using Lego library for types `sum` and `empty` and we have defined new type `reg` in previous subsections.

These considerations can be expressed in Lego module (file). This file is then imported into Lego system before we start with definition of DTD specifications. File is named `lib_dtd.l` and is defined in Figure 2.

Now we can describe individual DTD construct representation in Lego. Suppose elements $a : A$ and $b : B$, where A and B are either *pcdata* or types defined using DTD data constructors ($?, +, *, \cup$ or ,). Description is done on Figure 3.

Description	**DTD**	**Lego**
simple element	$a(\#PCDATA)$	`[a : pcdata]`
zero or one occ.	$a?$	`sum A empty`
one or more occ.	$a+$	`reg A`
zero or more occ.	$a*$	`sum (reg A) empty`
alternative	$a \mid b$	`sum A B`
sequence	(a, b)	`Record [C : Type]` `Fields [a : A] [b : B]`

Fig. 3: Representation of DTD Type Constructors in Lego

2.3 Example of DTD Specification in Lego

Now, we are ready to apply above discussed DTD type constructors and provide an example of DTD representation in Lego. We choose address book specification. Its DTD is done on Figure 1 on page 104.

```
Make lib_dtd;

Record [ADDRESS_1:Type]
Fields [street_1: sum pcdata empty]
       [no_1, place_1, country_1 : pcdata]
       [zip_1 : sum pcdata empty];

Record [FULL_NAME_1:Type]
Fields [name_1 : pcdata]
       [nickname_1 : sum pcdata empty]
       [surname_1 : pcdata];

Record [ADDRBOOK_1 :Type]
Fields [title_1 : sum (reg pcdata) empty]
       [full_name_1 : FULL_NAME_1]
       [address_1 : reg ADDRESS_1]
       [phone sum pcdata empty]
       [email_ICQ_1 : sum pcdata pcdata];
```

Fig. 4: Lego definition of address book

Remark 1: There is one important difference between Lego and DTD definitions. It lies in the definition of sequence. One have to name the record type defined in Lego if it is used later in definition as a component. But the name of the type should differ from the name of the element of that type. We can adopt a name convention such that the names of record types are written in capital and the names of element are written in small letters.

Remark 2: We add a suffix `_1` to every name of element and the type of above mentioned definition. It is due to avoid name ambiguities of `addressbook_1` defined above and the other DTD specification. We can suppose ambiguities in element names if we are working with two similar ontologies.

It is clear now, that transformation from DTD specification to Lego definition is straightforward. It should be done automatically employing standard algorithms known from compiler constructions. The same assertion holds also for transformations from Lego to DTD.

3 Working with DTD in Lego

We are able to specify DTD_1 and DTD_2 now. Let us discuss the next topic – creation of a transformation function $f : DTD_1 \longrightarrow DTD_2$ in this section. We have worked in Lego definition state yet, but we have to switch into proof state now. It is done through the specification of the goal:

```
Goal f : DTD1 -> DTD2;
```

3.1 Lego Proof state

Let us shortly describe how Lego works in the proof state. The main idea is to manipulate a context. Context means an ordered set of definitions and their types. There are several rules which defines how derivations - ie. changes of context are done. This set of rules is called type system (ECC in our case).

Usually we start with empty context and add a new definitions - DTD_1 and DTD_2 in our case. Remember, that usually many functions (rules) are defined with one type definition (mainly constructors and eliminators).

By specifying the goal: `Goal f : DTD1 -> DTD2;` Lego generates new (existential) variable named `?0` which is of the same type as the goal. To create a proof means to construct a term which is of the goal type. We can use definitions in context and several Lego commands – mainly `Intros` and `Refine`.

Intros: We can change the current goal by `Intros` if it is in the form `?0 : A->B`. It creates new hypothesis (for example term `h`) of type `A` and changes the current goal to `?1 : B`.

Refine: It is the most powerful command for proof generation. It takes a term like its argument and tries to unify the type of this term with the type of current goal. It can succeed then the goal is proved by this term. If it doesn't succeed and the refinement term `r` is of type `{x:A}B x` (`{x:A}B` means a functional type `A->B` where `x` is bind in term `B`, `{x:A}B x` means application of term `x` to term `{x:A\}B`), Lego then suppose that any special instance of `x` can be a proof of the goal. It generates a new goal `?n+1 : A`, specializes the refinement term `r` to `r ?n+1` which is of type `B ?n+1` and starts the new refinement with this specialized term. It may work in a cycle so several new specializations and new partial goals can be generated by one application of `Refine` command until the last generated specialization of refinement term doesn't unify with the specialized goal. Otherwise refinement process fails.

Let us now provide a small example. Suppose two very simple e-mail address book specifications:

$$e_1(name, surname, nickname, email+)$$

$$e_2(name, nickname*, surname, email)$$

All internal elements (in the brackets) are of type $PCDATA$ for simplicity. We want to create a function $f : e_1 \longrightarrow e_2$. Lego specification of e_1 and e_2 are as follows:

```
Record [E1:Type]
Fields [name1, surname1, nickname1 : pcdata]
       [email1 reg pcdata];

Record [E2:Type]
Fields [name2 : pcdata]
       [nickname2 : sum (reg pcdata) empty]
       [surname2 : pcdata]
       [email2 : pcdata];
```

Let us start with the transformation function development:

```
 Lego> Goal f : E1 -> E2;
Goal f
   ?0 : E1->E2
Lego> Intros h;
Intros (1) h
  h : E1
   ?1 : E2
Lego> Refine make_E2;
Refine by  make_E2
   ?2 : pcdata
   ?3 : sum (reg pcdata) empty
   ?4 : pcdata
   ?5 : pcdata
```

We started with a goal specification, Lego accepts it and generates a goal `?0`. We can apply `Intros h`, because the current goal is of type `E1 -> E2`. Lego generates hypothesis `h : E1`. It is exactly what we need – `h` is now in context and we can used it to satisfy goal `?1`.

We know, that E2 consists of a components. It is easier to work with individual components for further work, so we are trying to decompose current goal `?1`. It can be done by refinement using E2 constructor. It was generated automatically during definition of E2 type. Its name is `make_E2`. We can explore this term in Lego to see its value and type:

```
Lego> make_E2;
value = make_E2
type  = pcdata->(sum (reg pcdata) empty)->pcdata->pcdata->E2
```

The refinement by this term doesn't succeed at the first level, but it tries to specialize the refinement term by assumptions of its the leftmost parts. It generates four partial goals before unification succeed. These partial goals are identical with `E2` components.

Now we can follows with satisfying these particular goals. We have hypotheses `h : E1`. Some reduction rules were generated during the phase of this record definition. These rules have the same names as record components. We can explore their types and values:

```
Lego> name1;
value of name1 =
  E1_elim ([_:E1]pcdata) ([name1,_,_:pcdata][_:reg pcdata]name1)
type  of name1 = E1->pcdata
Lego> surname1;
value of surname1 =
  E1_elim ([_:E1]pcdata) ([_,surname1,_:pcdata][_:reg pcdata]surname1)
type  of surname1 = E1->pcdata
Lego> nickname1;
value of nickname1 =
  E1_elim ([_:E1]pcdata) ([_,_,nickname1:pcdata][_:reg pcdata]nickname1)
type  of nickname1 = E1->pcdata
Lego> email1;
value of email1 =
  E1_elim ([_:E1]reg pcdata) ([_,_,_:pcdata][email1:reg pcdata]email1)
type  of email1 = E1->reg pcdata
```

All are based on `E1_elim` which is the elimination rule for the type `E1`. Its type is:

```
type  =
  {C_E1:E1->TYPE}
  ({name1,surname1,nickname1:pcdata}{email1:reg pcdata}
   C_E1 (make_E1 name1 surname1 nickname1 email1))->{z:E1}C_E1 z
```

This elimination rule is also defined automatically during the definition of record `E1`. It is based on general principle of inductive data types, which is embedded in Lego system. See [Luo94], [LuPo] for details. For better understanding we can say that a computation rules belong to every elimination rule. For our case of `E1` computation rule is only one and looks like follows:

```
 [[C_E1:E1->TYPE]
 [f_make_E1:{name1,surname1,nickname1:pcdata}{email1:reg pcdata}
  C_E1 (make_E1 name1 surname1 nickname1 email1)]
 [name1,surname1,nickname1:pcdata][email1:reg pcdata]
    E1_elim C_E1 f_make_E1 (make_E1 name1 surname1 nickname1 email1)  ==>
    f_make_E1 name1 surname1 nickname1 email1]
```

The first three lines can be understood as a local declarations and the last three lines defines a computation. It can be understood like this - if appears the terms like it is written on the lines four and five, they are replaced by the sixth line. Lego also automatically generates a reduction rules using this eliminator this reduction rules are identical with components and are showed above.

Let us now continue with our example. We decomposed current goal and we are going to satisfy partial goals by using hypothesis `h`. We also have a reduction rules, which allows to express individual components from hypotheses `h`.

```
Lego> Prf;
  h : E1
   ?2 : pcdata
   ?3 : sum (reg pcdata) empty
   ?4 : pcdata
   ?5 : pcdata
Lego> Refine name1 h;                        (* name2 *)
Refine by  name1 h
   ?3 : sum (reg pcdata) empty
   ?4 : pcdata
   ?5 : pcdata
Lego> Refine in1 (do_reg (nickname1 h));  (* nickname2 *)
Refine by  in1 (do_reg (nickname1 h))
   ?4 : pcdata
   ?5 : pcdata
Lego> Refine surname1 h;                     (* surname2 *)
Refine by  surname1 h
   ?5 : pcdata
Lego> Refine undo_reg (email1 h);            (* email2 *)
Refine by  undo_reg (email1 h)
Discharge..  h
*** QED ***   (* time= 0.190000  gc= 0.0  sys= 0.020000 *)
```

We started with command `Prf` to review hypothesis and all unsatisfied goals. Satisfying the goals `?2` and `?4` is trivial. Components `name` and `surname` are in both specifications (`e1` and `e2`) of type `pcdata` so we have only to map them to each other. It is done be applying reduction rules `name1` and `surname1` to hypothesis `h`.

Let us explain the case of `nickname` mapping. We should consider that it is of type `pcdata` in specification `e1` and of type `sum (reg pcdata)empty` in specification `e2`. What we have to do is to construct a term of type `sum (reg pcdata)empty` from the term of type `pcdata`. It is just done by the term `in1 (do_reg (nickname1 h))`. Where `in1` is a constructor of sum type. It suppose that its argument belongs to the first set in the sum type, it takes this arguments and returns a sum type. Constructor `do_reg` is used for creation of type `reg pcdata` from the type `pcdata`.

The last goal (`email2`) seems to be trivial because it is of type `pcdata`. But we want to satisfy it using `email1` which is of type `reg pcdata`. To achieve our goal we can use function `undo_reg` which suppose argument of type `reg t` and returns type `t`. Refining by term `undo_reg (email1 h)` we have got the term of type `pcdata` which is unified with the last goal.

Now, all goals were satisfied, so we have finished. Lego finds this situation and signalize it by printing `*** QED ***`. We can now save our proof as a term and explore its value and type:

```
 Lego> Save;
"f"  saved as global, unfrozen
Lego> f;
```

```
value of f =
  [h:E1]
  make_E2 (name1 h) (in1 (do_reg (nickname1 h))) (surname1 h)
   (undo_reg (email1 h))
type  of f = E1->E2
```

Function `f` represents both – the algorithm of one possible data mapping function from specification `e1` to `e2` and the proof of its correctness. This function can be saved and its value can be used later:

```
Lego> Goal g: E1->E2;
Goal g
   ?0 : E1->E2
Lego> Intros h;
Intros (1) h
  h : E1
   ?1 : E2
Lego> Refine f h;
Refine by  f h
Discharge..  h
*** QED ***   (* time= 0.0  gc= 0.0  sys= 0.0 *)
```

3.2 Summary of DTD Reasoning Techniques in Lego

Now, we have a notion how the mapping function `f` was developed in a concrete example. It could be useful to summarize this techniques for every situation which can occur in `f : DTD1->DTD2` development process:

1. **Specification of the goal.** This step is every time the same:

   ```
   Goal f : DTD1 -> DTD2;
   ```

   It only differs in the name of function and the names of dtd specifications.

2. **Creating a hypothesis of type `DTD1`.** Our proof development mechanism suppose that we want to map specification `DTD1` to specification `DTD2` - it is done by command

   ```
   Intros h;
   ```

3. **Decomposition of `DTD2` specification.** This step is again standard, because we are starting with DTD specification. Each DTD specification consist from at least one element and hence it is implemented as record. Appropriate command is

   ```
   Refine make_DTD2;
   ```

Term `make_DTD2` is generated automatically with the definition of record `DTD2`.

4. **Satisfying partial goals.**
   - We have got a number of partial goals as a result of previous step.
   - Now we have to map individual components of hypothesis `h:DTD1` to appropriate partial goals.
   - Individual components can be obtain by applying hypotheses `h` to reduction term (of the same name as components).
   - Also we have to respect other DTD constructors $(?, +, *, |)$. All these constructors are in Lego represented by combinations of `reg`, `sum` and `empty`. We should be able to derive in both directions (from sum to its components and from components to sum for example). All possible useful terms for refinement are summarized in table on Figure 5.
   - In the case that components of `DTD1` or `DTD2` are records, we can use constructor `make_Component_name` or `Component_item_name` reduction rules and work recusively from step 4 of this algorithm.
5. **Saving finished proof.** This is done by command `Save;`

DTD	construction	reduction
$\|$	`in1 |A|B a : sum A B` `in1 |A|B b : sum A B`	`case([_:A]a) ([_:B]a) ab : A` `case([_:A]b) ([_:B]b) ab : B`
$+$	`do_reg:A -> reg A`	`undo_reg:reg A -> A`
$*, ?$	the same like $\|$ and $+$ with `emtpy_el:empty`	the same like $\|$ and $+$ with `emtpy_el:empty`
Record R Fields a:A, b:B	`make_R:A->B->R`	`a: R -> A` `b: R -> B`

Fig. 5: Useful terms for DTD reasoning in Lego

4 Conclusions

We have presented a way how to represent DTD specifications in Lego proof development assistant as like as the mechanism how to develop a mapping function f of type $DTD_1 \longrightarrow DTD_2$. This mapping function represents both the transformation mechanism and the proof of its correctness. This feature is probably the most valid argument why to follow this line of research.

Although our discussion were reduced only to limited set of DTD constructors (we didn't discuss attributes of DTD elements), some suggested implementations of DTD constructs are not very pretty for applying (especially `sum` type used for implementation of $|$ and $*$). It is considered topic for future work.

The future work is also intended to extend the set of DTD constructs by attributes of elements. Also the possibility of switch from DTD specifications to XML schemes seems as possible.

Rereferences

[Luo94] Z. Luo.: Computation and Reasoning. A Type Theory for Computer Science. Clarendon Press. Oxford 1994.

[LuPo] Z. Luo, R. Pollack.: LEGO Proof Development System: User's Manual. Dept. of Computer Science, University of Edinburgh 1992.
** http://www.dcs.ed.ac.uk/home/lego*

[Po00] J. Pokorný.: XML Functionality. Proceedings of IDEAS2000, B.C.Desai, Y. Kioki, M. Toyama (Eds.),
IEEE Comp. Society, 2000. pp. 266-274.
** http://195.113.17.94/texty/pokorny.ps/ideas2000a.ps*

Editor:	Michal Krátký
Department:	Department of Computer Science
Title:	DATESO '02
Place, year, edition:	Ostrava, 2002, 1.
Page count:	114
Edit:	VŠB-Technical University Of Ostrava Ostrava-Poruba, tř. 17. listopadu 15, 708 33
Print:	Repronis Ostrava, Nádražní 53
Edition:	100

Unsaleable

ISBN 80-248-0080-2

ISBN 80-248-0080-2