Computer Science Journal of Moldova, vol.**20**, no.3(60), 2012

Toward the Soundness of Sense Structure Definitions in Thesaurus-Dictionaries. Parsing Problems and Solutions*

Neculai Curteanu, Alex Moruz

Abstract

In this paper we point out some difficult problems of thesaurus-dictionary entry parsing, relying on the parsing technology of SCD (Segmentation-Cohesion-Dependency) configurations, successfully applied on six largest thesauri – Romanian (2), French, German (2), and Russian. **Challenging Problems: (a)** Intricate and / or recursive structures of the lexicographic segments met in the entries of certain thesauri; **(b)** Cyclicity (recursive) calls of some sense marker classes on marker sequences; **(c)** Establishing the hypergraph-driven dependencies between all the atomic and non-atomic sense definitions. Classical approach to solve these parsing problems is hard mainly because of depth-first search of sense definitions and markers, the substantial complexity of entries, and the sense tree dynamic construction embodied within these parsers. **SCD-based Parsing Solutions: (a)** The SCD parsing method is a procedural tool, completely formal grammar-free, handling the recursive structure of the lexicographic segments by procedural non-recursive calls performed on the SCD parsing configurations of the entry structure. **(b)** For dealing with cyclicity (recursive) calls between secondary sense markers and the sense enumeration markers, we proposed the Enumeration Closing Condition, sometimes coupled with New_Paragraphs typographic markers

* This paper is dedicated to Prof. Svetlana Cojocaru, IMI Director, as a tribute to her high professionalism, genuine friendship, passion and devotion to the special guild of researchers. The authors, with gratitude and best wishes for her sixtieth anniversary!

transformed into numeral sense enumeration. **(c)** These problems, their lexicographic modeling and parsing solutions are addressed to both dictionary parser programmers to experience the SCD-based parsing method, as well as to lexicographers and thesauri designers for tailoring balanced lexical-semantics granularities and sounder sense tree definitions of the dictionary entries.

Keywords: dictionary entry parsing; parsing method of SCD configurations; recursive lexicographic segments; recursive calls of sense markers; Enumeration Closing Condition; soundness of sense structure definitions.

1 Thesaurus-Dictionary Parsing with SCD Configurations

This section goal is two-fold: to briefly introduce the *parsing method* of SCD (Segmentation-Cohesion-Dependency) *configurations*, which was applied to parse six largest Romanian, French, German, and Russian dictionaries [7], [4], [3], [5], [6], and to outline the issue of the present paper.

The *parsing method* of SCD *configurations* consists in applying *breadth-first* (completed with depth-first, stack-type) searching algorithms for the recognition and establishing the dependencies between the sense marker classes of dictionary entries [4], [3], [5], [6], [7]. In general, an SCD *configuration* (hereafter, SCD*config*) has the following computational components: • A set of *marker classes*: a *marker* is a boundary for a specific linguistic category; • A *hypergraph-like hierarchy* that pre-establishes the dependencies among the marker classes; • A *searching* (*parsing*) *algorithm*.

When applied to dictionary entry parsing, the method of SCD configurations merges the following sequence of (at least) three specific configurations (*i.e.* lexical-semantics sense levels): **(a)** The *first one*, abbreviated hereafter SCD*config*1, performs the segmentation and dependencies for the *lexicographic segments* [11 :2], [10] of each dictionary entry [4], [5], [7]. **(b)** Stepping down into the lexicographic segments of a thesaurus-dictionary entry, the *second* SCD *configuration* (SCD-

*config*2) usually parses the *sense description* segment, extracting its *sense tree* structure [4], [3], [5], [7]. Actually, the SCD*config*2 parses the entry sense definitions of larger lexical-semantics granularity in the sense description segment: primary, secondary, and literal / numeral enumeration senses. **(c)** The *third* SCD *configuration* (henceforth SCD*config*3) continues to refine the sense definitions of SCD*config*2, parsing each node in the generated sense-tree for obtaining the atomic definitions / senses (*i.e.* finest-grained meanings) of the dictionary entry.

We experienced the method of SCD *configurations* for *modeling* and *parsing*, with outstanding results (over 90% accuracy), on *six* largest, complex, and sensibly different thesaurus-dictionaries for *Romanian*: **DLR** (The Romanian Thesaurus – new format) [3], [4], [7], and **DAR** (The Romanian Thesaurus – old format) [4], [7], [16]; for *French*: **TLF** (Le Trésor de la Langue Française) [4], [7], [12]; for *German*: **DWB** (Deutsches Wörterbuch – GRIMM) [4], [7], [8], and **GWB** (Göthe-Wörterbuch) [4], [7], [8]; and for *Russian* – **DMLRL** (Dictionary of the Modern Literary Russian Language) [5], [6], [7].

The paper is organized as follows: *Section* 2 discusses the problems met in SCD*config*1 for recognizing the intricate or recursive structure of the *lexicographic segments* in German **DWB**, Romanian **DAR**, and French **TLF** thesauri. *Section* 3 examines situations of cyclicity (recursive) calls that may occur between secondary sense markers and sense enumeration(s) in **DAR**, **DMLRL**, and **DLR**, the transformation of the typographic *New_Paragraphs* into sense enumeration markers (*e.g.* in **DLR**, **DAR**, and **DMLRL**), and the solution provided by the *Enumeration Closing Condition* when recursive calls occur [5], [6], [4], [7]. *Section* 4 points out few examples of (atomic) definition parsing problems in **DLR**, **TLF**, and **DMLRL** [5], [4], [7]. *Section* 5 outlines the impact of the discussed parsing problems and solutions on both the robust parser construction and the soundness of lexicographic design for the largest thesaurus-dictionaries, obtained within the optimal and portable framework of SCD configurations.

2 Parsing the Lexicographic Segments on SCD-*Config*1

2.1 Intricate Lexicographic Segments in German DWB

The German **DWB** (Deutsches Wörterbuch – GRIMM) entries comprise a complex structure of the lexicographic segments, which provide a non-uniform and non-unitary composition [8]. A special feature is that **DWB** (Deutsches Wörterbuch) and **GWB** (Göthe-Wörterbuch) [8] lexicographic segments are composed of two parts: a first (optional) *root-sense* subsegment, and the *body* subsegment, which contains the explicit sense markers, easily recognizable. For **DWB**, the parsing of lexicographic segments is not at all a comfortable task since they are defined by three distinct means, displaying a rather intricate structure:

(A) *After* the *root-sense* of a **DWB** entry, or *after* the *root-sense* of a lexicographic segment, (a list of) italicized-and-spaced key-words are placed to constitute the *label* of the lexicographic segment *that follows*. Samples of such key-word labels for **DWB** lexicographic segments are: "*Form, Ausbildung und Ursprung*", "*Formen*", "*Ableitungen*", "*Verwandtschaft*","*Verwandtschaft und Form*", "*Formelles und Etymologisches*", "*Gebrauch*", "*Herkunft*", "*Grammatisches*", etc., or, for **DWB** (most important) sense-description segment: "*Bedeutung und Gebrauch*" (or just "*Bedeutung*"). In the example below, they are marked in 25% grey.

Example 2.1.1. GRUND, *m.*, *dialektisch auch f. gemeingerm. wort; fraglich ist das geschlecht von got.* *grundus *in* grunduwaddjus, *vgl.* afgrundiþa; *sonst meist masc.: ahd.* grunt, crunt; *mhd.* grunt; *as.* grund; *mnd.* grunt *meist f., selten m.; mnl.* gront *meist m., selten f.; ndl.* grond; *afries.* grund, grond; *ofries.* grund; *wfries.* groun, grùwn; *ags.* grund; *engl.* ground; *anord.* grunnr *m.*, grund *f.; dän.* grund *comm. gen.; schwed.* grund; *als dem german. entlehnt gelten lit.* gruntas *m., preusz.* gruntan acc. m., grunte *f., lett.* grunts *m.*, grunte *f., poln. russ. slov. nlaus.* grunt *m.* *ſ o r m u n d h e r k u n ft*.

1) *für das verständnis der vorgeschichte des wortes ist die z w i e g e s c h l e c h t i g k e i t*

... ...

H. V. SACHSENHEIM *spiegel* 177, 30;

die neuen grundt zu der kirchen *zimm. chron.*^{2} 2, 539, 36; du findest noch vil gar alter meür und grunt und thürn SIGMUND MEISTERLIN *in städtechron.* 3, 51, 14. *auszerschwäb. im obd. nur selten:* mosige grunde SEBIZ *feldbau* (1579) 149. *anders, als rein graphische erscheinung versteht sich das fehlen des umlautzeichens in md. texten; häufig z. b. bei* LUTHER: grebt die grunde 1, 148; drey starcke grund 6, 290. b e d e u t u n g. *die bedeutungsgeschichte des wortes läszt sich schwer aufbauen, weil ihre wesentlichsten etappen in vorgeschichtliche zeit fallen. die auch auszerdeutsch altbezeugten verwendungen im sinne von 'tiefe'* (*s. u.* I) *und im sinne ron 'erde'* (II) *stellen offenbar die beiden cardinalen bedeutungsstränge dar. aber auch die bedeutung 'tal-, wiesengrund'* (III), *anscheinend auf der*

... ...

... ... *..hat* (*s. u.* II A 1 a). *nach* JAC. GRIMM *liegt der unterschied darin, 'dasz* gr. *mehr nach innen geht,* boden *die oberfläche bezeichnet'* (*th.* 2, 211). *das trifft mehrfach zu; doch erschöpft diese unterscheidung einer mehr räumlichen und mehr flächenhaften vorstellung die sache nicht.*

I. grund *bezeichnet die feste untere begrenzung eines dinges.*

A. *grund von gewässern; seit ältester zeit belegbar: profundum* (*sc. mare*) crunt *ahd. gl.* 1, 232, 18; latid thea odra (*fisch*) eft an gr. faran *Hel.* 2633.

1) *am häufigsten vom meer* (*in übereinstimmung mit dem anord. gebrauch*):

... ...

In Ex. 2.1.1 above, these notions are illustrated as follows: between the entry lemma **GRUND** and the label "ſ o r m u n d h e r k u n ſ t", it spans the *root-sense subsegment* of the first lexicographic segment for the entry "**GRUND**". The key-words "ſ o r m u n d h e r k u n ſ t" represent the first label for the *first segment* of the lemma, described with several sense markers, among which the first one is "**1)**". The segment "ſ o r m u n d h e r k u n ſ t" ends when the label "b e d e u t u n g" occurs for the next lexicographic segment. Between this label and the

effective description of the segment senses, which begins with the sense markers "**I.**" ... "**A.**" ... etc., it spans the *root-sense* of the segment labeled with "*b e d e u t u n g*". Thus each lexicographic segment in **DWB** may contain, optionally, in a "preamble", the root-sense (subsegment) description of that segment. The key-words (or a list of key-words) placed at the end of a segment correspond to (and represent) the label of the lexicographic segment *that follows*.

(B) *The second* way to specify the lexicographic segments in **DWB** is expressed as follows: *after* the primary sense markers, there are specified those key-words representing *the label* of the lexicographic segment that follows. The example 2.1.2 is enlightening:

Example 2.1.2. GEBEN, *dare.*

I. *Formen, ableitungen, verwandtschaft.*

1) *es ist ein allgemein, aber ausschlieszlich germanisches wort: goth.* giban (*praet.* gaf), *ahd.*

... ...

II. *Bedeutung und gebrauch.*

1) geben *und* nehmen, *die beiden sich ergänzenden gegenstücke, verdienen die erste...*

... ...

The entry **GEBEN** of **DWB** has the *Latin definition* "*dare*", which is at the same time the root-sense of the entry. The first segment (which begins with the marker "**I.**") is labeled with "*Formen, ableitungen, verwandtschaft*", while the the second segment (which begins with the marker "**II.**") has the label "*Bedeutung und gebrauch*". This is the proper *sense description segment* of the lemma **GEBEN** from **DWB**, actually.

(C) The *third* (and most frequent) way to identify the lexical description segment(s) of a **DWB** entry is simply the lack of a segment label at the beginning of the sense description segment. By default, after the entry *root-sense segment* (which can be reduced to the Latin definition, *i.e.* the translation of the German word-lemma), the sense-description segment comes without any "*Bedeutung*" label, introducing explicit sense markers and definitions.

Example 2.1.3. BESUCHEN, *ahd.* pisuochan (GRAFF 6, 84),

mhd. besuochen, *nnl.* bezoeken, *schw.* besöka, *dän.* besöge.

1) *den jägern,* das wild besuchen, *aufspüren.*

2) einen ort besuchen, *mhd.* einen turnei besuochen. *Engelh.* 2359; *nhd.* die kirchen, spielhäuser, theater besuchen, *franz.* fréquenter; das sie dein haus und deiner unterthanen ...

... ...

While the lexicographic segment structure is not easy to be obtained for **DWB** (SCD-*config*1), as shown in this subsection, the dependency hypergraph for the sense description segment (SCD*config*2), represented in [4 :Fig. 6], looks more feasible when the former task has been achieved.

2.2 Recursive Structure of Lexicographic Segments in DAR

We present here the recursive configuration for two lexicographic segments in **DAR** (the old format of **DLR**): the *French* and *Nest* segments.

The *French* segment [4], [7] "looks" like the *sense description* segment, while the *Nest* (Romanian "*cuib*") segment delivers, at smaller dimensions, a similar (thus *recursive*) lexicographic structure as that of **DAR** general entry.

Example 2.2.1. The entry **LĂMURÍ** [Eng: *elucidate, explane, clear up*] in **DAR**, followed by the *French* segment, the *sense description SenseSeg* segment, and a *Nest* segment (the segment and sense markers are highlighted in 25% grey:

LĂMURÍ vb. IV^a. 1°. *Purifier, raffiner.* 2°. *Préciser; fixer; éclairer; s'éclairer, s'élucider.* 3. *Expliquer.* 4°. *Distinguer, apercevoir.*

1°. T r a n s. (Despre metale, etc.) A curăţi prin foc de corpurile necurate; p. g e n e r. a c u r ă ţ i, a l i m p e z i, a p u r i f i c a. *Ca aurul în ulcea i-au lămurit.* MINEIUL (1776) 154^2/1. *În cuptoriul înfrânării ţi-ai lămurit trupul.* ib. 45$^1/_2$. *Argintarul lucrează argintul lămurindu-l prin foc cu plumb, care trage arama.* I. IONESCU, M. 714.

... ...

[Şi: **lămurá** † vb. I^{a}. *Hierul* (= fierul) *ce lămura* [făurarul]. herodot, 28. || A d j e c t i v e: **lămurít** (cu negativul **nelămurit**), **-ă** = curăţit, limpezit, purificat; clarificat, desluşit, limpezit, explicat, clar, limpede. (Ad 1°) *Argintul lămuritu iaste cuvântul lu Dumnezeu.* coresi, ev. 318/$_{5}$; cf. dosofteiu, ps. 38. *Tăia iarăşi bani de argint lămurit.* herodot, 262. *Argintul cel cu foc lămurit.* biblia (1688) 372$_{2}$. *Laptele cel lămurit.* mineiul (1776) ... ...

... ...

... ... *Să-şi facă o idee lămurită de sine însuşi.* marcovici, c. 11/$_{1}$. *Adevăruri lămurite.* i. ionescu, c. vi. *Să-i dea mai lămurit răspuns.* c. negruzzi, i 197. *Hotărirea împărătesei era lămurită.* ispirescu, l. 307; – (în poezia populară cu caracter mistic) **lămurát, -ă.** *Să rămână curat, Lămurat, Cum Dumnezeu l-o dat.* marian, d. 34, 39, 125; – **lămuritór,-oáre** adj. = curăţitor, limpezitor, purificator; care lămureşte, care desluşeşte, care clarifică. *Dovezi lămuritoare.* donici, f. 44. *Lămuritoare cuvinte de dreptate.* c. negruzzi, II 297. | A b s t r a c t: **lămuríre** s. f. = acţiunea de a lămuri; limpezire, curăţire, purificare; claritate, desluşire, explicaţiune. **Cu lămurire** loc. adv. = în mod lămurit, clar, limpede. *Urmează a se face socotealile tovărăşiei cu multă lămurire.* pravila (1814) 87. *Am văzut cu lămurire.* uricariul, i 216/$_{2}$. *Acest adevăr rămâne cu lămurirea cuvenită.* i. ionescu, c. 243. *Trebue să dăm mai întâi o lămurire despre acest rege.* c. negruzzi, i. 177. *Să aibă la cine alerga la lămuriri, când lecţia ar fi fost prea grea.* g. vifor, luc. iv 309. (Învechit) *Lămurire a socotelelor* = lichidare. pontbriant, barcianu. *Despre Bârlad... iarăşi avem preţioase lămuriri.* bogdan, c. m. 2.].

2.3 Recursive Configuration of Lexicographic Segments in TLF

Example 2.3.1. "**Rem.**", "**Étymol. et Hist.**", and "**DÉR.**" lexicographic segments in the **TLF** entry **ÉLÉPHANT**. Along with lexical-semantics *sense trees* (with primary, secondary, and enumeration-described subsenses) inside several lexicographic segments, see also the **Rem.** segment inside the last **DÉR.** segment!

... ...

Rem. On rencontre ds la docum. **a)** *Éléphantarque,* subst. masc., antiq. Chef d'une compagnie de soldats montés sur des éléphants. *Deux armées entières : trente mille hommes d'un côté, onze mille de l'autre, sans compter les éléphants avec leurs éléphantarques* (FLAUB., *Corresp.,* 1860, p. 384). **b)** *Éléphante,* subst. fém. rare. Femelle de l'éléphant. *Emploi métaph.* Femme lourde qui manque de souplesse (*cf.* HUYSMANS, *Art mod.,* 1883, p. 133). **c)** *Éléphas,* subst. masc. Nom scientifique de l'éléphant. *L'" Elephas meridionalis", comme d'ailleurs la plupart des éléphants qui se baladaient autrefois en Europe, n'avait pas de fourrure* (FARGUE. *Piéton Paris,* 1939, p. 129).

Prononc. et Orth. : [e l e f ä]. Ds *Ac.* dep. 1694. **Étymol. et Hist. 1.** 1121 *elefant* (*Ph. Thaon Best.,* 1416 ds T.-L. : une beste truvum qu'**elefan** apelum); **2.** 1825 p. ext. " personne à la démarche lourde et peu gracieuse " (BRILLAT-SAV., *Physiol. goût,* p. 227); **3.** 1560 *elephant de mer* (PARÉ, éd. Malgaigne, *Discours de la licorne,* III, chap. XI, p. 502). Empr. au lat. *elephantus* " éléphant ", en a. fr. on rencontre plus souvent la forme *olifant**. **Fréq. abs. littér. :** 926. **Fréq. rel. littér. :** XIX^{e} s. : a) 1 789, b) 2 429; XX^{e} s. : a) 678, b) 701.

DÉR. 1. Éléphanteau, subst. masc. Petit de l'éléphant; jeune éléphant. *Des éléphanteaux se séchant au soleil* (GREEN, *Journal,* 1938, p. 144). – [e l e f ä t o] – 1re attest. XVI^{e} s. (Ant. du Pinet ds DELB. *Rec.* ds *DG*); de *éléphant,* suff. *-eau**. – Fréq. abs. littér. : 1. **2. Éléphantesque**, adj. Comparable à l'éléphant; qui est, en poids et en taille, supérieur à la moyenne. Synon. *énorme, gigantesque, gros, monumental. C'est une dame* [*la comtesse Fontaine*] *aux proportions éléphantesques, dans la fleur de la soixantaine* (COPPÉE, *Toute une jeun.,* 1890, p. 220). – [e l e f ä t ɛ s k] – 1re attest. 1890 *id.*; de *éléphant,* suff. *-esque**. **3. Éléphantin, ine**, adj. **a)** Relatif à l'éléphant; qui rappelle l'éléphant. *L'épiderme éléphantin des mendiants* (HUYSMANS, *Là-bas,* t. 2, 1891, p. 20). *Belle autrefois* [*Taitou*], *de cette beauté grasse que recherchent les Orientaux, mais devenue avec le temps d'une corpulence éléphantine* (THARAUD, *Passant Éthiopie,* 1936, p. 110). ... ...

... ...

L'énorme Suédoise beauté éléphantique (SIMONIN, BAZIN, *Voilà taxi!* 1935, p. 141). Qui est atteint d'éléphantiasis. Synon. *éléphantiasique, éléphantiaque.* Attesté ds LITTRÉ, *Ac. Compl.* 1842, BESCH. 1845, *Lar.* $19^e - 20^e$ et QUILLET 1965. **Rem.** Certains dict. attestent l'emploi subst. dans le sens de " éléphantiasique, éléphantiaque ". – Dernière transcr. ds LITTRÉ : é-lé-fan-ti-k'. – 1^{res} attest. **a)** XV^e s. subst. (*Valenciennes,* ap. La Fons. ds GDF.), **b)** adj. " d'éléphant " 1506-1516 (FOSSETIER, *Chron. Marg.,* ms. Bruxelles, 10512, IX, II, 5 ds GDF. *Compl.*); *de éléphant,* suff. *-ique**.

BBG. – GILI GAYA (S.). Miscelánea. *Revista de Filologia española.* 1949, t. 33, pp. 145-146. – GOTTSCH. Redens. 1930, p. 42, 121. – GRIMAUD (F.). Pt gloss. du jeu de boules. *Vie Lang.* 1968, p. 194. – ROG. 1965, p. 42, 178, 180. – ROMMEL 1954, p. 98. – SPITZER (L.). Über einige Wörter der Liebessprache. Leipzig, 1918, p. 56. – VAGANAY (H.). Qq. mots peu connus. *In :* [*Mél. Chabaneau (C.)*]. *Rom. Forsch.* 1907, t. 23, p. 226 (*s.v. éléphantin).*

Example 2.3.2. Highly refined description of the sense tree for the "**Étymol. et Hist.**" lexicographic segment in the **TLF** entry **VENIR**.

... ...

Prononc. et Orth.: [v ə n **:** R], *(il) vient* [-v j ɛ̈]. Att. ds *Ac.* dep. 1694. Conjug. ind. prés.: *je viens, tu viens, il vient, nous venons, vous venez, ils viennent*; imp.: *je venais*; passé simple: *je vins*; fut.: *je viendrai*; passé composé: *je suis venu*; plus-que-parfait: *j'étais venu*; passé ant.: *je fus venu*; futur ant.: *je serai venu*, cond.: *je viendrais*; cond. passé: *je serais venu*; subj. prés.: *que je vienne*; imp.: *que je vinsse* ; passé *que je fus venu*; plus-que-parfait: *que je fusse venu*: impér.: *viens, venons, venez*; passé: *sois venu, soyons venu, soyez venu*; inf. prés.: *venir*; passé: *être venu*; part. prés.: *venant*; passé: *venu, -ue; étant venu.* **Étymol. et Hist. A. 1.** *Venir a* + subst. marquant le terme du mouvement **a)** *ca* 880 " se déplacer pour arriver près du point de référence " (*Eulalie*, 28 ds HENRY *Chrestomathie*, p. 3); *ca* 1050 *en venir " id. "* (*Alexis*, éd. Chr. Storey, 113); spéc. 1690 " atteindre un certain point " (FUR.); 1842 mar. (*Ac. Compl.:* **Venir** au vent [...]. **Venir** à bâbord ou à tribord); **b)** 1176-81 fig. *venir à* + subst. ab-

str. “ apparaître dans l’esprit, être conçu ” (CHRÉTIEN DE TROYES, *Charrete*, éd. M. Roques, 495); **2.** *venir de* + subst. indiquant l’origine du mouvement **a)** *ca* 1050 “ arriver en provenance de ” (*Alexis*, 251); **b)** *ca* 1170 fig. “ provenir, découler de ” (CHRÉTIEN DE TROYES, *Erec*, éd. M. Roques, 4392); spéc. *ca* 1250 “ descendre (de quelqu’un) ” (*Grant mal fist Adam*, I, 28 ds T.-L.); 1606 “ dériver (d’un mot) ” (NICOT, *s.v. bohourd*); **c)** loc. 1176-81 *don vos vient?* (CHRÉTIEN DE TROYES, *Charrete*, 137); 1580 *d’où venoit celà* (MONTAIGNE, *Essais*, I, 20, éd. P. Villey et V.-L. Saulnier, p. 96); 1664 *d’où vient que* (MOLIÈRE, *Tartuffe*, I, 1); **3. a)** *ca* 1050 *venir* sans compl. de lieu (*Alexis*, 467); *ca* 1050 *faire venir qqn* “ lui demander de venir ” (*ibid.*, 335); 1539 *venir au secours* (EST.); . . . . . .

. . . .

D. Avec l’inf. *venir* servant de simple auxil. **1.** fin X^{e} s. *venir* + inf. “ faire en sorte de ” (*Passion*, 407); **2.** *ca* 1050 *venir a* surtout à la 3^{e} pers. + inf. “ se trouver en train de ” (*Alexis*, 47); **3.** *ca* 1225 *venir de* + inf. “ avoir juste fini de ” (GAUTIER DE COINCI, *Mir.*, éd. V. Fr. Koenig, I Mir 12, 44). Du lat. *venire* “ venir ”, “ arriver, se présenter ”, “ parvenir à ”, “ venir à quelque chose, venir dans tel ou tel état ” et “ en venir à ”. **Fréq. abs. littér.:** 98 961. **Fréq. rel. littér.:** XIX^{e} s.: a) 142 843, b) 153 800; XX^{e} s.: a) 144 519, b) 129 650. **Bbg.** BAMBECK (M.). Galloromanische Lexikalia aus volksprachlichen mittelalterlichen Urkunden. *Mél. Gamillscheg (E.)* 1968, p. 69. – DABÈNE (L.). *Aller* et *venir*: de la ling. à la didact. *Mél. Pottier (B.)* 1988, pp. 217–224. – DEJAY (D.). Les Rel. actancielles appréhendées à travers un corpus de verbes fr. Thèse, Nancy, 1986, pp. 37–42. . . . .

It is clear that any dictionary parser should recognize first (explicitly expressed or by default) the lexicographic segments within the first SCD parsing configuration.

3 Parsing Problems at the Level of Primary and Secondary Sense Definitions on the SCD-*Config*2

3.1 Cyclicity Calls between Secondary Sense Markers and Literal Enumeration in DMLRL

Example 3.1.1. It is common in **DMLRL** that (primary and) secondary senses to be refined by literal enumeration. For the reverse, atypical and uncommon situation, where the literal enumeration is further refined through secondary sense markers // and ♢, the most interesting case we met in **DMLRL** is the entry **БЫ** [9 :844], under the primary sense no. "**3.**".

... ...

2. В придаточной части сложного предложения обозначает действие, обусловливающее собой то, о чем сообщается в главной части. *Когда б разбойника облавою не взяли, То многие еще бы пострадали.* Михалк. Бешен, пес

3. Обозначает различные оттенки желаемости действия; **а)** Собственно желаемость. *Учился бы сын. Были бы дети здоровы.* ♢ Если бы, когда бы, хоть бы и т. п. *О, если бы когда-нибудь Сбылись поэта сновиденья!* Пушк. Посл. к Юдину. [Николка:] *Хоть бы дивизион наш был скорее готов.* Булгаков, Дни Турб. ♢ С неопр. ф. глаг. *Полететь бы пташечке К синю морю; Убежать бы молодцу в лес дремучий.* Дельв. Пела, пела пташечка.. [Настя:] *Ах, тетенька, голубок! Вот бы поймать!* А. Остр. Не было ни гроша... — *Жара, дедушка Лодыжкин .. Нет никакого терпения! Искупаться бы!* Купр. Бел. пудель. // Употр. для выражения опасения по поводу какого-л. нежелательного действия (с отрицанием). *Не заболел бы он.* ♢ С неопр. ф. глаг., имеющей перед собой отрицание. — *Гляди, — говорю, — бабочка, не кусать бы тебе локтя! Так-таки оно все на мое вышло.* Леск. Воительница. ♢ Только бы (б) не. — *По мне жена как хочешь одевайся, .. только б не каждый месяц заказывала себе новые платья, а прежние бросала новешенькие.* Пушк. Арап Петра Вел. [Варя:] *Не опоздать бы только к поезду.* Чех.

Вишн. сад. **б)** Пожелание. *Условие я бы предпочел не подписывать.* Л. Толст. Письмо А. Ф. Марксу, 27 марта 1899. ♢ С неопр. ф. глаг. *Поохотиться бы по-настоящему, на коня бы денег добыть, — мечтал старик.* Г. Марков, Строговы. ♢ В сочетании с предикативными наречиями со знач. долженствования, необходимости, возможности. [Алеша Бровкин] *сверкнул глазами и понесся .. по гнилым полам приказной избы. Вслед ему косились плешивые повытчики: "Потише бы надо, бесстрашной, здесь не конюшня".* А. Н. Толст. Петр I. ♢ Только бы (б), лишь бы, Употр. со знач. желательности действия. [Скалозуб:] *Мне только бы досталось в генералы.* Гриб. Горе от ума. **в)** Желание-просьба, совет или предложение (обычно при мест. 2л.). [Марина:] *И чего засуетился? Сидел бы:* Чех. Дядя Ваня. — *Пошел бы ты к ним счетоводом, полковник.* Павлен. Счастье. — *Ты бы, Сережа, все-таки поговорил с Лидией:* Пришв. Кащ. цепь. **г)** Желаемость целесообразного и полезного действия. ♢ С неопр. ф, глаг. *Вам бы вступиться за Павла-то!* — воскликнула мать, вставая. — Ведь он ради всех пошел. М. Горький, Мать. ♢ С неопр. ф. глаг., имеющей перед собой отрицание. [Лиза:] *А вам, искателям невест, Не нежиться и не зевать бы.* Гриб, Горе от ума.

~ ***Во что бы то ни стало.*** См. Стать. ***Как бы не так.*** См. Как. ***Кто бы ни был, что бы ни было, как бы то ни было.*** См. Быть. ***Хоть бы хны.*** См. Хоть. ***Хоть бы что.*** См. Хоть.

— Срезневский: бы; Лекс. 1762: бы.

The parsing result of this part of **БЫ** entry is the following:

<entry>

<list>БЫ 1. ♢ ♢ ♢ ♢ ♢ 2. 3. а) ♢ ♢ // ♢ ♢ б) ♢ ♢ ♢ в) г) ♢ ♢ n-23</list>

<sense value="БЫ"class="0">

<definition> (сокращенно Б), частица. В сочетании с глаголами в форме прошедшего времени образует сослагательное наклонение. </definition>

<sense value="1."class="4">

... ...

<sense value="3."class="4">

<definition> Обозначает различные оттенки желаемости действия; </definition>

<sense value="а)"class="5">

<definition> Собственно желаемость. Учился бы сын. Были бы дети здоровы. </definition>

<sense value="♢"class="8">

<definition> Если <spaced> б ы </spaced>, когда <spaced> б ы </spaced>, хоть <spaced> б ы </spaced><spaced> и </spaced> т. п. О, если бы когда-нибудь Сбылись поэта сновиденья! Пушк. Посл. к Юдину. [Николка:] Хоть бы дивизион наш был скорее готов. Булгаков, Дни Турб. </definition>

< /sense>

<sense value="♢"class="8">

<definition> С неопр. ф. глаг. Полететь бы пташечке К синю морю; Убежать бы молодцу в лес дремучий. Дельв. Пела, пела пташечка.. ... ... </definition>

< /sense>

<sense value="//"class="6">

<definition> Употр. для выражения опасения по поводу ... ... </definition>

<sense value="♢"class="8">

<definition> С неопр. ф. глаг., имеющей перед собой отрицание. – Гляди, – говорю, – бабочка, не кусать бы тебе локтя! Так-таки оно все на мое вышло. Леск. Воительница. </definition>

< /sense>

<sense value="♢"class="8">

<definition> Только <spaced> б ы</spaced> (б) не. - По мне жена как хочешь одевайся, .. только б не каждый месяц ... ... </definition>

< /sense>

< /sense>

< /sense>

<sense value="б)"class="5">

<definition> Пожелание. Условие я бы предпочел не подписывать. Л. Толст. Письмо А. Ф. Марксу, 27 марта 1899. </definition>
<sense value="◊"class="8">
<definition> С неопр. ф. глаг. Поохотиться бы по-настоящему, на коня бы денег добыть, - мечтал старик. Г. Марков, Строговы. </definition>
< /sense>
<sense value="◊"class="8">
<definition> В сочетании с предикативными наречиями со знач. долженствования, необходимости, возможности. ... ... </definition>
< /sense>
<sense value="◊"class="8">
<definition> Только <spaced> б ы</spaced> (б), лишь бы, Употр. со знач. желательности действия. [Скалозуб:] Мне только бы досталось в генералы. Гриб. Горе от ума. </definition>
< /sense>
< /sense>
<sense value="в)"class="5">
<definition> Желание-просьба, совет или предложение.... ... ... </definition>
... ...
< /sense>
< /sense>
< /sense>
<EtymologicalPart>
 – Срезневский: <spaced> б ы</spaced>; Лекс. 1762: <spaced> б ы</spaced>.
< /EtymologicalPart>
< /entry>

The *Enumeration Closing Condition* (ECC) represents a deterministic, computational constraint devoted to check the sound termination (*i.e.* in a deterministic, finite number of steps) of the literal or numeral enumeration marker list, when higher-level sense markers break into this list. When this happens, contextual look-ahead verifications are needed

to obtain the correct closing of the enumeration list. More precisely, ECC means that whether after a certain (let us say, *current*) *letter* in the sense enumeration marker list occur higher-level sense markers (on the dependency hypergraph), then one should look ahead in the sense marker sequence until the *next* letter of the same enumeration type occurs. If such a letter does exist and follows *monotonously* (in the alphabetic order) the current one in the enumeration list, then the enumeration should continue. Otherwise, *i.e.* the letter does not exist or it begins another enumeration, of the same or another kind as the current one, then the ECC holds and the current literal enumeration must be closed. For instance, in the Romanian **DLR**, with the filled and empty diamonds ♦, ♢ as secondary sense markers, the enumeration list **a) b) c)** ♢ ♦ ♢ ♢♦ ♢ **d)...** should continue, while the marker sequence **a) b) c)** ♢ ♦ ♢ ♢♦ ♢ **a)...** should close the first literal enumeration (see also [5], [4], [6], [7]).

The same is true if non-enumerable sense markers (such as ♦, ♢) are replaced by another enumeration of sense markers, be it of numeral or another literal type. Two different enumerations, a standard, *literal* one, and a *numeral* one coming from transforming the *New_ Paragraphs* into sense markers, are illustrated by the entry **CAL** of the Romanian **DAR** thesaurus.

3.2 Cyclicity Calls between Secondary Sense Markers, Literal Enumeration, and *New_ Paragraphs* in DAR and DLR

Example 3.2.1. [7 :Chap. 9] In the **DAR** entry of the preposition **DE** (En: *of, by, for, to, from...*, Fr: *de*) we encounter the situation of the *NewPrg* (*New_ Paragraph*) use as *numeral* enumeration, pursued or not by another sense marker: *NewPrg* introduces component subsenses in the (Romanian) *RomSeg* segment, which follows the (French) *FreSeg* segment.

<FreSeg>

NewPrg **DE** prep. A. I. 1°. a). Marque le lieu d'où part une action...

... ...

NewPrg F. Elément de nombreaux mots composés.

< /FreSeg>

<RomSeg>

NewPrg De neaccentuat în frază şi proclitic, formează o singură unitate fonetică...

NewPrg Substantivul în legătură cu *de* rămâne de obiceiu nearticulat, dacă nu e urmat de un atribut al său...

NewPrg Cuvântul de sub regimul lui *de* are de cele mai multe... ... ...

{*RomSeg* contains 14 paragraphs introduced by *NewPrg*, followed by *RomSeg* and *SenseSeg*. Hence:}

... ...

< /RomSeg>

<SenseSeg>

NewPrg A. Construcţia prepoziţională are funcţiunea sintactică...

NewPrg I. Ca determinare privitoare la spaţiu sau la timp.

NewPrg 1^0. Complemente circumstanţiale de loc.

NewPrg a) Complementul circumstanţial de loc răspunde la întrebarea u n d e?...

...

< /SenseSeg>

Example 3.2.2. [7 :Chap. 9] The illustrative example of entry **CAL** from **DAR** is important and rather complex, showing the use of *NewPrg* markers as sense *numeral* enumeration, interleaving with the already existing sense *literal* enumeration.

NewPrg **CAL** s.m. *Cheval.*

NewPrg 1°. Numele generic al speţei cavaline; s p e c. individ masculin...

...

NewPrg Adecă amù cailoru zăbalele în gură lă...

... {a large block of definitions and *DefExems* of the entry **CAL**}

NewPrg În compoziţii:

NewPrg a.) (Entom.) **Cal-de-apă** = o specie a c a l u l u i - d r a c u l u i, numită...

...

NewPrg **Calul-dracului** = a.) insectă cu corpul lung... | (De aici) Babă rea... ; –b.) = **cal-de-apă**...

...

NewPrg **Calul-popii** = a.) c a l u l-d r a c u l u i...; –b.) = cal-de-apã... Insectă lungă şi cu aripile pătate...

NewPrg **Cal-turtit** = c a l u l-d r a c u l u i...

NewPrg b.) (Zool.; la românii din A.-U.) **Cal-de apă** s. (după germ. Nilpferd) –**cal-de-Nil** = h i p o p o t a m LB., BARCIANU ...

...

NewPrg **Cal-de-mare** = *hyppocampus brevirostris*...

...

NewPrg 2°. P. a n a l. (Mor.) *Caii* cu spetezele ţin coşul şi alcătuesc...

...

The sense dependency subtree between the sense markers "1°." and "2°." looks as follows (Fig. 1. below):

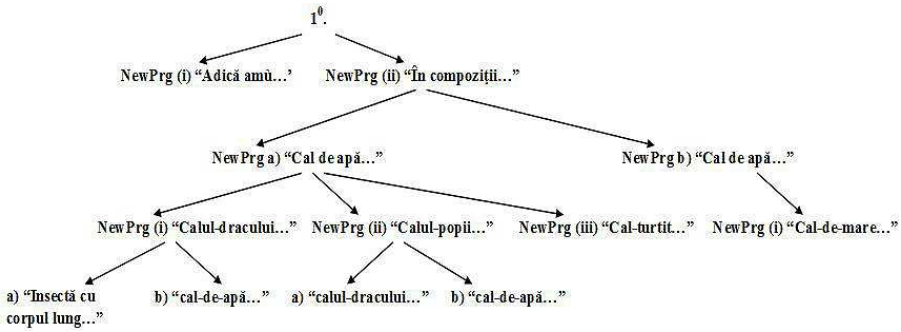

Figure 1. Partial sense dependency subtree of the **CAL** entry in **DAR**

A good exercise of solving this problem is to parse correctly the entry **CAL** in **DAR**, partially shown below. This complete representation extends a slightly less refined output obtained by the automatic SCD-based parser.

<entry>

<sense value="CAL" class="0">

<definition> s.m. Cheval.</definition>
<sense value="1°." class="12">
<definition> Numele generic al speţei cavaline; s p e c. individ masculin... M â n z u l dacă nu se ţine de prasilă</definition>
... ...
<sense value="NewPrg" class="1°.+i.">
<definition>Adecă amù cailoru zăbalele în gură lă [= le]băgăm COD. VOR. 122/13.
Nu fireţi...</definition>
< /sense>
<sense value="NewPrg" class="1 ° .+ii.">
<definition>În compoziţii:</definition>
< /sense>
<sense value="a.)" class="16">
<definition> (entom.) Cal-de-apă = o specie a c a l u l u i-d r a c u l u i, numită şi c ă l u ţ - d e - a p ă, c a l u l - d r a c u l u i, c a l u l - p o p i i, c ă l u ţ, p ă u n i ţ ă, p i ţ i n g ă u l - d r a c u l u i, s c ă l u ş - d e - a p ă, ţ â n ţ a r - d e -apă (Calopteryx splendens). MARIAN, INS. 559-560, cfr. H. XI 195.</definition>
< /sense>
<sense value="NewPrg" class="a+i.">
<definition>Calul-dracului =
<sense value="a.)" class="16"> <definition> insectă cu corpul lung şi turtit, de coloare galbenă închisă, cu aripile lungi şi late, şi străvezii ca o păioară. Zboară foarte iute, mai ales pe de-asupra apelor. Se mai numeşte: c a l u l - p o p i i, c a l-t u r t i t, c o b i l i ţ ă, c ă l u g ă r i ţ ă (H. x 355) (*Libellula depressa*). MARIAN, INS. 558 ž. u., „un fel de ţânţar mare" H. IX 52. Cfr. H. I 59, IV 54, V 116, IX 437, 473, x 259, XII 27, 374.</definition>
< /sense>
<sense value="#" class="22"> <definition> A fi ca calul-dracului, se zice de un om neastâmpărat. marian, ins. 565. </definition>
< /sense>

<sense value="|" class="20"> <definition> (De aici) Babă rea, cfr. n e a g a r e a. Cfr. coşbuc, b. 92. <i>Baba asta (vrăjitoare) erà calul-draculuì<i>: afurisită şi rea. PAMFILE, J. I, cfr. ZANNE, P. II 3;– </definition>

< /sense>

<sense value="b.)" class="16"> <definition> c a l-d e-a p ã. MARIAN, INS. 559. </definition>

< /sense>

< /definition>

< /sense>

<sense value="NewPrg" class="a+ii.">

<definition>Calul-popii =

<sense value="a.)" class="16">

<definition> c a l u l - d r a c u l u i. MARIAN, INS. 558; </definition>

< /sense>

<sense value="b.)" class="16">

<definition> c a l-d e-a p ă. id. ib. 559. Insectă lungă şi cu aripile pătate, având ochii mari. H. VII 481; cfr. H. I 59, II 307, 227, 117, V 280, X 151, 355, 498, XII 226, 429, XIV 350, 397, 467. </definition>

< /sense>

< /definition>

< /sense>

<sense value="NewPrg" class="a+iii.">

<definition> Cal-turtit = c a l u l - d r a c u l u i. MARIAN, INS. 558.

< /definition>

< /sense>

<sense value="NewPrg" class="b+i."="a+iv.">

<sense value="b.)" class="16">

<definition> (Zool.; la Românii din A.-U.) Cal-de apă s. (după germ. Nilpferd) -de-Nil = h i p o p o t a m LB., BARCIANU. </definition>

< /sense>

<sense value="NewPrg" class="1°.+iii."="b+ii.">

<definition> Cal-de-mare= hyppocampus brevirostris. BARCIANU. Cai-de-mare, albi ca spuma, EMINESCU, p. 114. </definition>

< /sense>

< /sense>

<sense value="2°." class="12">

<definition> P. a n a l. (Mor.) Caii cu spetezele ţin coşul şi alcătuesc... </definition>

<sense value="||" class="20">

<definition> (Dulgh.) S c a u n u l cu cleştele de strâns... </definition>

< /sense>

< /sense>

.. ..

<sense value="4°." class="12">

<definition> (Cor.) Numele unui danţ ţărănesc...</definition>

< /sense>

< /sense>

< /entry>

The partial sense marker sequence in the above representation is the following: ... ... 1°. i. ii. a.) BoldDefMark i. BoldDefMark a.) # | b.) ii. BoldDefMark a.) b.) iii. BoldDefMark i. b.) BoldDefMark ii. BoldDefMark 2°.|| ... . We remark the distinct role of *NewPrg* typographic-type sense marker in the context of subsequences NewPrg DefMark Enum and NewPrg Enum DefMark: the first sequence introduces lower, local level dependencies, while the second one defines higher level ones, all depending on the look-ahead sense markers. The subsequence contextual analysis and two passages along the whole sense marker sequence provide the correct sense dependencies.

Such an approach would be rather difficult to be implemented within the classical, formal grammar-based grammars, since it works depth-first search on *all* the dictionary forms, definition bodies, and sense markers, while ECC and the emphasized contextual analyses on the

marker subsequences are performed on the *bare sequence* of the *extracted sense markers* from the entry. Dependency structures such as in the entry **CAL** of **DAR** represent, in our evaluation, lexicographic mistakes or inadequacies at the dictionary design stage; parsing it correctly with the method of SCD configurations is both a technical challenge and also a warning for more sound and careful sense structure constructions in the greatest thesaurus-dictionaries.

Example 3.2.3. While the secondary sense markers are naturally refined through literal enumeration in **DLR** thesaurus, we found yet the reverse, atypical situation, *e.g.* for the entries **DOAR**, **DOÁSCĂ** (fragment below), and especially **LUMÍNĂ** (fragment below), where the recursive calls for literal enumeration is mixing with secondary sense markers. The first literal *enumeration* is notably further marked by another, *numeral* enumeration, introduced by the *NewPrg* (*New_ Paragraph*) markers.

DOÁSCĂ s. f. **1.** Nume dat unor scânduri, unor bucăţi de lemn sau unor obiecte făcute din acestea:

a) (Popular) Scândură (**1**). *Strunga de muls e închisă cu o doscă, scândură, până se pun la muls păcurarii.* DR. II, 336. *Şi-a lăsat abatajul nearmat ... şi coperişul fără doasce.* DAVIDOGLU, M. 70. *Îl pun caşul undeva pe-o doscă.* Com. din LUGAŞU DE JOS – ALEŞD, cf. ALR I 1 853/61, 65, 80, 107. ♦ *Gard de doşte* = gard de scânduri. Cf. ALR II/I h 267/64, ALRM II/I h 359/64. ... ...

g) (Învechit) Copertă de carte, confecţionată din lemn şi învelită în piele. *Mi se încredinţase un dulap nou-nouţ ... Era încărcat cu fel de fel de bucoavne vechi, cu doascele de lemn.* CIAUŞANU, R. SCUT. 55, cf. ARH. FOLK. VII, 121. ◊ Loc. adv. **Din doască-n doască** = în întregime, de la un capăt la altul. *Secretarul întreprinderii luă traducerea şi o citi din doască-n doască.* AGÎRBICEANU, A. 53.

2. (Regional) Perete subţire (Bonţ – Gherla). Cf. PAŞCA, GL. **3.** (Regional) Vas făcut din coajă de dovleac. *Sus pe corlată ... trei doaşte de dovlete.* PLOPŞOR, C. 39. ... ...

LUMÍNĂ s.f. **A.** (Predomină sensul concret de radiaţie; în opoziţie

cu î n t u n e r i c)

I. (Adesea cu determinări calificative) Radiaţie care face corpurile vizibile.

1. (Ca atribut al universului, al naturii ambiante; componentă a lumii înconjurătoare) *Lăudaţil toate stealele şi* ... ... *gonească Cât va fi câmp de gonit Şi lumină de zărit".* ALECSANDRI, O. I, 8.

a) (Ca radiaţie solară, element al peisajului diurn) *Voi întoarce lumira soarelui de către voi, de va fi întunrearecu* (a. 1600). CUV. D. BĂTR. II, 49/9. *Lumina soarelui face dzua.* PRAV. 141. ... ... *Deopotrivă se găseşte-n toate Amestecată umbră şi lumină.* ISANOS, V. 281. ⧫ L o c. a d j. **De lumină** = **a)** luminos, sclipitor; s p e c. (despre ochi) strălucitor. *Deunăzi ... mă simţii cufundat ca într-un nor întunecos ... Ancuţo! tu ai prefăcut acel nor în soare de lumină! Tu ai deşteptat în sufletu-mi o viaţă necunoscută!* ODOBESCU, S. I, 143. ... ... *Ochi de lumină avea fiul lui Ieronim, privirea lui în noapte fulgera.* ROMÂNIA LITERARĂ, 1970, nr. 93, 17/3 ; **b)** (despre un spaţiu, un loc) în care pătrunde lumina **(A I 1),** plin de lumină *Acest loc ... era pe atunci, în 1650, un ochi de lumină în mijlocul marelui codru al Căpoteştilor.* IORGA, C. I. II, 5 ; **c)** (despre plante) care trăieşte la lumină **(A I 1).** *După o fază de 2-3 ani cu floră de buruieni de lumină, urmează faza de fâneaţă cu ierburi cu rizomi.* CHIRIŢĂ, P. 71. ⧫ L o c. a d v. **Pe** (sau, rar, **la) lumină** = în timpul zilei **(I 2),** de ... ...

... ... ARHIVA R. I, 87/20. *A înviat din morţi ..., Lumina ducându-o Celor din morminte!* EMINESCU, O. IV, 359. *Zâmbetul sfânt al martirului care-ntrevede ... lumina vieţii eterne.* CARAGIALE, O. II, 64. (Contextul aduce sensul figurat privind viaţa interioară a individului) *Cine va îmbla zioa nu se va poticni ...; iară cine va îmbla noapte poticni-se-va, că lumină nu iaste întru el.* CORESI, EV. 95.

b) (Ca radiaţie reflectată de lună; element al peisajului nocturn) *Luna, ... fire are lumina ce iase den ea să turbure udăturile trupului.* CORESI, EV. 81. ... ...

Mare şi minunată este lucrarea luminii lunii asupra feţii pământului şi a sănătăţii locuitorilor lui. EPISCUPESCU, PRACTICA, 335/2. ... ...

The discussion and solution is similar as that for the entry **CAL** from **DAR**.

Example 3.2.4. In this **DLR** entry, the ♦ secondary sense is inserted within the literal enumeration and, irregularly, subordinated to it!

LÚBENE s.m. (Munt.) Numele dat unor plante din familia cucurbitaceelor: **a)** (şi în sintagma *lubene turcesc,* H II 326, ALR I 855/725, ib. 856/725, 730, 735, 740) dovleac *(Cucurbita maxima).* ♦ *Lubene scoromic* = pepene galben *(Cucurbita melo).* Cf. ALR I 857/740. *Era nouă morţi. Şedea ca lubenii.* GEORGESCU-TISTU, B. 35. Cf. ALR I 856/710, 725, 730, 735, 740, ALR SN I h 198/723, ALRM SN I h 137/723; **b)** dovleac, bostan *(Cucurbita pepo).* Cf. DDRF, SCRIBAN, D., ALR I 855/710, 725, 730, 735, 740. Cf. H II 79, 326, XI 321. *.Albina zbărrr! dup-o floare de lubene, unde se pitise ca s-audă ce va zice.* POP., ap. HEM 1 650

4 Parsing the Atomic Sense Definitions on SCD-*Config*3

The complete parsing of atomic definitions of a dictionary entry relies essentially on the pre-established *dependency hypergraph* of the SCD-*Config*3, as that in [5 :Fig. 2, p. 75], connected to the hypergraph(s) on SCD*Config*2. In this section we point out only few problems that may generate unsound dependencies within the sense trees of the parsed entries on the SCD*Config*3 level: **(1)** Reliable recognition of the atomic sense definitions, including context-depending ones (*e.g. TildaDef* in **DMLRL** [5 :48], *BoldDef* and *ItalDef* in **DLR**, **DAR** [4], [3], [7]); **(2)** Cycling calls between atomic sense definitions and literal enumeration, marked or not by *NewPrg*; **(3)** New kinds, non-standard types of sense definitions and examples-to-definitions; **(4)** Various situations

of *definition inheritance*, either explicit ones (*e.g.* with the *inheritance-dash* marker) as in **TLF** or **GWB**, or by implicit (non-marked) definition inheritance, as frequently occur in **DLR** or **DAR**, along with the sense dependencies they generate.

Remark 4.1. Atomic definitions *BoldDef* and *ItalDef* in **DLR-DAR** may often be refined through literal enumeration. Since the reverse situation is also frequent, when met together they may cause dependency assignment disagreements, as illustrated in examples 3.2.3 and 3.2.4 above.

Example 4.2. Here it is a sample of 'new' atomic definition, something between *ItalDef* and *DefExem* (excerpt from **LÎMBĂ** in **DLR**). Another (this time, very useful) case: *Indexed DefExem* (excerpt from **BRAVE** in **TLF**) [4], [7]. "Unknown" definition species may always be invented, either useful or not, but they may involve recognition problems in the parsing process.

... ...

Limba oase n-are (= poţi spune cuiva ceva, îl poţi sfătui, ştiind însă că nu va lua în seamă, nu se va conforma spuselor tale). I. CR. IV, 22. *Limba oase n-are, dar oase sfarmă* (= cu cuvântul mari lucruri săvârşim). I. GOLESCU, ap. ZANNE, P. II, 217, PANN, P. V. I, 21. *Limba izbeşte în dintele ce te doare* (= te defaimă unde îţi pasă). I. GOLESCU, ap. ZANNE, P. II, 223. *Toată pasărea pe limba ei piere* (= într-un fel sau altul, fiecare suportă consecinţele vorbelor, ale faptelor proprii). PANN, P. V. I, 25, NEGRUZZI, S. I, 247, LĂCUSTEANU, A. 127, ODOBESCU, S. III, 10, CREANGĂ, ...

... ...

A. — **1.** Homme courageux qui ne craint pas les dangers ou les entreprises difficiles, qui les a affrontés. *Il n'y a pas d'heures pour les braves* (VERLAINE, *Œuvres posthumes,* t. 1, Souvenirs, 1896, p. 206) :

- 11. ... ... tu es sûr du cœur et du bras de ce gladiateur? Il faut *un* **brave** pour défaire Sigognac, lequel, je l'avoue, bien que je le haïsse, n'est point *lâche,* puisqu'il a bien osé se mesurer contre moi-même. T. GAUTIER, *Le Capitaine Fracasse,* 1863, p. 347. ... ...

Example 4.3. When explicitly marked (as in **TLF**, **GWB**), the sense *definition inheritance* means to establish the correct mother-node in the sense tree from where the definition should be handed down. When inheritance is 'marked' by the *lack of definition* (as in **DLR**), the work on the entry sense tree is more complex and challenging. This is an exacting topic.

... ...

3. Titlu purtat de conducătorii Ţărilor Române; persoană care avea acest titlu; domnitor (**1**), vodă, voievod (**3**), (învechit) gospodar, vlădică, biruitor. V. principe[1] (**1**), prinţ. *La putenciosul domnu Pătru-Vodă amu fost de multe ori* (a. 1593). DOC. Î. (XVI), 181. ... ...

♦ (Atribuind calitatea ca un adjectiv) *Un părinte domnu să aşaze pe un fiiu al său în scaonul părintescu.* GHEORGACHI, CER. (1762), 271. ♢ Spec. Conducător al unui principat sau al unui cnezat; principe[1] (**1**), prinţ, cneaz. Cf. MARDARIE, L. 159/14. *Domnilor de Ardeal dzicem crai ungureşti.* M. COSTIN, O. 43. ... ...

5 Toward the Soundness of Sense Structures in Thesauri

This paper discussed a series of parsing problems and solutions in the context of parsing *six* very large and sensibly different dictionaries of *four* European languages. The typical parsing problems presented are related to the cyclicity (recursive) calls of sense markers on the parsing layers of *three* SCD *configurations*. Working on modules (SCD configurations), reducing the parsing problems (almost only) to sense *marker sequence* analysis, transforming the typographical *New_Paragraphs* into sense *numeral enumeration*, which interleaves with literal enumeration and other sense marker classes, employing the *Enumeration Closing Condition* to check the sound and deterministic (and possibly multiple) use of the sense enumeration device represent the solutions and novelty contributions of the present paper. They are addressing both the dictionary parser designers and thesauri lexicographers, since almost all

the raised problems can be seen as irregularities and / or inadequacies of the sense structure definitions, affecting their lexical-semantic soundness.

References

[1] N. Curteanu, E. Amihăesei (2004). *Grammar-based Java Parsers for DEX and DTLR Romanian Dictionaries.* ECIT-2004 Conference, Iasi, Romania.

[2] N. Curteanu (2006). *Local and Global Parsing with Functional* FX-*bar Theory and SCD Linguistic Strategy.* (I.+II.), Computer Science Journal of Moldova, Academy of Science of Moldova, Vol. 14 no. 1 (40): pp. 74–102 and no. 2 (41): pp. 155–182, http://www.math.md/files/csjm/v14-n2/v14-n2-(pp155-182).pdf.

[3] N. Curteanu, A. Moruz, D. Trandabăţ (2008). *Extracting Sense Trees from the Romanian Thesaurus by Sense Segmentation & Dependency Parsing*, Proceedings of CogAlex-I Workshop, COLING 2008, Manchester, United Kingdom, pp. 55–63, ISBN 978-1-905593-56-9, http://aclweb.org/anthology/W/W08/W08-1908.pdf.

[4] N. Curteanu, D. Trandabăţ, A. Moruz (2010). *An Optimal and Portable Parsing Method for Romanian, French, and German Large Dictionaries*, Proceedings of COGALEX-II Workshop, COLING-2010, Beijing, China, August 2010, pp. 38–47, http://www.aclweb.org/anthology-new/W/W10/W10-3407.pdf.

[5] N. Curteanu, S. Cojocaru, E. Burcă (2012). *Parsing the Dictionary of Modern Literary Russian Language with the Method of SCD Configurations. The Lexicographic Modeling.* Computer Science Journal of Moldova, Academy of Sciences of Moldova, Vol. 20, No.1(58), pp. 42–81, http://www.math.md/files/csjm/v20-n1/v20-n1-(pp42-82).pdf.

[6] N. Curteanu, S. Cojocaru, A. Moruz (2012). *Lexicographic Modeling and Parsing Experiments for the Dictionary of Modern Literary Russian Language*, ConsILR-2012, Bucharest, The Editorial House of "Al. I. Cuza" University, Iaşi, pp. 189–198.

[7] N. Curteanu. (2012). *The Segmentation-Cohesion-Dependency Parsing Strategy and Linguistic Theory*, TehnoPress, Iaşi, România, xix + 420 p., ISBN: 987-973-702-928-7.

[8] Das Woerterbuch-Netz (2010). http://germazope.uni-trier.de/Projects/WBB/woerterbuecher/.

[9] Dictionary of Modern Literary Russian Language (20 volumes – 1994). M.: Russian language; Second edition, revised and supplemented, 864 p.; 1991 – 1994. ISBN: 5-200-01068-3 (in Russian).

[10] R. Hauser, A. Storrer (1993). *Dictionary Entry Parsing Using the LexParse System*. Lexikographica 9 (1993), pp.174–219.

[11] M. Kammerer (2000). *Wöterbuchparsing Grundsätzliche Überlegungen und ein Kurzbericht über praktische Erfahrungen*, http://www.matthias-kammerer.de/content/WBParsing.pdf.

[12] Le Trésor de la Langue Française informatisé (2010). http://atilf.atilf.fr/tlf.htm.

[13] L. Lemnitzer, C. Kunze (2005). *Dictionary Entry Parsing*, ESSLLI 2005.

[14] C. Mărănduc (2010). *Dictionary of expressions, locutions, and phrases*, Corint Editorial House, Bucharest, 560 p., ISBN 973-135-570-2 (in Romanian).

[15] M. Neff, B. Boguraev (1989). *Dictionaries, Dictionary Grammars and Dictionary Entry Parsing*, Proc. of the 27th annual meeting on Association for Computational Linguistics Vancouver, British Columbia, Canada Pages: pp. 91 – 101.

[16] S. Puşcariu, *et al.* (1906). Dictionary of the Romanian Language (Dictionary of the Romanian Academy – **DAR**), Bucharest, Edition 1940 (old format).

[17] D. Tufiş (2001). *From Machine Readable Dictionaries to Lexical Databases*, RACAI, Romanian Academy, Bucharest, Romania.

[18] XCES TEI Standard, Variant P5 (2007). http://www.tei-c.org/Guidelines/P5/

Neculai Curteanu, Alex Moruz Received June 27, 2012

Neculai Curteanu

Institute of Computer Science,
Romanian Academy, Iaşi Branch
Str. Gh. Asachi, Nr. 3,
700483 Iaşi, România
E–mails: *ncurteanu@yahoo.com*,
nicu.curteanu@iit.academiaromana-is.ro

Alex Moruz

Institute of Computer Science,
Romanian Academy, Iaşi Branch,
Faculty of Computer Science,
"Al. I. Cuza" University of Iaşi,
E–mails: *alex.moruz@gmail.com*,
alex.moruz@iit.academiaromana-is.ro

Computer Science Journal of Moldova, vol.**20**, no.3(60), 2012

Gröbner Basis Approach to Some Combinatorial Problems

Victor Ufnarovski

Abstract

We consider several simple combinatorial problems and discuss different ways to express them using polynomial equations and try to describe the Gröbner basis of the corresponding ideals. The main instruments are complete symmetric polynomials that help to express different conditions in rather compact way.

Keywords: Gröbner basis, zero-dimensional ideal, finite configuration, complete symmetric polynomials.

1 Introduction

As far as it was found that Gröbner basis is a nice instrument to solve polynomial systems of equations, there appear many ideas how to translate problems that do not look as suitable object for the Gröbner basis approach to non-trivial system of equations. A classical example is graph coloring (see [1], where many other interesting problems can be found). In this article we want to consider some elementary instruments that can be applied for easy combinatorial problems. The main of them is the complete symmetric polynomial.

2 How to describe a finite set?

Let us try Gröbner basis approach to some combinatorial problems in order to understand when such approach can be useful.

We start from a magic square of size m. It can be described as $m \times m$ matrix, elements of which are different integers between 1 and m^2 and

such that the sums in every row, column and two main diagonals are the same. The sum conditions are nothing else than linear equations, thus the only difficulty is to express the conditions that all elements belong to the given finite set A and are different. Let us try to express this condition in equations as well.

If $A = \{a_1, a_2, \ldots, a_n\}$ is an arbitrary finite set of different numbers, then the condition $x \in A$ is trivially expressed as the equation $p_A(x) = 0$, where

$$p_A(x) = (x - a_1)(x - a_2) \cdots (x - a_n) = x^n + A_1 x^{n-1} + ... + A_n.$$

Note that the coefficients A_k are (up to sign $(-1)^k$) elementary symmetric polynomials in $a_1, \ldots, a_n$.

If y is another element from A then, of course, $p(y) = 0$, but to express the condition $y \neq x$ we need the equation $p_2(x, y) = 0$, where

$$p_2(x, y) = \frac{p(x) - p(y)}{x - y}.$$

This already allows us to write all necessary equations for the magic square, but we prefer a shorter way to express that $\{x_1, \ldots, x_n\}$ is the set A.

Theorem 1. *The conditions*

$$\sum x_i^k = \sum a_i^k, k = 1, \ldots n.$$

are equivalent to condition that all x_i are different and belong to A.

Proof. Obviously we have the similar equality for the elementary symmetric polynomials and therefore x_i are all different solutions of the equation $p_A(x) = 0$. ■

For example, it is easy now to find all magic squares of size 3 :

x_1	x_2	x_3
x_4	x_5	x_6
x_7	x_8	x_9

Simply write

$$x_1 + \cdots + x_9 = 1 + 2 + \cdots + 9,$$
$$x_1^2 + \cdots + x_9^2 = 1^2 + 2^2 + \cdots + 9^2,$$
$$\cdots$$
$$x_1^9 + \cdots + x_9^9 = 1^9 + 2^9 + \cdots + 9^9,$$

add all sum equations

$$x_1 + x_2 + x_3 = x_4 + x_5 + x_6 = x_7 + x_8 + x_9 = x_1 + x_4 + x_7 =$$

$$x_2 + x_5 + x_8 = x_3 + x_6 + x_9 = x_1 + x_5 + x_9 = x_3 + x_5 + x_7$$

and start Gröbner basis calculations! Here is the result.

$$[{x_9}^4 - 20\,{x_9}^3 + 140\,{x_9}^2 - 400\,x_9 + 384,$$
$${x_8}^2 + 2\,x_8x_9 + 2\,{x_9}^2 - 20\,x_8 - 30\,x_9 + 115,$$
$$x_7 + x_8 + x_9 - 15, x_6 + x_8 + 2\,x_9 - 20, x_5 - 5, x_4 - x_8 - 2\,x_9 + 10,$$
$$x_3 - x_8 - x_9 + 5, x_2 + x_8 - 10, x_1 + x_9 - 10].$$

We see that we have four choices for x_9 and two for x_8 – the rest is determined uniquely. Note that $x_5 = 5$ in any magic square.

When returning to general case note that in fact some a_i could be equal – the equations still describe the set A but in this case with the multiplicities.

The next step is to obtain the Gröbner basis for the ideal I, generated by the polynomials $\sum_i x_i^k - \sum_i a_i^k$. It is not an easy task for computer for large n, thus the following result can replace the calculations.

Let $h_i(x_1, \ldots, x_k) = \sum_{|\alpha_1 + \cdots \alpha_k| = i} x_1^{\alpha_1} \cdots x_k^{\alpha_k}$ be complete symmetric functions in k variables. We put additionally $A_0 = h_0 = 1$.

Theorem 2. *The set*

$$g_k(x_1, \ldots, x_k) = \sum_{i=0}^{n-k+1} A_i h_{n+1-k-i}(x_1, \ldots, x_k)$$

for $k = 1, \ldots, n$ describes the reduced Gröbner basis of the ideal I in the lexicographical ordering $x_n > x_{n-1} > \cdots > x_1$.

Proof. First we need to show that $g_k = 0$ is valid in $K[x_1, \ldots, x_n]/I$. As usual, the easiest way to prove is to use the generating function. If we rewrite the evident equality

$$(1-tx_1)\cdots(1-tx_n) = (1-ta_1)\cdots(1-ta_n) = 1+A_1t+A_2t^2+\cdots+A_nt^n$$

as

$$(1+h_1(x_1,\ldots,x_k)t+h_2(x_1,\ldots,x_k)t^2+\cdots)(1+A_1t+A_2t^2+\cdots+A_nt^n) =$$

$$\frac{1}{(1-tx_1)\cdots(1-tx_k)}(1+A_1t+A_2t^2\cdots+A_nt^n) = (1-tx_{k+1})\cdots(1-tx_n)$$

then the coefficient with t^{n+1-k} is $g_k(x_1, \ldots, x_k)$ at the beginning and zero at the end.

Second, note that the leading monomial of g_k is x_k^{n+1-k} which gives $n!$ different solutions for the system of equations $g_k = 0, k = 1, \ldots, n$. Thus this set should be a minimal Gröbner basis and it is easy to check that this Gröbner basis is reduced as well. ■

For $n = 3$ we have

$$g_1(x_1) = x_1^3 - (a_1 + a_2 + a_3)x_1^2 + (a_1a_2 + a_1a_3 + a_2a_3)x_1 - a_1a_2a_3,$$

$$g_2(x_1, x_2) = x_1^2+x_1x_2+x_2^2-(a_1+a_2+a_3)(x_1+x_2)+a_1a_2+a_1a_3+a_2a_3 =$$

$$x_2^2 - (a_1 + a_2 + a_3 - x_1)x_2 + (a_1a_2 + a_1a_3 + a_2a_3 - (a_1 + a_2 + a_3)x_1 + x_1^2),$$

$$g_3(x_1, x_2, x_3) = x_1+x_2+x_3-(a_1+a_2+a_3) = x_3-(a_1+a_2+a_3-x_1-x_2).$$

Note that if we take the elements g_k with $k \geq l$ we get the reduced Gröbner basis for the ideal I_l, generated by polynomials $\sum_i x_i^k - \sum_i a_i^k$ with $k \leq l$. This follows from the fact that the terms of higher degrees do not influence the reduction process. Naturally, $I_1 = I$ but for $l > 1$ we have infinitely many solutions of the corresponding system.

More interesting are the remaining equations.

Theorem 3. *The condition that m different numbers $x_1, \ldots, x_m$ belong to A is expressed as a system of equations:*

$$g_k(x_1, \cdots, x_k) = 0, \ k = 1, \ldots, m.$$

Proof. We already know that the conditions are valid. It remains to note that the equations have $n(n-1)\cdots(n-m+1)$ solutions and this exactly the number of ways to choose m ordered elements from n. ■

Note that for $m = 2$ we get our familiar conditions $p_A(x_1) = 0$, $p_2(x_1, x_2) = 0$, but we do not need the condition $p_A(x_2) = 0$, which follows from them. More generally it follows from the proof that the polynomials g_k form the reduced Gröbner basis of the corresponding ideal.

If some a_i are equal, the theorem is still valid if we allow the equality of x_i up to multiplicity (e.g. if $x_i = x_j = x_k = a$, then a should appear at least three times in A). For example, if $a_1 = a_2 = 0$, $a_3 = a_4 = 1$, then our equation is $x^4-2x^3+x^2 = 0$ and the condition that $x_1, x_2, x_3 \in A$ looks as

$$x_1^4-2x_1^3+x_1^2 = 0, x_1^3+x_1^2x_2+x_1x_2^2+x_2^3-2(x_1^2+x_1x_2+x_2^2)+(x_1+x_2) = 0,$$

$$x_1^2 + x_1x_2 + x_1x_3 + x_2^2 + x_2x_3 + x_3^2 - 2(x_1 + x_2 + x_3) + 1 = 0.$$

The last equation does not allow $x_1 = x_2 = x_3 = 0$, but $x_1 = x_2 = 0$, $x_3 = 1$ is a perfect solution.

If $A = \{0, 1, \ldots, n-1\}$ then a standard way to simplify the equations (see [1]) is to replace this set by $B = \{1, \varepsilon, \ldots, \varepsilon^{n-1}\}$ with $\varepsilon^n = 1$. In this case $g_1(x_1) = x_1^n - 1$ and $g_k(x_1, \ldots, x_k) = h_k(x_1, \ldots, x_k)$ for $k > 1$.

If the size of A is not too large the equations are rather robust – we can easily create bounds δ_k such that if all $|g_k| < \delta_k$, then $|x_i - a_j| < \varepsilon$ for some j. Thus the equations have some practical applications. For large A the number of terms makes this approach impractical and the equations from Theorem 1 are probably more convenient.

It would be interesting to understand how to obtain the intersections. If B is another finite set we can create the similar equations. Together two systems of equations describe the intersection $A \bigcap B$, but it is rather unclear how these two Gröbner bases cooperate to form the Gröbner basis , which describes $A \bigcap B$. Understanding this probably could open new ways to optimize Gröbner basis calculations.

One possible application of this approach is sudoku. The experiments on sudoku examples show that the computations are much less efficient than direct combinatorial searching of the solution. Again, we need the correct interpretation of the elimination process to improve the efficiency of Gröbner basis approach.

Another remark. As we will see later, it is possible to express even more difficult conditions, e.g. $x > y$. One way to do it is to write that $x-y$ belongs to the known finite set S of differences, thus $p_S(x-y) = 0$. But what is the Gröbner basis interpretation of transitivity law:

$$x > y, y > z \Rightarrow x > z?$$

Why such trivial things are so difficult to obtain?

3 Points on the plane

Suppose now that we have a set S consisting of n different points (a_j, b_j) in the plane and want to describe the conditions that m given points $P_k = (x_k, y_k)$ belong to S. The simplest case is when we deal with real numbers. Then it is sufficient to introduce complex numbers $w_j = a_j + ib_j$ and use Theorem 3 to get necessary equations in the complex form. Of course, using their real and imaginary parts we can get the equations in the real form as well. For example, to describe that P_1, P_2 are different and belong to the set $(0,0), (0,1), (1,0), (1,1)$ we introduce first four complex numbers $w_1 = 0$, $w_2 = 1$, $w_3 = i$, $w_4 = 1 + i$. The corresponding equation having w_i as roots is

$$w^4 - (2 + 2\,i)\,w^3 + 3\,iw^2 + (1 - i)\,w = 0.$$

Thus the equations

$$z_1{}^4 - (2 + 2\,i)\,z_1{}^3 + 3\,iw^2 + (1 - i)\,z_1 = 0,$$

$$z_1{}^3 + z_1{}^2 z_2 + z_1 z_2{}^2 + z_2{}^3 - (2 + 2\,i)\left(z_1{}^2 + z_1 z_2 + z_2{}^2\right) +$$

$$+3\,i\,(z_1 + z_2) + 1 - i = 0$$

describe the situation. Converting this to real equations does not look attractive, as we already can see in the case of the first equation:

$$x_1{}^4-6\,x_1{}^2y_1{}^2+y_1{}^4-2\,x_1{}^3+6\,x_1{}^2y_1+6\,x_1y_1{}^2-2\,y_1{}^3-6\,x_1y_1+x_1+y_1=0,$$

$$6x_1y_1{}^2-6x_1{}^2y_1+y_1-2x_1{}^3-4x_1y_1{}^3+2y_1{}^3-x_1+3x_1{}^2+4x_1{}^3y_1-3y_1{}^2=0.$$

The situation is more difficult when the numbers are not real. Nevertheless in the generic case we can also find some approach, though not so obvious. As in the previous section we can easy describe the conditions that $x_1,\dots,x_m$ belong to $A=\{a_1,\dots,a_n\}$ and similarly that $y_1,\dots,y_m$ belong to $B=\{b_1,\dots,b_n\}$. The trouble is to coordinate the choices. In the generic case we have an easy solution: because all the numbers a_i+b_j are different, all that we need to say is that the numbers x_k+y_k belong to the set $C=\{a_1+b_1,a_2+b_2,\dots,a_n+b_n\}$ and we can express this according to the previous section.

We illustrate this in the following case. Suppose that the set S consists of two different points $(a,b),(c,d)$ with the "generic" coordinates. We need to describe the conditions that two given points (x,y) and (z,t) belong to S and are different. We use Theorem 1 to describe the corresponding elements in the ideal shorter. Here the first line describes the condition that coordinates belong to A and B and the last ones that $x+y$ and $z+t$ belong to C:

$$\begin{gathered}\left\{x^2+z^2-a^2-c^2,x+z-a-c,\ y^2+t^2-b^2-d^2,y+t-b-d,\right.\\ x+y+z+t-a-b-c-d,x^2+2\,xy+y^2+z^2+2\,zt\\ \left.+t^2-a^2-2\,ab-b^2-c^2-2\,cd-d^2\right\}.\end{gathered}$$

We can easily obtain Gröbner basis using the generic condition:

$$\begin{gathered}[t^2+(-b-d)\,t+bd,(-d+b)\,z+(c-a)\,t-cb+ad,\\ y+t-b-d,(-d+b)\,x+(-c+a)\,t-ab+cd].\end{gathered}$$

Note that this is a Gröbner basis so long as $b\neq d$.

In the case $b=d$ the Gröbner basis is different:

$$[t-d,z^2+(-a-c)\,z+ac,y-d,x+z-a-c],$$

but this is obviously not a generic case.

4 Small combinatorial problem

In this section we want to so consider very small combinatorial example to illustrate some ways to translate other conditions on the Gröbner basis language.

The problem is to find a word, consisting of 5 different letters A, B, C, D, E and satisfying the following conditions:

1. Exactly one consonant is written between two vowels.
2. Every vowel is placed on an odd place.
3. The letter C is placed before D, which itself is placed before A.
4. The letter B is placed before E.
5. The number of letters between C and E is odd.

No one condition looks as an equation, but we want to find the equations that equivalently describe the problem.

First of all we have a permutation of letters, which means that we can suppose that every letter has some value – its place in the word. From the first section we know how to describe this shortly:

$$A^k + B^k + C^k + D^k + E^k = 1^k + 2^k + 3^k + 4^5 + 5^k$$

for $k = 1, \ldots 5$.

The first condition now can be expressed as

$$|A - E| = 2 \Leftrightarrow (A - E)^2 = 2^2.$$

The second condition we could express using Theorem 3, but if we note that it is equivalent with the condition that the third letter is a vowel, we get a trivial equation $(A - 3)(E - 3) = 0$.

How to express the condition $D > C$ as an equation? A possible way is to say that $D - C$ belongs to the set $\{1, 2, 3, 4\}$ and this is an equation. Similarly we express the remaining conditions (note that the last one means that $|C - E| = 2$ or $|C - E| = 4$.)

Now we are ready to start Maple session to implement this. The only difficulty is that the letter D is reserved in Maple and we replace it by T. To see the result directly we use the command *solve*, that (with the help of Gröbner basis) finds the solution of the system. The last two lines we need to print our nice result using the found substitution.

$> S := \{X - T + B, Y - T + C, Z - A + T,$
$A + B + C + T + E - (1 + 2 + 3 + 4) - 5,$
$A^2 + B^2 + C^2 + T^2 + E^2 - 1^2 - 2^2 - 3^2 - 4^2 - 5^2,$
$A^3 + B^3 + C^3 + T^3 + E^3 - 1^3 - 2^3 - 3^3 - 4^3 - 5^3,$
$A^4 + B^4 + C^4 + T^4 + E^4 - 1^4 - 2^4 - 3^4 - 4^4 - 5^4,$
$A^5 + B^5 + C^5 + T^5 + E^5 - 1^5 - 2^5 - 3^5 - 4^5 - 5^5,$
$expand((A-3)*(E-3)), expand((C-E+2)^2*(C-E+4)^2),$
$expand((Y-1)*(Y-2)*(Y-3)), expand((Z-1)*(Z-2)*(Z-3)),$
$expand((X-1)*(X-2)*(X-3)*(X-4)), expand((A-E)^2-4)\}:$
$> R := solve(S);$

$$R := \{A = 5, B = 2, C = 1, E = 3, T = 4, X = 2, Y = 3, Z = 1\}$$

> f := (x, y) − > subs(R, x) < subs(R, y):
> sort([A, B, C, T, E], f);

$$[C, B, E, T, A]$$

References

[1] W. Adams and P. Loustaunau, *An Introduction to Gröbner Bases*, Amer Mathematical Society, 1994,

Victor Ufnarovski

Received June 11, 2012

Centre for Mathematical Sciences, Mathematics,
Lund University, LTH
P.O. Box 118, SE-22100, Lund, Sweden
E–mail: *ufn@maths.lth.se*

Computer Science Journal of Moldova, vol.**20**, no.3(60), 2012

References and arrow notation instead of join operation in query languages

Alexandr Savinov

Abstract

We study properties of the join operation in query languages and describe some of its major drawbacks. We provide strong arguments against using joins as a main construct for retrieving related data elements in general purpose query languages and argue for using references instead. Since conventional references are quite restrictive when applied to data modeling and query languages, we propose to use generalized references as they are defined in the concept-oriented model (COM). These references are used by two new operations, called projection and de-projection, which are denoted by right and left arrows and therefore this access method is referred to as arrow notation. We demonstrate advantages of the arrow notation in comparison to joins and argue that it makes queries simpler, more natural, easier to understand, and the whole query writing process more productive and less error-prone.

Keywords: Data modeling, query languages, concept-oriented model, join, reference, arrow notation, data semantics.

1 Introduction

The main goal of a data model is providing suitable structure for representing things and connections between them. Operations for data access and analysis are performed by means of some kind of query language which reflects and relies on these structural principles. For a general purpose data model and query language, the key problem is in finding the simplest and most natural structure and operations which

cover a wide range of patterns of thought and mechanisms being used in data modeling.

Most data models are very similar in how they represent things but they are quite different in representing connections. There exist several major ways for representing connectivity such as relationships, links, references, keys, joins. A relationship is a thing which may have its own properties and identity. Relationships can connect many things but they do not have a direction. A link is a directed binary relationship, that is, a thing that connects two other things with special roles: an origin and a destination. A reference is also a directed connection between two things but in contrast to links it is not a thing and has neither separate identity nor properties. A key is a number of properties of the thing which are used for identification purposes. Join is an operation which relies on thing properties in order to establish a connection between them at the level of queries.

One of the main motivating factors for developing the relational model [1] was the desire to get rid of (physical) identifiers and to focus on the data itself rather than on how it is represented and accessed. However, removing physical identifiers led to removing connectivity from the model. As a consequence, data was broken into several isolated sets of tuples and the question was how to retrieve related (connected) tuples. The solution was extremely simple: tuples containing the same values were supposed to be related. For example, if both an employee record and a department record have an attribute with the value 'HR' then this employee was supposed to be related to this department. The operation which finds and combines such tuples was called *join*.

Although join was introduced as one of the main operations of the relational algebra, now it is used in almost any data model so it can be characterized as a pillar of data modeling. It is one of the most frequently used words in the literature on query languages and can be found in almost any data related context. The main purpose of join consists in connecting data elements which are modeled as existing separately in different relations. It can be viewed as a means of activating implicit relationship at the level of queries. Since joins are not declared

at the level of the model, they provide almost arbitrary control over the data at query time. This property makes it very powerful operation but at the same time rather difficult to use and even dangerous for inexperienced users. In this sense, join is analogous to the goto (jump) operator in programming languages which is also a powerful low level operator providing high freedom in programming but leading to unstructured code and difficult to find errors [2].

Another wide-spread mechanism of connectivity is *reference*. One of the most important properties of references is that they are not part of the represented thing. For example, a class in object-oriented models does not describe references that will be used for representing its instances. References are not stored as part of the object in any of its fields but rather are provided separately. Another important property is that things cannot be accessed without some kind of reference. Indeed, if a property needs to be accessed then it is not possible to use another property for this purpose just because it is not accessible yet. The pattern "accessing properties using properties" obviously contains a cycle and therefore it cannot be directly implemented. Therefore, it is always necessary to have something that exists separately from and is intended to provide access to object properties. This is precisely what references are intended for. The question is only whether they are described explicitly as integral part of the model, provided by the platform as it is done in object-oriented models, or completely removed and replaced by some other mechanism like primary keys as it is done in the relational model. Essentially, the question is whether references are data and hence the model has to provide adequate means for their modeling or references are not data and should be excluded from the model.

References have numerous advantages in comparison to joins. They are extremely easy to understand because they are widely used in everyday life where all things have some unique identifiers. They are also very easy in use. It is enough to know a reference in order to get the contents of the represented thing. There is no need in specifying what and how has to be compared and what criteria have to be satisfied to access the represented thing. For example, given an employee record

we could retrieve its department by using the reference stored in one of the employee properties. The use of join operation means that a database is a set of things with common values. To access data, it is necessary to specify a criterion which has to be satisfied by all elements. For example, to get a publisher we need to specify that both the book and the publisher must have the same value in some property (publisher id). Although references are very natural and simple to use, joins are much more powerful when it is necessary to manipulate sets of elements rather than their individual instances. For this reason, it is not that easy to replace joins by references and this is why join is still dominating in the area of data modeling and query languages although it is quite difficult to use.

This paper is devoted to comparing joins and references. We demonstrate that join operation has some significant drawbacks which make it difficult to use and error-prone in comparison to references. Therefore, we ask the question whether it is possible to eliminate joins from data modeling (or at least diminish their use) by retaining most of the possibilities this operation provides. Obviously, it is a highly non-trivial task and one difficulty is that thinking of data in terms of joins is so deeply penetrated into our minds that it is considered more a dogma than one of the alternatives in data modeling and querying. Another difficulty is that join is a set-oriented operation while references are instance-oriented and this is why references are not so popular in data modeling. As a reference-based solution to the problem of joins, we describe a novel approach to data modeling, called the concept-oriented model (COM) [8, 9, 10], which generalizes references. In particular, it allows for modeling domain-specific references which replace primary keys. What is more important, COM provides two novel operations, called projection and de-projection, which can be viewed as set-oriented analogue of the classical dot notation. These two operations are denoted by left and right arrows and therefore this approach is referred to as arrow notion. We demonstrate how typical tasks can be (easier) implemented using COM references and arrow notation without using joins. The paper has the following layout. Section 2 describes the operation of join, references and arrow notation in COM. Section

3 describes drawbacks of joins and how these problems can be solved by means of COM references. Section 4 makes concluding remarks.

2 Joins and references

2.1 Joins and common value approach

In mathematics, a Cartesian product is an operation which allows us to build a new set out of a number of given sets by producing all possible combinations of their members. Given two sets U and V, the Cartesian product $U \times V$ is defined as the set of all possible 2-tuples: $U \times V = \{\langle u, v\rangle | u \in U \wedge v \in V\}$. Each element of the Cartesian product connects two input elements. Including *all* combinations of the input tuples in the result set means that all these tuples are considered related, that is, every element of one set is associated with every element of the other set.

Normally, not all input tuples are related and therefore a mechanism is needed which would allow us to restrict the Cartesian product by specifying which tuples from two sets should match. This task is performed by join operation the basic idea of which is that only those combinations of tuples are included in the result set which both satisfy some common criterion. In most practical cases, the selection of related tuples is performed by using the equality condition (this join is therefore referred to as equijoin). Tuples in the relational model are composed of values which are accessed by means of attribute names. In this case, related (matching) tuples produced by join must contain equal values in the specified attributes: $U \bowtie_{p=q} V = \{\langle u, v\rangle | u \in U \wedge v \in V \wedge u.p = v.q\}$, where p and q are attributes which have to contain the same values in both tuples.

In order to be matched, two data elements have to contain the same value in some attributes and therefore we will refer to this mechanism as a *common value approach*. Thus records which store common values are considered related in the database. For example, records from two tables `Employees` and `Departments` could be defined as related if they have the same value in the `city` attribute.

Note also that the general idea of the common value approach is also present in formal logic and deductive databases [11]. In predicate calculus, if two predicates have the same free variable then they have to match (to be bound to the same value) in order for the resulting proposition to be true. Since relations can be represented as n-place predicates, join can be written in logical form. For example, given two predicates $Employees(\#e, cname, city)$ and $Departments(\#d, ename, city)$ representing relations `Employees` and `Departments`, respectively, we can find all combinations of free variables where the matching variable *city* takes the same value.

The common value approach has the following properties:

- The relationship defined by join (via common values) does not have a direction. We simply say that two records match because they have the same property. Although some variants of join like left and right outer join have a direction, it cannot be easily semantically interpreted and should be viewed as variations of one operation. In particular, we cannot say that one record is referenced or linked to the other. In this sense, the common value approach is similar to relationships in the entity-relationship model which also do not have a direction.

- It is defined in terms of values and attribute domains, that is, a connection between two relations is specified via some common domain. There is no direct way to define join in terms of other relations. For example, we cannot directly find `Employees` and `Departments` which have the same `address` attribute which represents a record from the `Addresses` table rather than a domain. The reason is that attributes contain values and cannot contain tuples.

2.2 References and dot notation

Reference is one of the corner stones of the object-oriented paradigm where it is assumed that any object has a unique identity which is

used to represent and access it. References have the following main properties:

- References are values which are passed by-copy. It is enough to store this value in order to represent the object and then access it. When a reference is copied, the contents of the object is not copied but can be accessed later by using this reference. References do not have their own references.

- References are not object properties (not included in the object contents) and not part of the object. They exist separately from the objects they represent.

- References hide the details of object identity so that different objects may have different structure of their references which however are not visible when they are accessed.

- References provide transparent access to objects by hiding its internal mechanics which can be quite complex. They create the illusion of instantaneous access.

- References are used along with a very convenient access pattern, called dot notation, where the result of access is considered a reference which can be used for the next access operation.

References make excellent job in the area of programming but they have a rather limited use in data modeling. So what is the problem in introducing references in query languages and combining features of object-oriented and relational approaches? In fact, it is a rather old idea and almost any new query language tries to use references and dot notion to make data manipulations easier. But the fact is that they all fail in eliminating joins which means that not everything can be done by references in the area of data modeling. The primary reason (for the failure of references in data modeling) is that references and dot notation were designed to manipulate instances rather than sets. In other words, programing is an instance-oriented area while data modeling is a set-oriented area. Indeed, only individual objects can

reference each other, not sets. We cannot easily adopt dot notation for manipulating sets. Another reason is that tuples in the relational model do not have identities because any tuple is unique and identifies itself by its own contents. In the next section we describe an approach to data modeling which does not have these drawbacks.

2.3 References in the concept-oriented model

The concept-oriented data model (COM) is a unified general purpose model the main goal of which is to radically simplify data modeling by reducing a large number of existing data modeling methods to a few novel structural principles. One of its principles is that identities and entities are supposed to be equally important. This distinguishes it from most other models which have a strong bias towards modeling entities while identities (references, addresses, surrogates, OIDs) are considered secondary elements which are either modeled by means of entities or provided by the platform.

COM makes identities and entities equally important parts of a data element both being in the focus of data modeling. An element in COM is defined as consisting of two parts, identity and entity, which are also called reference and object, respectively. Identity is passed by-value while entity is passed by-reference. Both constituents have arbitrary domain-specific structure which is modeled by means of a novel construct, called concept (hence the name of the model). Concept is defined as a pair of two classes: one identity class and one entity class. For example, if employees are identified by their passport number and characterized by name then they are described by the following concept:

```
CONCEPT Employees
   IDENTITY
      CHAR(10) passNo
   ENTITY
      CHAR(64) name
```

Note that objects (entities) of this concept will have only one field and these objects will be represented by a reference (identity) also consisting of one field. However, identity part is passed by-value and

stored in variables while entity part is passed by-reference. A concept can be thought of as a conventional class with an additional class for describing the format of references.

COM provides several benefits which are important in the context of this paper:

- COM does not distinguish between sets of values and sets of objects or, in relational terms, between domains and relations. There is only one type construct, concept, which is used for defining both domains and relations. In particular, relation attributes can be both relation typed and value typed.
- Concepts make it possible to describe arbitrary domain-specific references what is not possible in object-oriented models. In this sense, references in COM are similar to primary keys in the relational model. However, the difference is that they are treated and behave like true references while primary keys are treated as integral part of the entity used for identification purposes (more about these difference can be found in [10], Section 2).

COM introduces an operation of projection which is analogous to dot notation but is applied to sets. In the concept-oriented query language (COQL) it is denoted by right arrow and returns a set of elements which are referenced by the elements from the given set. Sets in COQL are enclosed in parentheses and can also include a condition for constraining its elements. For example, all publishers for a set of books can be obtained by projecting this set of books to the set of publishers:

```
(Books | year > '2005')
  -> publisher -> (Publishers)
```

COM also introduces the opposite operation of de-projection which can be viewed as a set-oriented reversed dot notation. It is denoted by left arrow and returns a set of elements referencing the elements from the given set. For example, given a set of publishers we can get all their books:

```
(Publishers | country = 'MD')
  <- publisher <- (Books)
```

Projection and de-projection can be applied to the result set returned by the previous operation and such an approach is referred to as arrow notation. Arrow notation has the following main properties:

- Operations are applied to sets rather than instances
- It uses domain-specific instances as they are defined in concepts rather than only primitive references
- The structure of references is hidden and is not exposed in the query

In the rest of the paper we describe how these two operations are used for querying instead of joins.

3 References for solving join problems

3.1 Connectivity

Perhaps the main use of joins consists in implementing what references are intended for. A database is thought of as a set of objects referencing each other. However, if the database is unaware of references and manipulates only values then these connections have to be expressed by means of joins. For example, if each employee record references its department then a set of departments for all employees in one country is retrieved by means of the following join-based query:

```
SELECT D.name FROM Departments D, Employees E
WHERE D.dept = E.dept AND E.country = 'MD'
```

Here we immediately see one problem: join is a symmetric construct while references are directed. Indeed, if we look at the above query then it is difficult to understand whether departments reference employees or employees reference departments. It is not surprising because joins have quite different purpose but this fact makes them not very appropriate for implementing references. The mechanism of foreign and primary keys can help here but it is optional and is used at the level of schema rather than in queries.

Another problem of joins is that they expose the structure of references by explicitly specifying all the details which actually do not belong to the domain-specific part of the query. Effectively, the low level mechanics of references becomes integral and explicit part of each and every query that involves more than one table. If the structure of connections changes then all queries where it is used have to be updated. Such program logic or query fragments which are scattered throughout the whole source code are referred to as cross-cutting concern. This problem is well known in programming [4] because it makes programs difficult to maintain and error prone. Such functions as logging, transaction management, persistence and security are typical examples of cross-cutting concerns because they are used in the same form across the whole program. The main goal here is to separate these functions or query fragments from the main business logic.

Join operation is a typical example of a cross-cutting concern because many queries solving different domain specific tasks involve the same fragments in the form of join conditions. The reason is that database schemas always follow certain structure of connections and relationships while joins simply materialize them at query time. In the previous example, the schema contains two tables `Departments` and `Employees` which are connected via the join condition `D.dept=E.dept`. Note however that this join is specified along with the second condition for selecting employees of one country only. The problem is that the first condition is a cross-cutting concern because it depends on the schema structure only and will be repeated in the same form in many queries involving these two tables. The second condition reflects business logic and is unique for each query. In a good query language they should be at least separated and, ideally, the join condition should be modularized so that it does not appear in explicit form in each query. This problem can be partially solved by using a dedicated `JOIN` clause for connectivity and `WHERE` clause for domain-specific conditions. However, this use is optional and the join condition will still be repeated for each and every query.

The mechanism of foreign and primary keys could help in hiding the structure of references at the level of schema. Once a foreign key

has been declared, it is then enough to specify its name instead of enumerating all the columns it (and the corresponding primary key) is composed of. However, foreign keys do not solve the problem of joins at the level of queries because we still have to write them as some condition within `WHERE` or `JOIN` clause along with other domain-specific conditions. Another possible solution consists in defining user-defined types (UDT) in the case of complex primary keys and the corresponding foreign keys. Here again, UDTs allow us to simplify join conditions but do not eliminate them completely so that all queries have to specify how two or more tables have to be joined.

In contrast to joins, the logic of conventional references and referencing is completely hidden so that we see only what has to be retrieved and not how it has to be done. Business logic is effectively separated from the mechanism of implementing references. For example, given an employee we can get the department name by using dot notation:

```
emp.dept.name
```

Here we see neither the structure of references nor the conditions used to match the objects. References can be implemented as 64-bit integers, character strings or more complex structures. Matching related objects could be implemented via look up tables or more complex indexes but these details are also not visible in the access statement. The benefit is that if the structure of references and connections between departments and employees changes then this line of code will still work without any modifications because it does not involve any details of how employees, departments and other objects are connected.

The question is then why not to use references instead of translating them into the representation via joins? One problem is that references need identities to be explicitly declared in referenced elements and referencing attributes have to be appropriately typed. Only in this case the reference structure can be hidden. This problem can be solved by adopting the mechanism of primary keys for identification and foreign keys for typing referencing attributes. One difficulty with this solution is that primary keys are not true references (they are identifying attributes [10]) and also they are optional. A more serious problem is that references cannot be applied to sets while joins are inherently

set-oriented. Indeed, if we apply dot notation to sets then what kind of result should be returned by such expressions?

The solution is provided by introducing COM concepts. First, they provide a mechanism for defining domain-specific references which are used instead of primary keys. Once a concept has been defined, it is used as a type of attributes in other concepts by replacing the mechanism of foreign keys. Thus COM references combine features of primary keys (which are not references) and object-oriented (true) references. For example, the structure of departments and employees can be declared as follows:

```
CONCEPT Departments
   IDENTITY // True reference
      INT dept
   ENTITY
      CHAR(64) name
CONCEPT Employees
   IDENTITY // True reference
      INT emp
   ENTITY
      Departments dept
```

Note that the last line does not expose the structure of connection, that is, *how* employees are connected to departments. If the department identity changes then all other attributes referencing it will not be changed.

Concepts not only allow us to remove the structure of references from schema but also remove it from set-oriented queries by using arrow notation. For example, all departments for a set of employees in one country can be retrieved as follows:

```
(Employees | country = 'MD')
   -> dept -> (Departments)
```

This roughly corresponds to the following instance-based query using dot notation:

```
employee.dept
```

References can also be followed in the opposite direction by means

of de-projection operation. For example, all employees of a set of departments located in one country is found as follows:

```
(Departments | country = 'MD')
   <- dept <- (Employees)
```

Operations of projection and de-projection can be applied consecutively and many fragments can be omitted because they can be easily reconstructed from the schema. Thus rather complex queries involving many tables with numerous joins can be written in a very simple and natural form [6]. What is more important, these queries are set-oriented and do not expose the structure of connections.

3.2 Semantics

One problem of joins is that they appear only at the level of queries and the database is unaware of possible and meaningful joins at the level of the model. For that reason join can be characterized as an application-specific operation. Every new application can issue its own query with arbitrary joins. On one hand, it is an advantage because applications are not restricted in the use of data and can do whatever they need. However, if the meaning and consistency of results is important, it is a drawback because arbitrary joins lead to arbitrary results. The database is unaware of what operations are meaningful and therefore cannot restrict applications from producing meaningless results. For instance, the database is not able to prevent an application or user from joining integer department ids with the number of product items which is obviously a meaningless operation. From the performance point of view, it is also a disadvantage because the database engine is not able to optimize its operations for executing predefined joins declared at the level of schema.

From this point of view, joins are somewhat analogous to the goto operator in programming which also ignores the program structure and provides the possibility to organize arbitrary control flow. It was clearly shown that such style of programming without any constraints is harmful [2] because goto not only ignores the semantics behind program structure but also the compiler is not able to restrict programmers

from making errors. The freedom in using joins has the same effect: the database is not able to restrict users and applications from issuing meaningless queries and cannot restrict them from making errors. The mechanism of joins essentially assumes that the meaning of data is described at the level of queries rather than in the model structure. In particular, by looking at queries we can get more information about data semantics then by looking at the schema. One way to overcome this problem is to use foreign keys which can be viewed as a way to declare what is meaningful in the database. Yet, this mechanism has significant limitations when used in queries and should be viewed as a workaround.

Since join is a low level operation, it can be used to implement many different patterns which are difficult to reconstruct from the query. For example, the join condition `WHERE A.id=B.id` (where `A` and `B` are two tables) says almost nothing about the real intention of the query. We do not know whether table `A` references table `B` or maybe it is not about referencing at all. We do not know whether the purpose of this query is to build a multidimensional space for OLAP analysis or to find related records connected via some relationships. And if this operation uses a relationship then is it containment or general-specific? Join is not an operation which can be easily semantically interpreted. Given a join we cannot say what kind of semantic relation it represents and how the joined elements are related. On the other hand, assume that we want to use existing relationships in the model. How should we join the tables in order to represent them in the query? The answer is not clear because the translation procedure is ambiguous and does not cover all possible situations. This problem has been studied in semantic data models [3, 5] but these models focus more on conceptual representation issues and less on query languages. Although many operations can be expressed at conceptual level, joins cannot be removed completely just because the lower logical level of the model is supposed to always exist.

COM allows us to remove the gap between low level join and high level query semantics because it is also a conceptual model with main constructs having some semantics behind them. In particular, references in COM are not simply a means of connectivity but rather a way

to represent semantics. More specifically, references in COM have the following semantic interpretations [9, 10]:

General-specific A referenced element is more general than the referencing (more specific) element. For example, if table `Products` references table `Categories` then products are more specific elements than their categories.

Containment A referenced element is interpreted as a container where the referencing element exists. For example, if an employee record references a department then this employee is supposed to be included in this department as one of its elements.

Relationships An element referencing other elements is interpreted as a relationship between them. For example, if a marriage record references two persons then it is interpreted as a relationship between them.

Multidimensional An element referencing other elements is interpreted as a point while the referenced elements are its coordinates. For example, since sales record references a product item and its price, this sale is considered a point while its characteristics are coordinates along some axes.

According to this interpretations, projection operation applied to a set means getting more general elements, containing elements, dependent elements (connected via this relationship) and coordinates for these elements. And de-projection has the opposite meaning by producing more specific elements, members of a container, relationships and points with these coordinates. As a result, references are used not only for navigating through a graph but rather for semantic navigation. This makes queries much more semantically rich and much easier to write and understand. For example, projecting a set of employees to departments means getting containers for employees because a department is interpreted as a container for a set of employees. At the same time, a department can be treated as a coordinate for employees which are points in a multidimensional space.

3.3 Common value approach

There is one pattern which cannot be modeled by references, namely, the original common value approach directly supported by join operation. This pattern cannot be ignored because in many cases it is precisely what needs to be done. The common value pattern has its own value and the question is how it can be implemented by means of references without joins. For example, if it is necessary to find departments and employees having the same location then it is not clear how it can be done without join operation.

This task can be solved by using product operation which takes two or more collections as input and returns all combinations of their elements as a result collection. In COQL, input collections along with their instance variables are written in parentheses (instance variables are analogous to table aliases in SQL). For example, all combinations of departments and employees are built as follows:

```
(Departments D, Employees E)
```

If we need to return records having some common value then this condition is specified as an additional constraint:

```
(Departments D, Employees E | D.city = E.city)
```

Obviously, it is very similar to how join operation works:

```
SELECT D.*, E.* FROM Departments D, Employees E
WHERE D.city = E.city
```

So the question is why COM is better. The difference is that product in COM is used exclusively to produce combinations of elements. In particular, it is not used for referencing and navigation purposes. Its typical application is in data analysis where it is necessary to produce a multidimensional cube. For that reason, queries in COM much easier to interpret because the purpose of operations is clearer: arrow notation is used for set-based navigation while product is used to build multidimensional space with combinations of records. In other words, COM reflects the real purpose of each operation. Also, product in COM is more general because there is no difference between value domains and relations (see [10], Section 2, for more information). In particular, it is

possible to use any common collection rather than only direct domains of two relations. The following query retrieves all employees who live in the city where their department is located:

```
(Employees E | E.city = E.dept.city)
```

Here we do not use product operation at all although its relational analogue would require joining two tables. The next query finds a set of departments which have at least one employee living in a different city than this department location:

```
(Employees E | E.city != E.dept.city)
  -> dept -> (Departments)
```

Again, here we do not use product operation but still can do what would require joining in SQL.

Since product operation constrained by some common values is a quite frequent pattern, it can be simplified and generalized. Instead of explicitly specifying a condition the combined elements have to satisfy, it is easier to just specify a common greater collection for the input collections. The paths from the input collections to this common collection are then reconstructed automatically from the schema. (In the case of multiple alternative paths, the condition has to be specified explicitly.) For example, the query

```
(Departments, Employees | (Cities) )
```

returns all combinations of departments and employees which have the same city where `Cities` is their common greater collection. Note that `Cities` need not be a direct greater collection and a longer path can lead from the input collections to the `Cities` collection.

An interesting use of product operation restricted by common values consists in implementing inference which is a procedure where constraints can be automatically propagated through the model [7]. For example, assume that we want to relate departments and employees by the city they are located in. The final goal is to impose constraints on departments and then automatically find employees living in these cities (by ignoring departments people work in). Inference is always performed via some common lesser collection. In our example it is defined as a product of employees and departments with the condition

that they have to belong to the same city. Inference consists of two steps: first de-project to the common lesser collection and then project to the target collection:

```
(Departments | name = 'HR')
   <- (Departments D, Employees E | (Cities) )
   -> (Employees)
```

Note how simple and natural this query is. It specifies only collection names and has no indication how they have to be joined. Even if it is necessary to specify connections, they are specified as paths rather than explicit joins. If the schema changes and the collections will be connected differently then in many cases this query will still work.

4 Conclusion

In this paper we have provided a critical analysis of join operation and its use for data querying and retrieving related elements. Although join is an extremely powerful operation which makes it possible to dynamically (at the level of query) relate arbitrary tuples and retrieve quite complex result sets it has several major problems:

- Join is not appropriate for implementing references which is one of its main uses and one of the main data modeling mechanisms. Join exposes the details of reference implementation and is a cross-cutting concern of query languages which cannot be easily modularized.
- Join is not appropriate for representing semantics behind the higher level operation or pattern it implements. From join structure, it is quite difficult to understand what kind of relationship is used in this query. Joins do not reflect their purpose and cannot be unambiguously interpreted from the point of view of business purpose of the query.

Of course, these are not absolute flaws but rather consequences of the low level character of this operation which makes it inappropriate for domain-specific queries in general purpose query languages where

the criteria of simplicity, closeness to the domain concepts, structural and semantic consistency are of primary importance. Therefore, joins not only require high expertise but also can easily result in semantic bugs which are very difficult to find.

Data access via references and dot notation does not have the problems of join – it is more intuitive, much easier to use and more reliable. Yet, this approach is intended for manipulating instances rather than sets and therefore its benefits in the context of query languages are very limited. To overcome these limitations, we proposed to use generalized references and arrow notation as they are defined in the concept-oriented model. This new representation and access method allows us to combine set-orientation of joins with the simplicity and naturalness of references. The use of generalized references and arrow notation instead of join will result in simpler queries, more natural and structured model design, less errors and higher productivity in query writing.

References

[1] E.Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6): 377–387, 1970.

[2] E.W.Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM*, 11(3): 147–148, 1968.

[3] R.Hull, R.King. Semantic database modeling: survey, applications, and research issues. *ACM Computing Surveys (CSUR)*, 19(3): 201–260, 1987.

[4] G.Kiczales, J.Lamping, A.Mendhekar, C.Maeda, C.Lopes, J.-M.Loingtier, J.Irwin. Aspect-Oriented Programming. *ECOOP'97*, LNCS 1241: 220–242, 1997.

[5] J.Peckham, F.Maryanski. Semantic data models. *ACM Computing Surveys (CSUR)*, 20(3): 153–189, 1988.

[6] A.Savinov. Logical Navigation in the Concept-Oriented Data Model. *Journal of Conceptual Modeling*, Issue 36, 2005.

[7] A.Savinov. Query by Constraint Propagation in the Concept-Oriented Data Model. *Computer Science Journal of Moldova*, 14(2): 219–238, 2006.

[8] A.Savinov. Concept-Oriented Query Language for Data Modeling and Analysis. *Advanced Database Query Systems: Techniques, Applications and Technologies*, L.Yan, Z.Ma (Eds.), IGI Global, 2010, 85–101.

[9] A.Savinov. Concept-Oriented Model: Extending Objects with Identity, Hierarchies and Semantics. *Computer Science Journal of Moldova*, 19(3): 254–287, 2011.

[10] A.Savinov. Concept-Oriented Model: Classes, Hierarchies and References Revisited. *Journal of Emerging Trends in Computing and Information Sciences*, 3(4): 456–470, 2012.

[11] J.D.Ullman, C.Zaniolo. Deductive databases: achievements and future directions. *ACM SIGMOD Record*, 19(4): 75–82. 1990.

Alexandr Savinov,

Received June 28, 2012

SAP Research Dresden,
SAP AG
Chemnitzer Str. 48,
01187 Dresden, Germany
E–mail: *alexandr.savinov@sap.com*
Home page: *http://conceptoriented.org/savinov*

Computer Science Journal of Moldova, vol.**20**, no.3(60), 2012

Basics of Intensionalized Data: Presets, Sets, and Nominats

Mykola Nikitchenko, Alexey Chentsov

Abstract

In the paper we consider intensional aspects of the notion of data. We advocate an idea that traditional set-theoretic platform should be enhanced with new data structures having explicit intensional component. Among such data we distinguish the notions of preset and nominat. Intuitively, presets may be considered as collections of "black boxes", nominats may be considered as collections of "grey boxes" in which "white boxes" are names and "black boxes" are their values, while sets may be treated as collections of "white boxes". We describe intensions and properties of the introduced notions. We define operations over such data as functions computable in a special intensionalized sense.

Keywords: Set theory, alternative set theories, notion intension, intensionality, presets, nominats, computability, intensionalized computability.

1 Introduction

Formal methods of software development require precise specifications of the system under construction. Such specifications are usually grounded on set-theoretic platform [1]. For example, well-known B Method [2] and Z Notation [3] declare that they are based on Zermelo-Fraenkel set theory (ZF theory).

The set-theoretic platform is understandable, elaborated, and powerful formalism for describing systems and investigating their properties. Its expressive power is confirmed by the fact that main parts of mathematics can be presented in a unified form within set theory

[1]. But at the same time this power is often excessive and cumbrous. Therefore there were various attempts to restrict classical set theory or even to construct alternative set theories. These attempts were inspired both by immanent development of set theory and by its application for problem domains. Some of these proposals will be considered in section 5 devoted to related work.

Our approach for constructing modified set theory aims to support the software development process which usually starts from abstract system specification and proceeds to concrete implementation. At the abstract levels many system components are described only partially thus objects under investigation are underdetermined. In this case many conventional properties of sets may fail. In particular this concerns the extensionality principle. Recall that this principle is supported by the very first axiom of set theory – the extensionality axiom: two sets are equal if they consist of the same elements [1]. But now we can see more and more facts when a pure extensional orientation becomes restrictive for further development of computer science, artificial intelligence, knowledge bases, and other disciplines dealing with the notions of data, information, and knowledge. Therefore it seems reasonable to enhance extensional definitions of the notion of set and its derivatives (such as *data* and *function*) with *intensional* components. In a broad sense the intension of a notion means properties which specify that notion, and the extension means objects which fall under the notion, i.e. have the properties specified by the notion intension. The *intension/extension* dichotomy was studied primarily in logic, semiotics, and linguistics; we advocate more active investigations of this dichotomy in computer science too. In this paper we continue our investigations on intensionality of basic computer science notions initiated in [4]. Being oriented on computer science, we are inspired by mathematical constructivism with its emphasis on finiteness of objects and constructions. Therefore we restrict our considerations to 1) intensionalized data with finite structure, and 2) computable (in the intensionalized sense) operations over such data.

The rest of the paper is structured in the following way. In section 2 we introduce the general idea of intensionalized data and intuitively

define intensions of objects which can be considered as collections of elements. In section 3 intensions (properties and operations) of special collections called presets, sets, and nominats, are described. In section 4 formal definitions of intensions of collections that have finite structure are given. Based on these definitions, special computability of function over intensionalized data with finite structure is defined; computable functions over presets, sets, and nominates are specified. Section 5 is devoted to related work. In conclusions we summarize obtained results and discuss directions for future work.

2 Intensionalized data

Considering computer science notions in integrity of their intensional and extensional aspects we obtain new possibilities to define more first-level notions as basic notions of mathematical formalisms. Here we will focus on the notion of data trying to transform set theory to a *theory of intensionalized data*. Such data can be considered as certain objects with prescribed intensions. This idea is similar to the notion of typed data, but the latter is usually understood in the extensional sense while we aim to emphasize intensional features of data. The first steps in developing the notion of intensionalized data were made in [5, 6].

The main difficulty in constructing theories of intensionalized data is concerned with the definition of data intension. We start with intuitive understanding of intensions, and then construct their formal explications. We will move from abstract understanding of data to their more concrete representations.

At the most abstract level of consideration data are understood as some objects. Objects can be considered as *unstructured* (as *wholes* with intension I_W) or as *structured* (with *parts*, intension I_P). An object with the intension I_W can be regarded as a "black box" (intuitively it means that nothing is "visible", and therefore nothing is known about the object, intension I_{WB}) or as a "white box" (everything is "visible" and recognizable, intension I_{WW}). An intermediate intension is denoted by I_{WBW} ("black" or "white box").

To come to richer intensions we should treat objects as structured

(with intension I_P). We start with simple structures: all parts of an object are identified and fixed. In this case each part can be regarded as a whole. Relations within the object are also identified and fixed. The above specification of object structure permits to call it *hard structure*. Thus, we divide intension I_P into two subintensions I_{PH} and I_{PS} specifying objects with hard and soft structures respectively.

We continue with I_{PH} concretization caused by possible relationships between object parts. Such relationships are classified along the line *tight–loose*. Loose relationships mean that parts are not connected with each other (intension I_{PHL}); tight relations mean that parts are connected (intension I_{PHT}). In this paper we will primarily consider objects with intension I_{PHL}. In this case such objects are called *collections*; their parts are called *elements*. Empty collection is denoted in a traditional way as $\emptyset$.

Considering elements as unstructured wholes, we can treat them with intensions of "black" and/or "white boxes". Thus, three new intensions stem from this: I_{PHLB}, I_{PHLW}, and I_{PHLBW}.

Objects with intension I_{PHLB} should be regarded as collections of "black boxes". Such objects we call *presets*. Collections of "white boxes" (intension I_{PHLW}) are called *explicit multisets*; if repetition of elements is not allowed then we obtain *explicit sets*. Collections with intension I_{PHLBW} contain "black" and "white" elements (*mixed presets*).

A collection of playing cards is a good example for the introduced notions. Each playing card has two sides: the face and the back. Normally, the backs of the cards should be indistinguishable (identical). As to the faces of the cards, they may all be unique, or there can be duplicates. If all cards of a collection are placed face down on the table (are "black boxes"), then such collection is a preset. If some cards are exposed (placed face up on the table) while others are not exposed (placed face down), then we obtain a mixed preset. If all cards are exposed (are "white boxes"), then we get an explicit multiset, or a set if duplicates are not allowed.

We will make here one more concretization of intension I_{PHLB}. Under this concretization we treat each element as constructed of a

"white box" and "black box". The "white box" is considered as a name of the "black box"; thus, the "black box" is the value of this name. We call such collections *nominats* (from Latin *nomen* – name) and denote a corresponding intension as I_{ND}.

A good example of nominats is a collection of addressed envelopes. The address ("white box") written on an envelope may be considered as a name of the letter ("black box") inserted (placed) into the envelope.

Nominats are a special case of nominative data [7]. It is important to admit that nominative data can model the majority of data structures used in computer science [7, 8].

Thus, we propose to introduce additionally to the notion of set the above specified notions: presets, mixed presets, and nominats as the basic mathematical notions. These notions are enriched with intensional components and are non-extensional. Please also note that these notions are related with each other, say, sets and nominates can be treated as concretizations of presets.

To realize the idea of introducing these notions we have to describe their intensions in more detail.

3 Intensions of presets, sets, and nominats

Data intensions specify properties of corresponding data. Operations over such data should be defined in such a way that they use only those possibilities that are prescribed by the intension. In this paper we introduce the notions of "weak" operation, operation with copying, and "strong" operation. For weak operations it is allowed to construct the result of these operations using only those data components that are present in the input data; for operations with copying it is also allowed to make copies of existing components; and for strong operations it is additionally possible to generate new components. For example, the card game players are not allowed to make copies of cards or generate new cards; thus, they must use only weak operations. In computer science we also meet situations when we usually do not have possibilities to copy existing objects (say, for hardware components) or have such possibilities (say, for software components) or even have tools to

produce new objects. These situations correspond to weak operations, operations with copying, and strong operations respectively.

3.1 Preset intension

Intuitively, presets can be understood as collections of externally undistinguishable objects (elements) which have hidden content.

One more example of presets is a collection of tickets of an instant lottery. The surfaces of tickets should be covered by opaque material making them "black boxes" that hide the content of tickets.

Having this example in mind we can specify our understanding of presets by the following intuitive properties:

- each element of a preset is some whole;
- elements are separated from one another;
- elements are independent of one another, i.e., close relations between them are absent;
- all elements "are available", i.e., each element can be obtained for processing;
- exhaustive processing of all elements of a preset is possible;
- elements do not vary until it is explicitly mentioned (the law of identity of elements).

Let us admit that these properties are very weak and do not specify membership relation, so, given a preset and an element, it is not possible to say whether this element belongs to the preset. Also the equality relation is not specified. It is possible to have many hidden equal elements (duplicates) in a preset, thus, extensionality axiom is not valid. These properties of presets have a negative character restricting possibilities for processing of presets. But what operations for preset processing are available?

Analysis of the above formulated properties leads to the conclusion that the following operations are allowed for presets with the intension I_{PHLB}:

- union $\cup$ which given presets pr_1 and pr_2 yields a new preset consisting of elements of pr_1 and pr_2 ;
- nondeterministic choice *ch* which given a preset *pr* yields some element *e* of *pr*;
- nondeterministic choice with deletion *chd* which given a preset *pr* yields some element *e* of *pr* and a preset pr' without this element;
- empty function $\bar{\emptyset}$ which given a preset *pr* yields an empty preset $\emptyset$;
- cardinality operation *card* which given a preset *pr* yields the number of elements in *pr*.

The above defined operations *conform to the intension* (respect the intesion) I_{PHLB} (are *preset-conforming* operations). It means that during their execution these operations will not require additional information hidden in "black boxes" thus they use only that information which is prescribed by the intension. According to this, the intersection of presets is not available, contrary to set theory.

Still, the idea of a preset says that elements contain some hidden content; therefore operations working with this content are also required. The most natural of such operations is *open* operation. Given a preset *pr* this operation constructs a multiset *ms* which consists of "white box" elements that are content of the elements of the initial preset. We use multisets here because cardinality of *pr* and *ms* are to be the same. It means that duplicates should be preserved. The *open* operation does not conform to the intension I_{PHLB} because it opens "black boxes". Therefore, theory of presets should contain two parts: one part describes operations that conform to the intension I_{PHLB} while the other part specifies more powerful operations which can change intensions of preset elements.

3.2 Set intension

The notion of set can be considered as the "final" concretization of the notion of preset. The main new feature of sets is that their ele-

ments are considered as "white boxes", thus no hidden information is present. From this follows that elements are "recognizable" and can be compared upon distinction and equality. Therefore, to the previously formulated properties of presets (to the preset intension) we add the following new property:

- each element of a set is "recognizable" and can be checked upon distinction and equality with any other element.

Usually this property is formalized via set membership relation $\in$. From this follows that we additionally have new operations for set processing, for example, intersection and difference of sets. Still, the powerset operation will be not considered here as it should have possibility to construct copies of elements.

Set intension I_{PHLW} will also be denoted as I_{ST}. As the notion of set is well studied we will not go further into detail of set properties and operations.

3.3 Nominat intension

Intuitively, a nominat can be considered as a concretization of a preset in which each element consists of "white box" and "black box". To make this abstract consideration more concrete we should involve practical observations which permit to say that the "white box" can be considered as a *name* of the "black box"; and their relation is a *naming* (*nominative*) relation. In Slavic languages the term 'nominat' has two different meanings: a naming expression or a value of such expression; thus our treatment unites these meanings, because nominat is a unity of names and values. Nominats are also called *flat nominative data* [7].

Nominats have the dual nature: first, they may be considered as certain collections of elements; second, they may be considered as functions due to relation that connects names and their values.

Traditionally, notations of functional style are chosen to represent nominats. For example, a nominat with names $v_1, \ldots, v_n$ and values $a_1, \ldots, a_n$ respectively, is denoted by $[v_1 \mapsto a_1, \ldots, v_n \mapsto a_n]$. If values themselves are nominats, then we get the notion of *hierarchic nominats*

(*hierarchic nominative data*); for example

$$[v_1 \mapsto [u_1 \mapsto b_1, \ldots, u_k \mapsto b_k], \ldots, v_n \mapsto [t_1 \mapsto c_1, \ldots, t_m \mapsto c_m]]$$

is a 2-level nominat.

It is important to admit that nominats can model the majority of data structures used in computer science [7]. For example, a set $\{e_1, \ldots, e_m\}$ can be represented as $[1 \mapsto e_1, \ldots, 1 \mapsto e_m]$, where 1 is a standard name which has different values $e_1, \ldots, e_m$; a tuple $(e_1, \ldots, e_m)$ can be represented as $[1 \mapsto e_1, \ldots, m \mapsto e_m]$ with $1, \ldots, m$ as standard names; a sequence $\langle e_1, \ldots, e_m \rangle$ can be represented as $[1 \mapsto e_1, 2 \mapsto [\ldots, 2 \mapsto [1 \mapsto e_m, 2 \mapsto \emptyset] \ldots]]$, where 1, 2 are standard names.

The main new operations over nominats are the following:

- *naming* $\Rightarrow v$ (with name $v \in V$ as a parameter) which given a value a yields a nominat $[v \mapsto a]$;
- *denaming* $v\Rightarrow$ (partial multivalued operation with name $v \in V$ as a parameter) which given a nominat d yields a value of v in d if it exists;
- *checking* $v!$ (with name $v \in V$ as a parameter) which given a nominat d yields d if the value of v exists in d; or yields $\emptyset$ if such a value does not exist;
- *overriding* ∇ which given two nominats d_1 and d_2 yields a new nominat d consisting of named values of d_2 and those of d_1, the names of which do not occur in d_2.

These operations *conform to the intension* I_{ND} (are *nominat-conforming* operations). Thus, these operations are allowed for nominats processing.

Now we will describe briefly the distinctions between the notions of set and nominat as mathematical primitives. To do this, various criteria can be used. First, nominats, contrary to sets, have hidden content. This permits to make their further concretizations not possible

for sets. Second, nominats have functional "spirit" of naming relation simplifying nominat processing. We will illustrate this statement by the following observations. We start with the notion of ordered pair (a, b) that can be defined as nominat $[1 \mapsto a, 2 \mapsto b]$ where 1 and 2 are standard names. The notion of ordered pair in set theory has many definitions:

- $(a, b) = \{\{\{a\}, \emptyset\}, \{\{b\}\}\}$ – Norbert Wiener, 1914;
- $(a, b) = \{\{a, 1\}, \{b, 2\}\}$ – Felix Hausdorff, 1914 (1 and 2 are two distinct objects different from a and b);
- $(a, b) = \{\{a\}, \{a, b\}\}$ – Kuratowski, 1921;
- etc.

It seems that these definitions do not look fully adequate to the intuitive notion of ordered pair, because they require detailed analysis of bracket structure (Wiener's definition), or are restrictive (Hausdorff's definition), or collapse to singleton $\{\{a\}\}$ when $a = b$ (Kuratovski's definition). It is interesting to admit that in *Principia Mathematica* the notion of ordered pair was considered as primitive, and even N. Bourbaki took the same position. So, introduction of special primitives like ordered pairs (and nominats in our case) is not a new idea.

Concerning further relationships of ordered pairs and tuples with nominats, we would like to emphasize that nominats are more adequate to computer science practice than tuples. To make this claim more understandable, let us consider questions of operating with tuples and nominats. Indeed, given two tuples $(a_1, \ldots, a_m)$ and $(b_1, \ldots, b_n)$ we can combine them practically only as concatenation $(a_1, \ldots, a_m, b_1, \ldots, b_n)$ or $(b_1, \ldots, b_n, a_1, \ldots, a_m)$. But concatenation is a coarse operation that ignores possible coincidence of some values from $\{a_1, \ldots, a_m, b_1, \ldots, b_n\}$ representing the same attributes. Thus, we are forced to make finer combinations of $(a_1, \ldots, a_m)$ and $(b_1, \ldots, b_n)$ manually that complicates processing of such data. Instead of this data structure (tuples) we propose to consider nominats. In this case we have more natural combining operations, for example, given nominats $[x \mapsto 7, y \mapsto 5, z \mapsto 8]$ and

$[t \mapsto 7, u \mapsto 5, x \mapsto 8]$ we obtain $[y \mapsto 5, z \mapsto 8, t \mapsto 7, u \mapsto 5, x \mapsto 8]$ as their overriding combination (cf. with combination of tuples $(7, 5, 8)$ and $(7, 5, 8)$). Also, other combining operations can be defined. This richness of combining operations simplifies processing of nominats compared with tuples. The reason of this is that the abstraction level of "position" in a tuple is lower than that of "name" in a nominat since position depends more strongly on other positions than a name depends upon other names. Thus, operating with names (with nominats) is more "soft" with respect to data transformations. The above considerations shortly argue in favour of using nominats as one more basic data structure in computer science.

Properties of intensionalized data and operations over them were discussed in this section informally. To make the proposed approach more precise we need formal definitions of these notions.

4 Formal definitions of intensionalized data

To give formal definitions of intensionalized data we will use reduction methods. Roughly speaking it means that given data class D with intension I_D, we construct a reduction procedure to some data class D' that has an understandable and well studied intension. Also, operations over D will be reduced to operations over D'. In our case we will use several reduction steps.

Still, this idea is difficult to be realized if no restrictions are imposed on intension I_D. Taking into consideration that computer science is the intended application domain for intensionalized data, we restrict ourselves to data having finite structures (intension I_{PHF}) and to operations that are computable in a special intensionalized sense. Note that this intension is subintension of I_{PH}; thus, data with intension I_{PHF} can have loose relations between their components (intension I_{PHL}), or can have tight relations (intension I_{PHT}), for example, in finite lists their components are tightly related.

In the sequel we will use the following notations for classes of functions from D to D':

- $D \xrightarrow{p} D'$ – the class of partial single-valued functions;
- $D \xrightarrow{b} D'$ – the class of total single-valued bijective functions;
- $D \xrightarrow{m} D'$ – the class of partial multi-valued functions. Function f is *multi-valued* (non-deterministic) if being applied to the same input data d it can yield different results during different applications to d (and possibly be also undefined);
- $D \xrightarrow{t} D'$ – the class of total functions. Function f is total if the value f on d is always defined;
- $D \xrightarrow{i} D'$ – the class of injective functions. A multi-valued function is *injective*, if it yields different values on different arguments. The inverse of injective function is a single-valued function;
- $D \xrightarrow{\nu} D'$ – the class of total multi-valued injective functions.

4.1 Intensionalized data with finite structures

Let D be a class of data with intension I_D. Assume that we treat data of D as finite structured data. Our intuitive understanding of such a data is the following: any such data d consists of several basic (atomic) components $b_1, \ldots, b_m$, organised (connected) in a certain way. If there are enumerably many different forms of organisation, each of these data can be represented in the (possibly non-unique) form $(k, \langle b_1, \ldots, b_m \rangle)$, where k is the *data code* and the sequence $\langle b_1, \ldots, b_m \rangle$ is the *data base*. Data of this form are called *natural data* [9]. More precisely, if B is any class and Nat is the set of natural numbers, then the class of *natural data* over B is the class $Nat(B) = Nat \times B^*$. An implicit assumption is that the code represents 1) all information that can be "extracted" from those elements of B which are contained in d, and 2) interrelations between such elements. (This will be discussed in more detail in the next subsection.) These properties specify a fixed intension of natural data: they have the form $(k, \langle b_1, \ldots, b_m \rangle)$ where k is a natural number and $\langle b_1, \ldots, b_m \rangle$ is a list of elements treated as "black boxes". As finite

structured data can have different representations, we should use total multi-valued injective functions for constructing such representations.

Note that we use the term 'class' for collections of intensionalized data; term 'set' is used for collections, the intensions of which are subintensions of sets.

Now we are ready to give the formal definition of a class of intensionalized data with some intension I_D which is a subintension of I_{PHF}. A class D is called a class *of finite structured data*, if a class B and a total multi-valued injective mapping $nat\colon D \xrightarrow{\nu} Nat(B)$ are given. This mapping *nat* is called the *naturalization* mapping. Naturalization mapping is actually an *analysing* mapping: it finds in a data d its components and their interrelations according to the properties of data prescribed by its intension. Dually to *nat* we introduce *denaturalization* mapping *denat* which reconstructs (*synthesizes*) data of class D from natural data. For simplicity's sake we assume that $denat = nat^{-1}$. Denaturalization mapping is a partial single-valued mapping. Naturalization and denaturalization mapping are also called *concretization* and *abstraction* mappings respectively.

Example 1 (naturalization mapping for a class B of basic elements)**.** As nothing is known about elements of B, we treat such elements as "black boxes"; therefore B is a preset with intension I_{PHLB}. Thus, we define $nat_B\colon B \xrightarrow{t} Nat(B)$ to be such mapping that $nat_B(b) = (0, \langle b \rangle)$ for any b of B. It means that nothing is known about b (its code is 0) and b has no parts except itself (its base is $\langle b \rangle$).

Example 2 (naturalization mapping for the set Nat of natural numbers)**.** These numbers are treated as "white boxes" without parts. Thus, we define $nat_{Nat}\colon Nat \xrightarrow{t} Nat(B)$ to be such mapping that $nat_{Nat}(n) = (n, \langle\rangle)$ for any n of Nat. It means that n is known (its code is n) and n has no parts (its base is empty sequence $\langle\rangle$).

Example 3 (naturalization mapping for an enumerated set S)**.** The set S is considered as enumerated set (has the intension of enumerated set) if a bijective mapping $u\colon Nat \xrightarrow{b} S$ is given. In this case we define

$nat_S\colon S \xrightarrow{t} Nat(B)$ to be such mapping that $nat_S(e) = (u^{-1}(e), \langle\rangle)$ for any e of S.

Example 4 (naturalization mapping for the class B^* of finite sequences over preset B)**.** For any element $\langle b_1, \ldots, b_n\rangle$ of B^* we know its structure (which is a list of length n), but we know nothing about elements of B. Thus, we define $nat_{B^*}\colon B^* \xrightarrow{t} Nat(B)$ to be such mapping that $nat_{B^*}(\langle b_1, \ldots, b_n\rangle) = (n, \langle b_1, \ldots, b_n\rangle)$.

The above given definitions may be considered as a special formal definition of intension I_D: given a finite structured data class D a pair (B, nat) is called *naturalized intension* of D; a tuple $(D, (B, nat))$ is called a *naturalized class of intensionalized data*.

Still, these definitions which reduce intuitive understanding of data of D to $Nat(B)$ lack precise description of their intensions because we did not define operations over D and over $Nat(B)$; in other words, we do not have complete description of the intensions of these classes. As mentioned earlier, we are oriented on mathematical constructivism, thus, we will treat operations over D and over $Nat(B)$ as computable in a special sense. Computabillity considered here is called weak natural computability.

4.2 Weak natural computability over intensionalized data

To formalize operations that conform to data intensions we will use a special computability called *intensionalized* computability. This computability will be reduced in several steps to traditional computability of n-ary functions defined on integers or strings. Traditional computability may be called Turing computability. In the light of our investigations traditional computability does not pay much attention to the variety of data intensions, because it concentrates on computability over integers (or strings) which have fixed intensions.

The idea behind intensionalized computability is the following: for data processing it is allowed to use only those operations that conform

to their intensions. Thus, intensionalized computability is *intensionally restricted computability*. In fact, such computability is a *relative computability* – relative to data intensions.

Defining this computability we follow [5] with several modifications: 1) we define computability for functions of the type $D \xrightarrow{m} D'$ instead of $D \xrightarrow{m} D$, 2) we consider weak computability (without copying) instead of computability with copying.

Introduction of naturalization mapping is a crucial moment for defining intensionalized computability. This mapping is regarded as a formalization of data intension; and this enables us to explicate an intuitive notion of intensionalized computability over D with intension I_D via formally defined *weak natural computability* over D. The latter is then reduced to a new special computability over $Nat(B)$ called *weak code computability*. To define this type of computability we should recall that natural data has a fixed intension under which the code collects all known information about data components and their interrelations, and the base is treated as a list of "black boxes". Thus, weak code computability should be independent of any specific manipulation (processing) operations of the elements of B and can use only information that is explicitly exposed in the natural data. The only explicit information is the data code and the length of the data base. Therefore in code computability the data code plays a major role, while the elements of the data base virtually do not affect the computations. These elements may be only used to form the base of the resulting data. To describe the code of the resulting data and the order in which elements of the initial base are put into the base of resulting data, a special function of type $Nat^2 \xrightarrow{m} Nat \times Nat^*$ should be defined. Such a function is called *weak index-computable*. These considerations lead to the following definition.

A multi-valued function $g\colon Nat(B) \xrightarrow{m} Nat(B)$ is called weak code-computable if there exists a weak index-computable multi-valued function $h\colon Nat^2 \xrightarrow{m} Nat \times Nat^*$ such that for any k, m from Nat, $b_1, \ldots, b_m$ from B, $m \geq 0$, we have $g(k, \langle b_1, \ldots, b_m \rangle) = (k', \langle b_{i_1}, ..., b_{i_l} \rangle)$ if and only if $h(k, m) = (k', \langle i_1, \ldots, i_l \rangle)$, $1 \leq i_1 \leq m$, $\ldots$, $1 \leq i_l \leq m$, $l \geq 0$, and all indexes $i_1, \ldots, i_l$ are distinct. If one of the indexes

$i_1, \ldots, i_l$ lies outside the interval $[1, m]$, or there are equal indexes in the sequence $i_1, \ldots, i_l$, or $h(k, m)$ is undefined, then $g(k, \langle b_1, \ldots, b_m \rangle)$ is also undefined.

In other words, in order to compute g on $(k, \langle b_1, \ldots, b_m \rangle)$, we have to compute h on (k, m), generate a certain value $(k', \langle i_1, \ldots, i_l \rangle)$, and then try to form the value of the function g by selecting the components of the sequence $\langle b_1, \ldots, b_m \rangle$ pointed to by the indexes $i_1, \ldots, i_l$.

This definition actually completes our formalization of the natural data intension because it specifies operations over natural data as weak code-computable.

Note that weak computability defined here differs from the computability with copying defined in [4, 5] by the requirement that all evaluated indexes should be distinct.

It is clear that index computability of $h\colon Nat^2 \xrightarrow{m} Nat \times Nat^*$ may be reduced by traditional methods of recursion theory to conventional computability of a certain function $r\colon Nat \xrightarrow{m} Nat$.

We are ready now to give the formal definition of a weak natural computable function.

Let $(D, (B, nat))$ and $(D', (B, nat'))$ be naturalized classes of intensionalized data (w.l.o.g. we treat these classes as based on one class B). A function $f\colon D \xrightarrow{m} D'$ is called *weak natural computable* (with respect to naturalized intensions (B, nat) and (B, nat')) if there is a weak code-computable function $g\colon Nat(B) \xrightarrow{m} Nat(B)$ such that $f = denat' \circ g \circ nat$.

This definition completes our formalization of the data intension of the class D because it gives possibility to formalize operations over D as weak natural computable.

Thus, intensionalized computability has been defined via a sequence of the following reductions: intensionalized computability – weak natural computability – weak code computability – weak index computability – partial recursive computability. Analysing the definitions we can also conclude that weak natural computability is a generalization (relativization) of enumeration computability. In fact, for $B = \emptyset$ weak code computability is reduced to partial recursive computability on Nat, and weak natural computability is reduced to enumeration com-

putability [10]. Therefore, the notions of weak code and weak natural computability defined above are quite rich.

In the sequel weak natural computability will also be denoted as wn-computability.

Example 5 (wn-computability over preset B)**.** The naturalization mapping nat_B was defined in Example 1. To define the complete class of wn-computable functions over $(B, (B, nat_B))$ of type $B \xrightarrow{m} B$, we have to describe all weak index-computable function of the type $h\colon Nat^2 \xrightarrow{m} Nat \times Nat^*$. It is easy to understand that under the naturalization mapping nat_B we need to know the results of weak index-computable function only on the element $(0, 1)$. On this input data a weak index-computable function can 1) yield $(0, \langle 1 \rangle)$, 2) yield a value distinct from $(0, \langle 1 \rangle)$, or 3) be undefined. For cases 2) and 3) the denaturalization mapping will be undefined. This induces the following functions of type $B \xrightarrow{m} B$: 1) the identity function *id*, 2) the everywhere undefined function *und*, and 3) the multi-valued (non-deterministic) function *und-id* such that *und-id*(d) is equal to d or is undefined. Actually it means that the following result was proved: *the complete class of weak natural computable partial multi-valued functions over preset B consists of functions und, id, and und-id.* In other words, the three functions defined above are the only computable functions over "black box" intensionalized data.

Example 6 (wn-computability over the set Nat of natural numbers)**.** The naturalization mapping nat_{Nat} was defined in Example 2. Under this naturalization we are interested in weak index-computable functions defined on the sets of elements of the form $(n, 0)$. This set is isomorphic to Nat. Thus (as expected), the set of all wn-computable functions over Nat is exactly the set of all partial recursive functions.

Example 7 (wn-computability over the enumerated set S)**.** The naturalization mapping nat_S was defined in Example 3. Under this naturalization we are again interested in weak index-computable functions defined on the sets of elements of the form $(n, 0)$, $n \in Nat$. This set is isomorphic to Nat. Thus (as expected), the wn-computability over S coincides with the enumeration computability over S [10].

Example 8 (wn-computability over the class B^*)**.** The naturalization mapping nat_{B^*} was defined in Example 4. Under this naturalization we are again interested in weak-index computable functions defined on the sets of elements of the form (n, n) with results of the form $(k, \langle i_1, \cdots, i_k\rangle)$, where $k \in Nat$, $k \leq n$, $1 \leq i_1 \leq n$, ..., $1 \leq i_k \leq n$, and all indexes $i_1, \cdots, i_k$ are distinct. One among such functions is a function h_{tail} such that $h_{tail}(n, n) = (n-1, \langle 2, \ldots, n\rangle)$. It means that *tail* operation (such that $tail(\langle b_1, \ldots, b_n\rangle) = \langle b_2, \ldots, b_n\rangle$, $n > 0$) is wn-computable. Note that doubling operation $doubl(\langle b_1, \ldots, b_n\rangle) = (\langle b_1, \ldots, b_n, b_1, \ldots, b_n\rangle)$ is not wn-computable.

Having defined the notion of natural computability, we can now check whether operations over intensionalized data (presets, sets, and nominats) intuitively defined in the previous section indeed conform to the corresponding intensions.

As domains and ranges of operations can be constructed with the help of Cartesian product, now we will give definition of the intension for such a product. Let $(D_1, (nat_1, B))$, ..., $(D_n, (nat_n, B))$ be naturalized classes of intensionalized data. Then naturalization mapping $nat_{D_1\times\cdots\times D_n}\colon D_1 \times \cdots \times D_n \xrightarrow{\nu} Nat(B)$ is defined as follows (d_1 is of D_1, ..., d_n is of D_n):

$$\begin{aligned} nat_{D_1\times\cdots\times D_n}(d_1, \ldots, d_n) = \\ (c(n, c(c(k_1, l_1), c(\ldots c(c(k_{n-1}, l_{n-1}), c(k_n, l_n))\ldots)))), \\ \langle b_{11}, \ldots, b_{1l_1}, \ldots, b_{n1}, \ldots, b_{nl_n}\rangle), \end{aligned}$$

where $nat_j(d_j) = (k_j, \langle b_{j1}, \ldots, b_{jl_j}\rangle)$, $1 \leq j \leq n$; c is a pairing function that uniquely encodes two natural numbers into a single natural number, say, the Cantor pairing function.

The idea behind this definition is simple: given a tuple $(d_1, \ldots, d_n)$ we first find naturalizations $nat_j(d_j) = (k_j, \langle b_{j1}, \ldots, b_{jl_j}\rangle)$, then construct the code of the resulting natural data by encoding codes and lengths of tuple components, and at last we construct the base by concatenating components' bases.

4.3 Computability of preset operations

First, we should define naturalization mapping for presets. Let B be a class of elements and $PreF(B)$ be a class of finite presets with elements of B. Naturalization mapping $nat_{PS}\colon PreF(B) \xrightarrow{\nu} Nat(B)$ is defined as follows: given a preset pr with elements $e_1, \ldots, e_n$ function nat_{PS} on pr can yield any natural data of the form $(n, \langle e_{i_1}, \ldots, e_{i_n}\rangle)$, where $e_{i_1}, \ldots, e_{i_n}$ is a permutation of $e_1, \ldots, e_n$.

Example 9 (wn-computability of choice function $ch\colon PreF(B) \xrightarrow{m} B$)**.** The naturalization mapping nat_{PS} was defined in this section and the mapping nat_B was defined in Example 1. For choice operation ch a weak index-computable multi-valued function $h_{ch} : Nat^2 \xrightarrow{m} Nat \times Nat^*$ is defined by the formula: $h_{ch}(n, n) = (0, \langle i\rangle)$, where $1 \leq i \leq n$. This function is obviously Turing computable; therefore ch is wn-computable.

Example 10 (wn-computability of union $\cup\colon PreF(B)^2 \xrightarrow{t} PreF(B)$)**.** Let preset pr_1 of $PreF(B)$ consists of elements $b_1, \ldots, b_n$ and preset pr_2 of $PreF(B)$ consists of elements $e_1, \ldots, e_m$; $nat_{PS}(pr_1) = (n, \langle b_{i_1}, \ldots, b_{i_n}\rangle)$ and $nat_{PS}(pr_2) = (m, \langle e_{j_1}, \ldots, e_{j_m}\rangle)$, where b_{i_1}, $\ldots$, b_{i_n} and e_{j_1}, $\ldots$, e_{j_m} are permutations of b_1, $\ldots$, b_n and $e_1, \ldots, e_m$ respectively. According to the definition of naturalization of Cartesian product, we obtain the following natural data for the pair (pr_1, pr_2): $(c(2, c(c(n, n), c(m, m))), \langle b_{i_1}, \ldots, b_{i_n}, e_{j_1}, \ldots, e_{j_m}\rangle)$. Index-computable function $h_\cup$ such that

$$h_\cup(c(2, c(c(n, n), c(m, m))), n + m) = (n + m, \langle 1, 2, \ldots, n + m\rangle)$$

is partial recursive and determines wn-computable binary function. It is clear that the result does not depend on permutations of b_1, $\ldots$, b_n and $e_1, \ldots, e_m$, thus obtained union function is single-valued.

In the same way we can prove that other operations over presets defined in section 3 are computable. Thus, the following statement is valid.

Proposition 1. *The following operations over presets: union* $\cup$*, choice ch, nondeterministic choice with deletion chd, empty function* $\bar{\emptyset}$*, and cardinality card, are weak natural computable.*

It means that these operations conform to preset intension.

Actually, using such techniques we can formally describe all preset-conforming operations of different types. For example, any preset-conforming operation *op* of type $PreF(B) \xrightarrow{m} Nat$ can be represented as a composition of a certain multi-valued partial recursive function $na\colon Nat \xrightarrow{m} Nat$ and a cardinality operation *card*, thus, $op = na \circ card$.

We can also prove that some operations, say, intersection of two presets, are not preset-conforming operations.

4.4 Computability of set operations

Set intensions assume that elements of sets are "white boxes". The naturalization approach requires that such elements can be encoded. It means that we should consider B as an enumerated set $B = \{b_0, b_1, \dots\}$. Thus, bijective enumeration function $u : Nat \xrightarrow{b} B$ is given. Let $SetF(B)$ be a class of finite sets with elements of enumerated set B.

We can define two naturalization mappings: weak and strong.

The weak naturalization mapping $nat_{SFW} : SetF(B) \xrightarrow{\nu} Nat(B)$ is defined as follows: given a set s with elements $e_1, \dots, e_n$, mapping nat_{SF} on s, can yield any natural data of the form $(k, \langle e_{i_1}, \dots, e_{i_n}\rangle)$, where $k = c(n, c(k_1, \dots c(k_n, 0)\dots))$, $u(k_1) = e_{i_1}, \dots, u(k_n) = e_{i_n}$; and $e_{i_1}, \dots, e_{i_n}$ is a permutation of $e_1, \dots, e_n$.

The idea behind this definition is very simple: we encode the cardinality of s and numbers of its elements according to the enumeration function; as to the base we include in it all elements of s. (The definition may be simpler if we take into account ordering of elements induced by enumeration function, cf. with definitions in the next subsection.)

The strong naturalization mapping $nat_{SFS}\colon SetF(B) \xrightarrow{\nu} Nat(B)$ is defined as follows: given a set s with elements $e_1, \dots, e_n$, mapping nat_{SFS} on s, can yield any natural data of the form $(k, \langle\rangle)$, where

$k = c(n, c(k_1, \dots c(k_n, 0) \dots))$, $u(k_1) = e_{i_1}, \dots, u(k_n) = e_{i_n}$; and $e_{i_1}, \dots, e_{i_n}$ is a permutation of $e_1, \dots, e_n$. The base of the obtained natural data is empty.

The difference between these naturalization concerns the possibility of producing new elements. In the first case this is not allowed because code-computable functions construct the base of result using only the base of initial natural data. In the second case we can evaluate any code and then (using denaturalization mapping) produce any elements of S.

These naturalization mappings define different intensions of the class $SetF(B)$.

Example 11 (wn-computability of intersection $\cap$: $SetF(B)^2 \xrightarrow{t} SetF(B)$)**.** Let set s_1 of $SetF(B)$ consists of elements $b_1, \dots, b_n$ and set s_2 consists of elements $e_1, \dots, e_m$; $nat_{SFW}(s_1) = (q_1, \langle b_{i_1}, \dots, b_{i_n} \rangle)$ and $nat_{SFW}(s_2) = (q_2, \langle e_{j_1}, \dots, e_{j_m} \rangle)$, where $q_1 = c(n, c(k_1, \dots c(k_n, 0) \dots))$, $u(k_1) = b_{i_1}, \dots, u(k_n) = b_{i_n}$; $q_2 = c(m, c(r_1, \dots c(r_m, 0) \dots))$, $u(r_1) = e_{j_1}, \dots, u(r_m) = e_{j_m}$; $b_{i_1}, \dots, b_{i_n}$ and $e_{j_1}, \dots, e_{j_m}$ are permutations of $b_1, \dots, b_n$ and $e_1, \dots, e_m$ respectively. According to the definition of naturalization of Cartesian product, we obtain the following natural data for the pair (s_1, s_2):

$$(c(2, c(c(q_1, n), c(q_2, m))), \langle b_{i_1}, \dots, b_{i_n}, e_{j_1}, \dots, e_{j_m} \rangle).$$

How to define an index-computable function $h_\cap$? The following algorithm can be proposed. First, the code $c(2, c(c(q_1, n), c(q_2, m)))$ should be analyzed and all pairs (k_i, r_j) such that $k_i = r_j$ should be identified. Then a list of their positions (say, in s_1) should be formed (this list is a list of indexes in the result of index-computable function). At last, a code of the result should be evaluated; in this code we include the numbers (under naturalization mapping) of the elements of the intersections and its cardinality. Defined function is partial recursive, the results do not depend on permutations of the elements of the initial sets. So, intersection is wn-computable.

In the same way we can prove that conventional operations over sets are computable. Thus, the following statement is valid.

Proposition 2. *The following operations over sets: union $\cup$, intersection $\cap$, difference $\setminus$, choice ch, nondeterministic choice with deletion chd, empty function $\bar{\emptyset}$, and cardinality card are weak natural computable.*

So, we have proved that these operations conform to set intension. But some operations over sets, say powerset operation, are not wn-computable. Still, this operation is natural computable with copying.

4.5 Computability of nominat operations

Nominat intensions assume that elements of a nominat are constructed of names ("white boxes") and their values ("black boxes"). Thus, naturalization mapping of nominats is constructed of naturalizations for names and values. We assume that the set of names $V = \{v_0, v_1, \dots\}$ is enumerated by bijective enumeration function u (see the previous subsection). It is also reasonable to choose a strong naturalization mapping nat_{SFS} because normally any name can be generated. As to values, we assume that they are elements of a preset (with intension I_{PHLB}).

Let $NomF(V, B)$ be a class of finite nominats constructed over V and B and $nm = [v_1 \mapsto b_1, \dots, v_n \mapsto b_n]$ be a nominat of this class. W.l.o.g. we can assume that names are ordered according to their numbers with respect to enumeration mapping, that is $u^{-1}(v_1) < u^{-1}(v_2) < \dots < u^{-1}(v_n)$. Under this assumption weak (with respect to B) naturalization mapping $nat_{NMW} \colon NomF(V, B) \xrightarrow{t} Nat(B)$ is defined as follows: given a nominat $nm = [v_1 \mapsto b_1, \dots, v_n \mapsto b_n]$ mapping nat_{NMW} on nm yields natural data of the form $(k, \langle b_1, \dots, b_n\rangle)$, where $k = c(n, c(k_1, \dots c(k_n, 0) \dots))$, $u(k_1) = v_1, \dots, u(k_n) = v_n$.

Example 12 (wn-computability of denaming $v\Rightarrow \colon NomF(V, B) \xrightarrow{m} B$)**.** The naturalization nat_{NMW} has been defined just now and the mapping nat_B was defined in Example 1. For denaming function $v \Rightarrow$ a weak index-computable multi-valued function $h_{v\Rightarrow} \colon Nat^2 \xrightarrow{m} Nat \times Nat^*$ is defined by the formula: $h_{v\Rightarrow}(c(n, c(k_1, \dots c(k_n, 0) \dots)), n) = (0, \langle k_i\rangle)$, where $1 \le i \le n$, $u(k_i) = v$; in other cases the value is

undefined. This function is obviously Turing computable; therefore $v\Rightarrow$ is wn-computable.

In the same way we can prove that other operations over nominats defined in the previous section are computable. Thus, the following statement is valid.

Proposition 3. *The following operations over nominats: naming $\Rightarrow v$, denaming $v\Rightarrow$, checking $v!$, and overriding ∇ are weak natural computable.*

As to computability with copying, in [4, 9] several theorems were proved that may be considered as descriptions of complete classes of natural computable (with copying) functions over various kinds of intensionalized data, and hierarchic nominats, in particular.

Summing up, we can say that proposed naturalization approach permits to define preset-, set-, and nominat-conforming operations (for finite collections), thus giving possibility for further development of the theory of intensionalized data.

5 Related work

The notion of data, being one the main notion of computer science, has many aspects, definitions, and explications. The analysis of such diversity of data concepts is worth a special investigation the authors plan to fulfill in forthcoming papers. In this paper, oriented on enhancement of the notion of set, we will consider only those works that are related to set theory variations.

Set theory, being a primary foundation for mathematical research, has been debated for decades. Paradoxes, controversies and inconsistencies with mathematical practice in some areas have led to multiplicity of set theories as well as rise of quite uncommon alternative theories. The approaches used by different "schools of thoughts" can be classified by many criteria like extensionality, kind of logic employed, intensionality, finiteness, well-foundedness, characteristics of membership relation, predicativity, incompleteness of knowledge, information

hiding, etc. Most "radical" departures from standard set theory concern base logic. Less radical ones modify or reject some principle of ZFC through system of axioms or more informally.

We start with variations caused by set theory paradoxes. If U is a set-theoretic universe then it should satisfy equation that can be stated in the abstract form as $\mathcal{P}^*(U) \simeq U$. In order to avoid paradoxes, $\mathcal{P}^*(U)$ cannot be powerset of U but rather collection of some distinguished subsets of U. The solution of this equation $\mathcal{U} = \langle U, f\rangle$, where $f\colon U \simeq \mathcal{P}^*(U)$, is called Frege structure. It determines abstract set-theoretic universe where membership relation is interpreted as follows: $u \in_{\mathcal{U}} v$ iff $u \in f(v)$.

Conventional remedy to paradoxes was in limitation of the cardinality of the sets. This limitation was quite restrictive turning ZF theory (with axiom of foundation) into the theory of small and iterative sets [11]. Some alternative theories do not reject ZF completely but rather look for extensions of ZF that avoid paradoxes by other means than limitation of size.

In [12] class of subsets $\mathcal{P}^*(U)$ is selected from topological considerations to be either open or closed subsets of topological space U. Moreover bijection in this case can be required to be homeomorphism.

A few alternatives (in order to avoid Russell's paradox) are based on modification of the concept of (co-)extension. Formalizing notion of 'partial information' in [13] a concept of partial set was proposed. Though partial set extension and coextension are disjoint, they do not necessarily cover the universe. The theory of partial sets introduces new primitive operators $\notin$, $\neq$. Construction of sets and abstraction axioms are allowed only for formulas without negations – positive formulas. Extensionality principle cannot be used to identify partial sets (it is possible to express positively negative properties). Intensionality can be used instead implying some sort of set naming and pure term models [14].

Positive sets can be seen as simplification of partial sets (though have their own motivation) [15]. In this case operators $\notin$, $\neq$ and abstractors are dropped while extensionality is restored. This theory has models known as 'hyperuniverses' constructed using topological set-

theoretic structures [16]. [17, 18] studied the first-order generalization of positive sets theory known as GPK^+_∞. In this theory additionally axiom of infinity and existence of least set that contains "extension" for given (arbitrary) formula (closure principle) are postulated. This theory disproves axiom of choice and class of its hereditary well-founded sets interprets ZF. Some peculiar constructions are possible in GPK^+ models like self containing singleton (auto-singleton) [19].

Paradoxical set theory is another consistent theory without extensionality. It is dual to theory of partial sets. In it set extension and coextension are not necessarily disjoint but cover the universe [20]. Analogously set theory HF (Hyper-Frege) is counterpart to the GPK^+ [21]. Its models are built on the same bases as GPK. Stronger theory HF_∞ (with axiom of infinity) is capable of interpreting ZF.

In double extension set theory to avoid classical paradoxes the concept of extension was bifurcated [22]. There are two membership relations $\in$, $\in'$. Extensionality axiom for this theory is formulated as follows: $\forall z(z \in x \leftrightarrow z \in' y) \rightarrow x = y$. Some analog of infinite ordinal is possible to construct in this theory without explicitly stating axiom of infinity. Also it is possible to interpret ZF in some form in the theory [23]. Serious shortcoming of this theory is lack of proof of its consistency.

Rough set theory [24] presumes incomplete knowledge which is formalized using equivalence relation of indiscernibility. Based on this approach, [11] proposed generalized Proximal Frege Structures which are universes of sets with additional modal operators. This gives prospects for axiomatic modal set theory.

Another line of research is related to category theory. Category theory emphasizes external properties of objects. Concept of morphism or function is abstract and primitive in category theory and is not reduced to sets. Typically objects of a category are instances of the structure of certain kind, and morphisms are structure-preserving functions [25, 26]. Structure of objects and properties of morphisms are described in terms of other objects and morphisms only.

Sets together with functions between sets form a category. It is possible to give purely category-theoretic characterization to this cat-

egory which leads to a concept of elementary topos. Toposes can be provided with internal language which is very similar to that of set theory and can be interpreted inside the topos in category-theoretic terms [27]. Thus topos may be regarded as a mathematical domain of discourse or "world" in which mathematical concepts can be interpreted and mathematical constructions performed [28]. This idea was further developed in local set theory [28].

Frege structure can be considered in categorical framework. In Heyting categories (some generalization of toposes) it is possible to introduce the notion of "smallness" defining sets. If such category has a powerclass functor of subsets then its free algebras are models of set theory [29, 30]. Membership relation in these models is determined algebraically. Field, known as algebraic set theory, researches some aspects of set theory through these models. Primarily intuitionistic ZF theory is targeted. But models of other set theories can be constructed by the same algebraic method simply varying particular category and notion of "smallness".

As a contrary Lawere advocates that set theory should not be based on membership but rather on isomorphism-invariant structures. He proposed an Elementary Theory of the Category of Sets (ETCS) for this purpose [31, 32]. Objects of ETCS are abstract sets. In short, abstract set is an assemblage of featureless but distinct "dots". From technical standpoint ETCS is non-degenerate well-pointed topos with natural numbers object for which axiom of choice holds. It is argued that strong case can be made for ETCS logical and conceptual autonomy [33].

Martin-Löf type theory emphasizes constructivity [34]. It follows Curry-Howard correspondence to represent propositions as sets thus interpreting predicate logic. Sets also can be seen as problem descriptions. The equality between sets is intensional which means it is definitional or syntactical. Theory has formal language that is used as programming language, specification language and programming logic. Axiom of choice is provable in Martin-Löf type theory [35] while in constructive or intuitionistic set theory it implies the law of excluded middle.

The admissible set theory [36] aims to present a weaker axiomatic system more adequate for processing of finite domains. Additionally this theory includes basic elements (praelements).

Now we would like to say a few words about the term 'preset'. Probably Bishop [37] was the first who introduced this term. Toby Bartels explains that for Bishop a preset is like a set without an equality relation; conversely, a set is a preset equipped with an equality relation. This understanding stems from Bishop's three steps definition of a set: you should first state how to construct an element of the set; then you should describe how to prove that two elements are equal; and at last you should prove that this (equality) relation is reflexive, symmetric, and transitive. If you only do the first step, then you don't have a set, according to Bishop; you only have a preset. A given preset may define many different sets, depending on the equality relation. From this follows that a membership relation is defined for Bishop's presets, but extensionality axiom fails. Thus, our understanding of presets is weaker and different from Bishop's treatment.

Such numerous examples (of course, not exhaustive) of set theory variations give good evidence that many scientists are aware of restrictedness of traditional set theory. We argue for intensional approach to constructing set theory variants. We also emphasize constructiveness of such variants through explicit computability aspects.

Summing up, we would like to admit that the proposed notions of preset and nominat differ from the conventional notion of set in several aspects: from the one side, theories of presets and nominats are weaker than conventional set theory, in particular, extensionality fails, also membership relation and equality are not definable; but on the other hand, these notions seem to be more adequate to computer science domain because operations are defined as computable in a special intensionalized sense, presets and nominats are constructed over basic elements (praelements) which may have hidden content, from this stems a possibility to change levels of abstraction of data consideration (up to non-wellfoundedness). Still, investigation on the topic should be continued in order to establish more precise relations between theories under investigations.

6 Conclusions

Set theory is the main formal system that is used for construction of problem domain models. Being well-developed and studied, it gives a powerful mathematical instrument for investigations of models constructed on the set-theoretic platform. But at the same time more and more examples demonstrate that in certain cases set theory is not adequate to problem domain formalization especially when only partial information about domain is available. The reason of this inadequacy lies in the fundamentals of set theory, in particular, in membership relation and extensionality principle. For problem domains with incomplete information a membership relation cannot be defined, also the extensionality principle fails. We propose to consider a weaker "set" theory with explicit intensional component. Such a theory may be called theory of intensionalized data. The first-level notions of this theory are notions of preset, set, and nominat. Presets may be considered as collections of "black boxes", sets may be treated as collections of "white boxes", and nominats are collections of "grey boxes" in which "white boxes" are names and "black boxes" are their values. In the paper we have defined these notions and described their main properties. Being oriented on mathematical constructivism we have defined operations over such data as computable in a special intensionalized sense. Obtained computability has been called weak natural computability. It has been defined via several steps of reduction to conventional Turing computability.

The results presented in the paper can be considered as the initial steps in developing the theory of intensionalized data.

In the forthcoming papers we plan to construct complete classes of weak/strong natural computable functions over classes with different intensions and demonstrate how these notions can be used for describing intensionalized semantics of specification languages and program logics.

References

[1] N. Bourbaki. *Theory of Sets.* Berlin: Springer-Verlag, 2004.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[3] J.M. Spivey. *The Z Notation: A Reference Manual*, 2nd ed. Prentice Hall, 1992.

[4] N.S. Nikitchenko. *Intensional aspects of the notion of program.* Problems of Programming, No. 3–4 (2001), pp. 5–13. [In Russian]

[5] M.S. Nikitchenko. *Gnoseology-based Approach to Foundations of Informatics.* In: Ermolayev, V. et al. (eds.) Proc. 7-th Int. Conf. ICTERI 2011, Kherson, Ukraine, May 4-7, 2011, CEUR-WS.org/Vol-716, ISSN 1613-0073, pp. 27–40.

[6] M.S. Nikitchenko. *Intensional aspects of main mathematical notions.* In: Contemporary problems of mathematics, mechanics and computing sciences: N.N. Kizilova, G.N. Zholtkevych (eds). Kharkov: Apostrophe Publ. (2011), pp. 183–191.

[7] N.S. Nikitchenko. *A Composition-nominative approach to program semantics.* Technical Report IT-TR 1998-020, Technical University of Denmark, ISSN 1396-1608, 1998.

[8] I.A. Basarab, N.S. Nikitchenko, V.N. Redko. *Composition Databases.* Kiev: Lybid Publ., 1992. [In Russian]

[9] N.S. Nikitchenko. *Abstract computability of non-deterministic programs over various data structures.* In: Perspectives of System Informatics. LNCS, vol. 2244, Berlin: Springer (2001), pp. 471–484.

[10] Yu. L. Ershov. *Enumeration Theory.* Nauka Publ., Moscow, 1977. [In Russian]

[11] P. Apostoli, R. Hinnion, A. Kanda, T. Libert. *Alternative set theories.* In: Philosophy of Mathematics: Irvine A.D. (ed.). Elsevier (2009), pp. 461–491.

[12] O. Esser and T. Libert. *On topological set theory.* Mathematical Logic Quarterly, vol. 51 (2005), pp. 263–273.

[13] P. C. Gilmore. *The consistency of partial set theory without extensionality.* In: Axiomatic Set Theory: Jech, Th., (ed.). American Mathematical Society (1974), pp. 147–153.

[14] R. Hinnion. *Intensional solutions to the identity problem for partial sets.* Reports on Mathematical Logic, 42 (2007), pp. 47–69

[15] R.J. Malitz. *Set theory in which the axiom of foundation fails.* Ph.D. thesis, UCLA, 1976.

[16] M. Forti, R. Hinnion. *The consistency problem for positive comprehension principles.* Journal of Symbolic Logic, 54 (1989), pp. 1401–1418.

[17] O. Esser. *On the consistency of a positive theory.* Mathematical Logic Quarterly, 45, No. 1 (1999), pp. 105–116.

[18] O. Esser. *Une theorie positive des ensembles.* Cahiers du Centre de Logique, 13, Academia-Bruylant, Louvain-la-Neuve (Belgium), 2004.

[19] R. Hinnion. *Stratified and positive comprehension seen as superclass rules over ordinary set theory.* Zeitschrift für mathematische Logik und Grundlagen der Mathematik, 36 (1990), pp. 519–534.

[20] M. Crabbé. *Soyons positifs: la complétude de la théorie näive des ensembles.* Cahiers du Centre de Logique, vol. 7 (1992), pp. 51–68.

[21] T. Libert. *ZF and the axiom of choice in some paraconsistent set theories.* Logic and Logical Philosophy, vol. 11 (2003), pp. 91–114.

[22] A. Kisielewicz. *Double extension set theory.* Reports on Mathematical Logic, 23 (1989), pp. 81–89.

[23] M. Holmes. *The structure of the ordinals and the interpretation of ZF in double extension set theory.* Studia Logica, vol. 79 (2005), pp. 357–372.

[24] Z. Pawlak. *Rough sets*. International Journal of Computer and Information Sciences, vol. 11, No. 5 (1982), pp. 341–356.

[25] S. Awodey. *Category theory*. Oxford: Clarendon Press, 2006.

[26] J. Goguen. *A categorical manifesto*. Mathematical Structures in Computer Science, 1 (1991), pp. 49–67.

[27] P. Johnstone. *Topos theory*. London Mathematical Society Monographs, vol. 10, Academic Press, London, New York, San Francisco, 1977.

[28] J. L. Bell. *Toposes and local set theories: An introduction*. Oxford: Clarendon Press, 1988.

[29] A. Joyal and I. Moerdijk. *Algebraic Set Theory*. Cambridge University Press, 1995.

[30] S. Awodey. *A brief introduction to algebraic set theory*. Bulletin of Symbolic Logic, 14, No. 3 (2008), pp. 281–298.

[31] F. W. Lawvere, R. Rosebrugh. *Sets for Mathematics*. Cambridge University Press, 2003.

[32] J. L. Bell. *Abstract and Variable Sets in Category Theory*. In: What is Category Theory? Polimetrica Publisher, Italy (2006), pp. 9–16.

[33] Ø. Linnebo, R. Pettigrew. *Category Theory as an Autonomous Foundation*. Philosophia Mathematica, vol. 19, No. 3 (2011), pp. 227–254.

[34] B. Nordström, K. Petersson, J. M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[35] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[36] J.Barwise. *Admissible sets and structures*. Perspectives in Mathematical Logic, Volume 7. Berlin: Springer-Verlag, 1975.

[37] E. Bishop. *Foundations of Constructive Analysis*. New York: McGraw-Hill, 1967.

Mykola Nikitchenko, Alexey Chentsov, Received July 5, 2012

Mykola Nikitchenko
Taras Shevchenko National University of Kyiv
01601, Kyiv, Volodymyrska st, 60
Phone: +38044 2590519
E–mail: *nikitchenko@unicyb.kiev.ua*

Alexey Chentsov
Taras Shevchenko National University of Kyiv
01601, Kyiv, Volodymyrska st, 60
Phone: +38044 2590511
E–mail: *chentsov@ukr.net*

Computer Science Journal of Moldova, vol.**20**, no.3(60), 2012

P systems based on tag operations

Yurii Rogozhin Sergey Verlan

Abstract

In this article we introduce P systems using Post's tag operation on strings. We show that the computational completeness can be achieved even if the deletion length is equal to one.

1 Introduction

The tag operation was invented by E. Post during his Procter fellowship at Princeton during the academic year 1920-21 [12, 13]. This operation deletes first n letters of a word and appends an appendant depending on the first deleted letter. Computational devices based on this operation, *the tag systems*, are one of the simplest examples of universal devices [8, 3]. The number of deleted symbols, *the deletion number*, permits to establish a frontier between decidability and undecidability – if it is equal to two, then the corresponding class is undecidable, while if it is equal to one, then the corresponding class is decidable. There exist other interesting properties of tag systems, we refer to [7] for a review on the recent results in this field.

P systems [10, 11] are distributed computational devices inspired from the structure and the functioning of a living cell. The cell is considered as a set of compartments (membranes) nested one in another and which contain objects and evolution rules. The base model does not specify neither the nature of these objects, nor the nature of rules. Numerous variants specify these two parameters by obtaining a lot of different models of computing, see [15] for a comprehensive bibliography.

In the case of P systems with tag operations the basic objects are strings and the operations in membranes are tag operations. In a formal

way, an n-tag P systems can be considered like a graph, whose nodes contain sets of strings and sets of tag rules with the deletion number n. Every rule permits to perform a tag operation and to send the result to some other node. Such an approach is close to the idea of graph-controlled or programmed grammars, where a similar control mechanism is used, but for rewriting rules. We show that using P systems permits to strictly increase the power of the tag operation and to achieve the universality with the deletion number equal to one.

2 Definitions

In this section we recall some very basic notions and notations we use throughout the paper. We assume the reader to be familiar with the basics of formal language theory. For more details, we refer to [14].

A *tag system* of degree $m > 0$, see [3] and [9], is the triplet $T = (m, V, P)$, where $V = \{a_1, \ldots, a_{n+1}\}$ is an alphabet and where P is a set of productions (tag operations) of form $a_i \to P_i, 1 \leq i \leq n$, $P_i \in V^*$. We remark that for every a_i, $1 \leq i \leq n$, there is exactly one production in P. The value m is also called the deletion number of T. The symbol a_{n+1} is called the halting symbol. A configuration of the system T is a word w. The application of the tag operation permits to pass from a configuration $w = a_{i_1} \ldots a_{i_m} w'$ to the next configuration z by erasing the first m symbols of w and by adding P_{i_1} to the end of the word: $w \Longrightarrow z$, if $z = w' P_{i_1}$.

The computation of T over the word $x \in V^*$ is a sequence of configurations $x \Longrightarrow \ldots \Longrightarrow y$, where either $y = a_{n+1} a_{i_1} \ldots a_{i_{m-1}} y'$, or $y' = y$ and $|y'| < m$. In this case we say that T halts on x and that y' is the result of the computation of T over x. We say that T recognizes the language L if there exist a recursive coding ϕ such that for all $x \in L$, T halts on $\phi(x)$, and T halts only on words from $\phi(L)$.

We note that tag systems of degree 2 are able to recognize the family of recursively enumerable languages [3, 9]. Moreover, the construction in [3] has non-empty productions and halts only by reaching the symbol a_{n+1} in the first position. It is also known that tag systems of degree 1 are decidable [6, 16]. It thus follows that the deletion number m is

one decidability criterion [5] for tag systems with $m = 2$ as the frontier value.

Now we introduce the notion of the circular Post machine (CPM).

Definition 1 *A* circular Post machine *(of type 0) is a tuple* $(\Sigma, Q, \mathbf{q}_1, \mathbf{q}_f, R)$ *with a finite alphabet* Σ *where* $0 \in \Sigma$ *is the blank, a finite set of states* Q*, the initial state* $\mathbf{q}_1 \in Q$*, the final state* $\mathbf{q}_f \in Q$*, and a finite set of instructions* R *with all instructions having one of the forms* $\mathbf{p}x \to \mathbf{q}$ *(erasing the symbol read by deleting a symbol),* $\mathbf{p}x \to y\mathbf{q}$ *(overwriting and moving to the right),* $\mathbf{p}0 \to y\mathbf{q}0$ *(overwriting and inserting a blank symbol), where* $x, y \in \Sigma$ *and* $\mathbf{p}, \mathbf{q} \in Q$*,* $\mathbf{p} \neq \mathbf{q}_f$*.*

We also refer to all instructions with $\mathbf{q}_f$ in the right hand side as halt instructions. The storage of this machine is a circular tape, the read and write head moves only in one direction (to the right), and with the possibility to delete a cell or to create and insert a new cell with a blank.

Notice that a circular tape can be thought of as a finite string of symbols (from the one following the state to the one preceding the state in the circular representation). In this way, CPM0 is a finite-state machine, which reads the leftmost symbol of the string, possibly consuming it, and uses the symbol+state information to change the state, possibly writing a symbol on the right.

There are several other variants of CPM [4, 1] which differ in the way the lengthening instructions work. All these variants are computationally equivalent, although their descriptional complexity can be different.

Now we define P systems that use the tag operation.

An *n-tag P system* is the construct

$$\Pi = (O, T, \mu, M_1, \cdots, M_n, R_1, \cdots, R_n), \text{ where}$$

- O is a finite alphabet,
- $T \subseteq O$ is the terminal alphabet,

- μ is the membrane (tree) structure of the system which has n membranes (nodes) and it can be represented by a word over the alphabet of correctly nested marked parentheses,
- M_i, for each $1 \leq i \leq n$ is a finite language associated to the membrane i,
- R_i, for each $1 \leq i \leq n$ is a set of rules associated to membrane i, of the following forms: $a \to P_a; tar$, $a \in O$ where $a \to P_a$ is a tag rule and tar is the *target indicator* from the set $\{here, in_j, out \mid 1 \leq j \leq n\}$, where j is a label of the immediately inner membrane of membrane i.

An n-tuple $(N_1, \cdots, N_n)$ of finite languages over O is called a configuration of Π. The transition between the configurations consists of applying the tag rules (with the deletion length n) in parallel to all possible strings, non-deterministically, and following the target indications associated with the rules.

More specifically, if $w = aa_2 \dots a_n w' \in N_i$ and $r = a \to P_a; tar$ then the word $w' P_a$ will go to the region indicated by tar. If $tar = here$, then the string remains in N_i, if $tar = out$, then the string is moved to the region immediately outside the membrane i (maybe, in this way the string leaves the system), if $tar = in_j, j = 1, ..., n$, then the string is moved to the immediately below j-th region.

A sequence of transitions between configurations of a given insertion-deletion P system Π, starting from the initial configuration $(M_1, \dots, M_n)$, is called a computation with respect to Π. The result of a computation consists of all strings over T which are sent out of the system at any time during the computation. We denote by $L(\Pi)$ the language of all strings of this type. We say that $L(\Pi)$ is generated by Π.

We denote by $ELSP_k(n - tag)$ the family of languages $L(\Pi)$ generated by n-tag P systems with $k \geq 1$ membranes.

3 Results

Theorem 1 *Any CPM0 M can be simulated by a 1-tag P system.*

Proof. Consider a CPM0 $M = (\Sigma, Q, q_1, q_f, R)$ with symbols $\Sigma = \{a_j \mid 0 \leq j \leq n\}$, where $a_0 = 0$ is the blank symbol, and states $Q = \{q_i \mid 1 \leq i \leq f\}$, where q_1 is the initial state and the only terminal state is $q_f \in Q$; let $Q' = Q \setminus \{q_f\}$.

Consider the following 1-tag P system

$$\Pi = (V, \Sigma, \mu, M_{m_s}, \ldots, M_{m_f}, R_{i_s}, \ldots, R_{i_f}) :$$

$$\begin{aligned} V &= \Sigma \cup Q, \\ \mu &= [\prod_{q_i a_j \in Q \times \Sigma} \Big([\ ]_{m_{ij}} \Big)]_{m_s}\ , \\ M_i &= \emptyset,\ i \neq m_s, \text{ and the rules are given and explained below.} \end{aligned}$$

Hence the membrane structure of Π consists of the skin membrane m_s and inner membranes $m_{ij},\ 1 \leq i \leq f, 0 \leq j \leq n$. The set of rules is defined as follows:

$$\begin{aligned} R_{m_s} &= \{1.ij : q_i \to \varepsilon; m_{ij} \mid 1 \leq i \leq f-1,\ 0 \leq j \leq n\} \\ &\cup \{2.j : a_j \to a_j; here \mid a_j \in \Sigma\} \\ &\cup \{3 : q_f \to \varepsilon; out\}, \\ R_{m_{ij}} &= \{4.ij : a_j \to a_k q_l; out \mid q_i a_j \to a_k q_l \in R, j > 0\} \\ &\cup \{5.ij : a_j \to q_l; out \mid q_i a_j \to q_l \in R, j > 0\} \\ &\cup \{6.i : a_0 \to a_k q_l a_0; out \mid q_i a_0 \to a_k q_l a_0 \in R\}. \end{aligned}$$

A configuration $v = q_i a_j W$ of M describes that M in state $q_i \in Q$ considers symbol $a_j \in \Sigma$ to the left of $W \in \Sigma^*$. This configuration is encoded by the string v in the skin membrane m_s of Π.

The machine M starts a computation from a configuration $q_1 a_j W$ and Π starts computation from the corresponding string $q_1 a_j W$ in membrane m_s (other regions of Π are empty). We shall show now how the rules of M are simulated in Π.

Consider rule $\mathbf{q_i a_j \to a_k q_l} \in \mathbf{R}, q_i \in Q', q_l \in Q, a_j, a_k \in \Sigma$ of M. It is simulated in Π as follows.

Let $q_i a_j W \overset{q_i a_j \to a_k q_l}{\Longrightarrow} q_l W$ be a computation step in M, i.e., rule $q_i a_j \to a_k q_l$ is applied to configuration $q_i a_j W$ yielding $q_l W a_k$ ($W \in \Sigma^*$).

This rule is simulated in Π as follows. One of rules $1.ip$ is nondeterministically applied to string $q_i a_j W$ and the resulting string $a_j W$ moves to region m_{ip}. We denote this action as follows:

$$(m_s, q_i a_j W) \overset{1.ip}{\Longrightarrow} (m_{ip}, a_j W).$$

If $p \neq j$, then the corresponding string cannot evolve anymore as there is no applicable rule in membrane m_{ip}. If $p = j$, then the following evolution is possible yielding $W a_k q_l$ in the skin membrane:

$$(m_{ij}, a_j W) \overset{4.ij}{\Longrightarrow} (m_s, W a_k q_l).$$

Next, the only possibility to continue is to apply the group of rules $2.j$ until string $q_l W a_k$ is obtained:

$$(m_s, W a_k q_l) \overset{2.j_1}{\Longrightarrow} \ldots \overset{2.j_t}{\Longrightarrow} (m_s q_l W a_k).$$

Thus we showed that Π correctly simulates rule $q_i a_j \to a_k q_l$ of M.

It is not difficult to see that rules of type $\mathbf{q_i a_j \to q_l}$, $q_i \in Q', q_l \in Q, a_j \in \Sigma$, resp. $\mathbf{q_i a_0 \to a_k q_l a_0}$, $q_i \in Q', q_l \in Q, a_j \in \Sigma$, can be simulated in a similar manner replacing $4.ij$ by $5.ij$, resp. $6.ij$.

We observe that for a string that reached a halting configuration $q_f W$ in M, only rule 3 is applicable on the corresponding string $q_f W$ of Π. This leads to the word W that is sent out of the system.

Hence we obtain that for any transition $w \Longrightarrow w'$ in M there is a unique sequence of transitions $(m_s, w) \Longrightarrow (m_{ij}, w_1) \ldots \Longrightarrow (m_s, w_k) \Longrightarrow (m_s, w')$ in Π, for some $w_j \in O^*$ and $k > 0$. □

Corollary 1 *There exists a universal 1-tag P system with 73 instructions.*

Proof. Consider the universal CPM0 from [2]. It has 6 states and 6 symbols. By applying Theorem 1 to this machine we obtain a universal 1-tag P system with 73 rules. □

4 Conclusion

In this article we considered the tag operation in the context of P systems. The obtained variant is universal even with the deletion number equal to one. Moreover, the obtained system has 73 instructions while best actually known constructions for universal tag systems have around 480 [7]. An open problem is if this number can be decreased.

P systems framework for the tag operation can be considered as a particular variant of the graph-controlled derivation using the tag operation. We observe that the particular structure of the graph from Theorem 1 corresponds to a matrix control with the depth (size of the matrices) equal to two. Hence Corollary 1 also holds for matrix tag systems. It could be interesting to consider other control mechanisms like random-context control with the tag operation.

References

[1] A. Alhazov, A. Krassovitskiy, Yu.Rogozhin. *Circular Post Machines and P Systems with Exo-insertion and Deletion.* Lecture Notes in Computer Science, **7184** (2011), pp. 73–86.

[2] A. Alhazov, M. Kudlek, Yu. Rogozhin. *Nine Universal Circular Post Machines.* Computer Science Journal of Moldova, **10, no.3** (2002), pp. 247–262.

[3] J. Cocke, M. Minsky. *Universality of tag systems with p=2.* Journal of the ACM, **11**, 1, (1964), pp. 15–20.

[4] M. Kudlek, Yu. Rogozhin. *Small Universal Circular Post Machines.* Computer Science Journal of Moldova, **9(1)** (2001), pp. 34–52.

[5] M. Margenstern. *Frontier between decidability and undecidability: A survey,* Theoretical Computer Science, **231**(2) (2000), pp. 217–251.

[6] S. Maslov. *On E. L. Posts Tag problem.*, (In Russian) Trudy Matematicheskogo Instituta imeni V.A. Steklova (1964b), no. 72, pp. 5-56, English translation in: American Mathematical Society Translations Series 2, 97, pp. 1–14, 1971.

[7] L. De Mol. *On the complex behavior of simple tag systems – An experimental approach.* Theoretical Computer Science, **412**(1-2) (2011), pp. 97–112.

[8] M. Minsky. *Recursive unsolvability of Posts problem of tag and other topics in the theory of Turing machines*, Annals of Mathematics, **74** (1961), pp. 437–455.

[9] M. Minsky. *Computations: Finite and Infinite Machines.* Prentice Hall, Englewood Cliffts, NJ (1967).

[10] G. Păun. *Membrane Computing. An Introduction.* Springer, 2002.

[11] G. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.

[12] E. Post. *Formal reductions of the general combinatorial decision problem*, American Journal of Mathematics, **65**(2) (1943), pp. 197–215.

[13] E. Post. *Absolutely unsolvable problems and relatively undecidable propositions – account of an anticipation, The Undecidable.* In Martin Davis, ed., Basic papers on undecidable propositions, unsolvable problems and computable functions, Raven Press, 1965, pp. 340–433.

[14] G. Rozenberg, A. Salomaa. *Handbook of Formal Languages,* 3 volumes. Springer Verlag, Berlin, Heidelberg, New York (1997).

[15] The P systems Web page. `http://ppage.psystems.eu/`

[16] H. Wang. *Tag systems and lag systems,* Mathematische Annalen, **152** (1963a), pp. 65–74.

Yu. Rogozhin[1], S. Verlan[2,1],

Received July 9, 2012

[1] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
5 Academiei str., Chişinău, MD-2028, Moldova

[2] LACL, Departement Informatique
UFR Sciences et Technologie
Universite Paris Est – Créteil Val de Marne
61, av. Géńeral de Gaulle
94010 Creteil, France

E–mails:
Dr.hab. Yurii Rogozhin: *rogozhin@math.md*,
Dr.hab. Sergey Verlan: *verlan@univ-paris12.fr*,

Computer Science Journal of Moldova, vol.**20**, no.3(60), 2012

Static and Dynamic Membrane Structures

Sergiu Ivanov

Abstract

While originally P systems were defined to contain multiset rewriting rules, it turned out that considering different types of rules may produce important results, such as increasing the computational power of the rules. This paper focuses on factoring out the concept of a membrane structure out of various P system models with the goal of providing useful formalisations. Both static and dynamic membrane structures are considered.

Keywords: Computing model, P system, membrane structure, semi-lattice, active membranes.

1 Introduction

P systems are computational models inspired from the structure of living cells, introduced by Gh. Păun in 1998 [1]. The principal idea behind this model is that the chemical reactions happening in a biological cell can be interpreted as applications of rewriting rules to multisets of objects. Since formal grammars can be treated as computational devices, a cell can be basically viewed as a collection of compartments, each hosting computation. Further, communication between compartments is allowed, which binds the computing devices into a network where information is produced and consumed to be eventually combined into the final result. For a more thorough introduction to the subject the reader may turn to [2].

One of P systems types which is commonly brought about in examples is the transitional P systems [3]. In transitional P systems, the compartments of P systems hold multiset rewriting rules. It has been shown (see Chapter 4 of [4] for a summary) that membrane structure

does not add any computational power to what is already provided by the class of multiset rewriting rules in use. The idea behind this result is simple: since a membrane structure is finite and static in this case, it can very well be dropped by considering "labelled" symbols: instead of having a in compartment 1, have the symbol a_1, for example. In this way, one can simulate any communication between the computing compartments which can happen in a transitional P system and which could enhance the overall computational power.

While this conclusion may look rather disconcerting in what concerns the utility of transitional P systems, static membrane structures may actually be rather significant in certain situations. The authors of [5] show that, if one places insertion-deletion rules in the compartments of a membrane structure, one obtains a computational device which is more powerful than the class of insertion-deletion rules in use. In fact, this is not the only well-known example of placing other types of rules in compartments of membrane structures; consider, for example, splicing P systems (Chapter 8 of [4]) and P systems with string objects (Chapter 7 of [4]). Note that in these cases, the rules placed in the compartments of the membrane system do make sense outside of the context of membrane structures. I find it necessary to explicitly contrast this with communication P systems (Chapter 5 of [4]) and P systems with active membranes (Chapter 11 of [4]), in which cases the investigated rules seem to be very intimately connected with the membrane structure itself.

The reasoning exposed in the previous paragraph brings attention to the membrane structure, rather than to the P system that results from combining a membrane structure and rules. Some basic formal representations are widely used in which membrane structures are considered as rooted trees [3, 4]. However, as this paper shows, the underlying tree of a membrane structure is a skeleton which, while being essential, is far from covering all the features associated with the membranes. Further note that, while formalising static membrane structures is an interesting and useful task in itself, it is the dynamic membrane structures arising in different flavours of P systems with active membranes that are the most attractive object of formalisation.

This paper focuses on studying membrane structures as separate objects, apart from the containing context of P systems. An approach to formalising static and dynamic membrane structures as algebraic structures is suggested, and then applications of the obtained formalisation are shown.

2 Preliminaries

2.1 Multisets

Given a finite set A, by $|A|$ one understands the number of elements in A.

Let V be a finite alphabet; then V^* is the set of all finite strings of a V, and $V^+ = V^* - \{\lambda\}$, where λ is the empty string. By $\mathbb{N}$ one denotes the set of all non-negative integers, by $\mathbb{N}^k$ – the set of all vectors of non-negative integers.

Let V be a finite set, $V = \{a_1, \ldots, a_k\}$, $k \in \mathbb{N}$. A *finite multiset* M over V is a mapping $M : V \to \mathbb{N}$. For each $a \in V$, $M(a)$ indicates the number of "occurrences" of a in M. The value $M(a)$ is called the *multiplicity* of a in M. The size of the multiset M is $|M| = \sum_{a \in V} M(a)$, i.e., the total count of the entries of the multiset. A multiset M over V can also be represented by any string x which contains exactly $M(a_i)$ instances of a_i, $1 \leq i \leq k$. The support of M is the set $supp(M) = \{a \in V \mid M(a) \geq 1\}$, which is the set which contains all elements of the multiset. For example, the multiset over $\{a, b, c\}$ defined by the mapping $\{(a, 3), (b, 1), (c, 0)\}$ can be written as a^3b. The support of this multiset is $\{a, b\}$.

Let x, y be two multisets over V. Then x is called a *submultiset* of y, written as $x \subseteq y$, if and only if $\forall a \in V \,.\, x(a) \leq y(a)$. The union of x and y, denoted by $x \uplus y$ is defined in the following way: $\forall a \in V \,.\, (x \uplus y)(a) = x(a) + y(a)$. The difference of x and y, denoted by $x \backslash y$, is defined similarly: $\forall a \in V \,.\, (x \backslash y)(a) = x(a) - y(a)$.

2.2 P Systems

A *transitional* membrane system is defined by a tuple (Chapter 1 of [4])

$$\begin{array}{lcl} \Pi & = & (O, \mu, w_1, w_2, \ldots, w_m, R_1, R_2, \ldots, R_m, i_0), \text{ where} \\ O & & \text{is a finite set of objects,} \\ \mu & & \text{is a hierarchical structure of } m \text{ membranes, bijectively} \\ & & \text{labelled with } 1, \ldots, m, \\ w_i & & \text{is the initial multiset in region } i, 1 \leq i \leq m, \\ R_i & & \text{is the set of rules of region } i, 1 \leq i \leq m, \\ i_0 & & \text{is the output region.} \end{array}$$

The rules have the form $u \to v$, where $u \in O^+$, $v \in (O \times Tar)^*$. The target indications from $Tar = \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$, where j are the labels of the corresponding inner membranes. The target *here* is typically omitted. In case of non-cooperative rules, $u \in O$.

The rules are applied in a maximally parallel way: no further rule should be applicable to the idle objects. In the case of non-cooperative systems, all objects evolve by the associated rules in the corresponding regions (except objects a in regions i such that R_i does not contain any rule $a \to u$, but these objects do not contribute to the result). Rules are non-deterministically chosen at each moment in time when a change occurs in the configuration of the P system. The process of choosing which rules should be applied does not take any time.

A P system with *active membranes* is defined by a tuple (Chapter 11 of [4]):

$$\begin{array}{lcl} \Pi & = & (O, H, E, \mu, w_1, w_2, \ldots, w_m, R, i_0), \text{ where} \\ O & & \text{is a finite set of objects,} \\ H & & \text{is the alphabet of names of membranes,} \\ E & & \text{is the set of electrical charges,} \\ \mu & & \text{is the initial hierarchical structure of } m \text{ membranes,} \\ & & \text{bijectively labelled by } 1, \ldots, m; \end{array}$$

w_i	is the initial multiset in region $i, 1 \leq i \leq m$,
R	is the set of rules,
i_0	is the output region.

The rules in P systems with active membranes can be of the following five basic types:

(a) $[a \to v]^e_h$, $h \in H$, $e \in E$, $a \in O$, $v \in O^*$;

(b) $a[\,]^{e_1}_h \to [b]^{e_2}_h$, $h \in H$, $e_1, e_2 \in E$, $a, b \in O$;

(c) $[a]^{e_1}_h \to [\,]^{e_2}_h b$, $h \in H$, $e_1, e_2 \in E$, $a, b \in O$;

(d) $[a]^e_h \to b$, $h \in H \backslash \{s\}$, $e \in E$, $a, b \in O$;

(e) $[a]^{e_1}_h \to [b]^{e_2}_h [c]^{e_3}_h$, $h \in H \backslash \{s\}$, $e_1, e_2, e_3 \in E$, $a, b, c \in O$.

It is often considered that $E = \{0, -, +\}$. The rules apply to elementary membranes, i.e., membranes which do not contain other membranes inside.

The rules are applied in the usual non-deterministic maximally parallel manner, with the following details: any object can be subject of only one rule of any type and any membrane can be subject of only one rule of types (b)–(e). Rules of type (a) are not counted as applied to membranes, but only to objects. This means that when a rule of type (a) is applied, the membrane can also evolve by means of a rule of another type. If a rule of type (e) is applied to a membrane, and its inner objects evolve at the same step, it is assumed that first the inner objects evolve and then the division takes place, so that the result of applying rules inside the original membrane is replicated in the two new membranes.

2.3 Semilattices

A binary relation $\leq$ is a partial order if it is reflexive, symmetric, and transitive. A set $(S, \leq)$ endowed with such a binary relation is called

a partially ordered set. If $x, y \in S$ such that $(x, y) \notin \leq$, the elements x and y are called incomparable; this is written as $x \not\leq y$. The *interval* between two comparable elements $x, y \in L$, denoted by $[x, y]$ is the set of all elements in L which are "between" x and y:

$$\forall x, y \in S \,.\, x \leq y \,.\, [x, y] \stackrel{def}{=} \{a \in L \mid x \leq a \text{ and } a \leq y\}$$

An interval is called simple if it only includes its "endpoints":

$$\forall x, y \in L \,.\, [x, y] \text{ – simple } \stackrel{def}{\iff} [x, y] = \{x, y\}.$$

In this case x is called the predecessor of y (or y – the successor of x), which is denoted by $x \prec y$.

A partially ordered set $(S, \leq)$ is a *meet-semilattice*, if for any $x, y \in S$ the greatest lower bound $x \wedge y$ (the meet) of the two exists:

$$\forall x, y \in S \,.\, \exists x \wedge y \in S \,.\, \forall z \in S \,.\, (z \leq x \text{ and } z \leq y) \Longrightarrow z \leq x \wedge y.$$

Dually, one defines the join-semilattice. A partially ordered set $(S, \leq)$ is a *join-semilattice*, if for any $x, y \in S$ the least upper bound $x \vee y$ (the join) of the two exists:

$$\forall x, y \in S \,.\, \exists x \vee y \in S \,.\, \forall z \in S \,.\, (x \leq z \text{ and } y \leq z) \Longrightarrow x \vee y \leq z.$$

Any of these two can be defined as an algebraic structure. For example, a meet-semilattice is the structure $(S, \wedge)$ in which the binary operation is idempotent, commutative, and associative:

$$(S, \wedge)\text{– semilattice} \stackrel{def}{\iff} \begin{array}{ll} \forall x, y, z \in S \,. & x \wedge x = x \\ \text{and} & x \wedge y = y \wedge x \\ \text{and} & x \wedge (y \wedge z) = (x \wedge y) \wedge z. \end{array}$$

3 Static Membrane Structures

3.1 Construction of Static Membrane Structures

Consider a finite meet-semilattice $(L, \wedge)$ with the properties that the semilattice includes the minimal element, denoted by 0:

$$\exists 0 \in L \,.\, \forall x \in L \,.\, 0 \leq x, \tag{1}$$

and that any element of L except 0 has only one predecessor:

$$\forall x \in L\backslash\{0\} \,.\, \exists! y \in L \,.\, y \prec x. \tag{2}$$

The following lemma shows that finite semilattices with these two properties are essentially trees.

Lemma 1. *Let $(L, \wedge)$ be a finite meet-semilattice. Consider the graph $G = (V, E)$ with vertexes all elements of L and edges all corresponding simple intervals:*

$$V \;=\; L, \qquad E \;=\; \{(x, y) \in L \times L \mid x \prec y\},$$

If $(L, \wedge)$ has the properties (1) *and* (2)*, then G is a tree.*

Proof. Let $n = |L| = |V|$ be the number of elements in the set $L = V$. Since any element $a \in L\backslash\{0\}$ has exactly one predecessor, the count of edges in G is $|E| = n - 1$. Further, G is connected, because $\forall x \in L = V \,.\, 0 \leq x$, which means that there exists a sequence of elements $(x_i)_{i=1}^{m} \subseteq L$, $m \in \mathbb{N}$, such that

$$0 = x_1 \prec x_2 \prec \ldots \prec x_m = x,$$

which gives the path in G connecting 0 and x. Since G is a connected graph in which $|E| = |V| - 1$, G is a tree [7]. □ □

In particular, if L satisfies the properties (1) and (2), then L contains no meets for any incomparable elements: $\forall x, y \in L \,.\, x \not\leq y \Longrightarrow x \wedge y \notin L$.

For a set S and a set H, a mapping $l : S \to H$ will be called a *labelling* of S with the label set H. Note that l is not required to be injective, which means that several objects in S may have the same label.

Definition 1. *The following tuple will be called a* membrane structure*:*

$$\begin{array}{rcl} M & = & ((L, \wedge), H, l)\,, \textit{ where} \\ (L, \wedge) & & \textit{is a meet-semilattice with the properties } (1) \textit{ and } (2), \\ H & & \textit{is a set of labels}, \\ l & & \textit{is a labelling of } L \textit{ with } H. \end{array}$$

The elements of L will be called membranes*.*

It is easy to see that a membrane structure in this definition is exactly the same thing as what is defined in numerous articles on P systems (for example, Chapter 1 of [4]). The important part is that the meet-semilattice $(L, \wedge)$ was shown to be a tree. The set of labels H and the corresponding labelling l obviously corresponds to the usual labelling of membranes.

Example 1. *Consider the structure*

$$[\,[\,[\;\;]_3\,]_2[\;\;]_4\,]_1,$$

in which the membrane with label 1 *contains a membrane with label* 4 *and a membrane with label* 2*, which, in its turn, contains a membrane with label* 3*, will be translated to the membrane structure* $M = ((L = \{a, b, c, d\}, \wedge), H = \{1, 2, 3, 4\}, l, \varnothing)$*, where* $l(a) = 1$*,* $l(b) = 2$*,* $l(c) = 3$*,* $l(d) = 4$*, and the partial order on* L *is given by the following set of pairs:*

$$\leq = \{(a, b), (a, d), (a, c), (b, c)\}\,.$$

$(L, \wedge)$ *satisfies the properties* (1) *and* (2)*. Indeed,* $\forall x \in L \,.\, a \leq x$*, thus* $a = 0$ *in the terminology introduced in this section. Further, it is easy to check that each element in* L*, except for* a*, has exactly one predecessor. Thus,* M *is a valid membrane structure.*

It should be clear now that, if $x, y \in L$ and $x \prec y$, then x is the parent membrane of y.

Note that while the definition given in this paper generalises the majority of other definitions of tree-like membrane structures, it does not cover much more than what is covered by the said definitions. Thus, the notion of membrane structure as introduced in the present paper is sufficiently narrow.

Remark that there has not been any mentioning of the environment, which is sometimes considered as a compartment with some limitations (Chapter 1 of [4]). It is easy, however, to extend the semi-lattice $(L, \wedge)$ by adding an element $0'$ with the property that $\forall x \in L \,.\, 0' \leq x$ to represent the environment.

Also note that a join-semilattice could have been chosen instead of a meet-semilattice. Obviously, any reasoning about membrane structures

considered as meet-semilattices can be converted to join-semilattices by substituting the word "meet" for "join", $\wedge$ for $\vee$ and reversing the direction of comparisons.

Finally, I would like to discuss the usefulness of the new formalisation. While it has been shown that the principal component of a membrane structure, the semilattice $(L, \wedge)$, is always a tree, the advantage of this approach is that a membrane structure is defined as an algebraic structure, which makes it easier to define morphisms, as will be shown in the concluding sections of this paper.

3.2 Construction of P Systems with Static Membrane Structures

Consider a finite set O and a set of rules R over this alphabet. No other restrictions on the two sets are imposed, i.e., any type of rules over O is allowed. Define the application $\sigma : R \times O \to O \cup \{\perp\}$, $\perp \notin O$, in the following way: if a rule $r \in R$ is applicable to an object $o \in O$, then $\sigma(r, o)$ is the result of application of r to o. If r is not applicable to o, $\sigma(r, o)$ is defined to be $\perp$. The terms "applicable" and "application" are expected to be defined during the construction of the sets O and R. For the purposes of this article, the inner structure of the rules and objects is inessential, as long as some basic statements can be asserted about either of them.

Consider a membrane structure $M = ((L, \wedge), H, l, A, a)$ and two labellings of L: $object : L \to O$ and $rules : L \to 2^R$, where 2^X is the set of all subsets (the power set) of X. Setting up such labellings can be intuitively perceived as creating a system of nested compartments, with an object and rules in each compartment. Note that since no restriction has been imposed on O, an object may be anything, including a set, a multiset, a string, a set of strings, etc.

Further, introduce the function $outer : L \to L \cup \{\perp\}$, which yields the containing membrane for the given membrane, or $\perp$ if the argument is 0:

$$\begin{aligned} m \in L \backslash \{0\} \Longrightarrow \quad & outer(m) \stackrel{def}{=} p, p \prec m; \\ & outer(0) \stackrel{def}{=} \perp . \end{aligned}$$

Similarly, the function $inner : L \to 2^L$ yields the immediately inner membranes of the given membrane:

$$inner(m) \stackrel{def}{=} \{c \in L \mid m \prec c\}.$$

To simplify further expressions, the convenience function $adjacent : L \to 2^L$ will be introduced:

$$\begin{array}{rcl} m \in L\backslash\{0\} \Longrightarrow adjacent(m) & \stackrel{def}{=} & inner(m) \cup outer(m); \\ adjacent(0) & \stackrel{def}{=} & inner(0). \end{array}$$

Now define two applications $iLabels, oLabels : L \times R \to 2^H$ in the following way: if $m \in L$ and $r \in rules(m)$, then $iLabels(m, r)$ is the set of input labels for the rule r in membrane m, and $oLabels(m, r)$ is the set of output labels for r. If $r \notin rules(m)$, then $iLabels(m, r) \stackrel{def}{=} oLabels(m, r) \stackrel{def}{=} \varnothing$. These functions annotate a rule with the information about the labels of the membranes whose contents it may use or modify. To ensure the validity of the labels in the context of the membrane structure, one defines the function $validLabels : L \times 2^H \to 2^H$ in the following way:

$$validLabels(m, H') \stackrel{def}{=} H' \cap \{l(b) \mid b \in adjacent(m)\}.$$

Thus, *validLabels* insures that a set of labels only contains the labels of the outer and inner membranes of m, enforcing the well-known pattern of communication along the tree in P systems.

Finally, define the applications

$$\begin{array}{rcccll} buildInput & : & L \times R & \to & O & \cup\{\perp\}, \\ outputBuilder & : & L \times R & \to & \hom_{\mathbf{Set}}(O \times L, \quad O) & \cup\{\perp\}. \end{array}$$

where $\hom_{\mathbf{Set}}(A, B)$ is the set of applications between the sets A and B.

To understand the meaning of the last two applications, consider again a membrane $m \in L$, and a rule r in the associated set of rules

$rules(m)$. $buildInput(m, r)$ constructs the objects belonging to the compartments the rule r depends on:

$$\{object(m) \mid m \in adjacent(r) \text{ and } l(m) \in iLabels(m, r)\},$$

then "combines" these objects and $object(m)$. The meaning of the verb "combine" should be defined in the description of the rules R and how they act on the objects in O.

The value $outputBuilder(m, r)$ is a function $f : O \times O \to O$ which, for an object o and a membrane $b \in adjacent(m)$, returns the "combination" of the object o with $object(b)$, or produces other modifications to $object(b)$. Again, the term "combination" should be defined in the description of the rules R and of how they act on the objects in O.

In the case when r does not belong to the set of rules associated with m, the last two applications take the value $\perp$:

$$\begin{array}{lll} r \notin rules(m) \implies & buildInput(m, r) & \stackrel{def}{=} \perp \\ \text{and} & outputBuilder(m, r) & \stackrel{def}{=} \perp \end{array}$$

If some input conditions are not satisfied in $buildInput$, this function should take the value $\perp$.

Definition 2. *The following construction will be referred to as a* P system with static (tree-like) membrane structure*:*

$$\begin{array}{rcl} \Pi & = & (M, O, R, \sigma, object, rules, iLabels, \\ & & oLabels, buildInput, outputBuilder, i_0), \end{array}$$

where $i_0 \in H$ is the label of the output membrane (s).

Similarly to the usual definition, a configuration $C : L \to O$ of Π is the collection of the contents of the compartments, indexed by membranes: $C(m) = object(m)$.

Before proceeding to extending the formalisation to the semantics of the P systems, an example would be helpful in showing how the static structure of familiar constructs of P systems maps to the definition given in the current paper.

Example 2. *Consider a transitional P system*

$$\Pi' = (O', \mu, w_1, w_2, \ldots, w_n, R_1, R_2, \ldots, R_n, i_0).$$

In the previous sections it has already been shown how μ maps to the semilattice $(L, \wedge)$. The set of labels H is the set of numbers 1 through n: $H = \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$ and the (bijective) labelling l is defined in the obvious way.

The set of objects O is the set of multisets over $O'' = O' \cup \{(o, t) \mid o \in O' \text{ and } t \in Tar\}$. The set of rules R contains all multiset rewriting rules over the alphabet O'', whose left-hand sides do not include target indications:

$$R = \{u \to v \mid u \in O'^* \text{ and } v \in O''^*\},$$

where X^* was used to denote the set of multisets over X. The application σ carries out the usual application of a multiset rewriting rule to a multiset. The labelling object associates to the membrane labelled with i, $1 \leq i \leq m$, the multiset w_i. Similarly, the labelling rules associates to the membrane with label i, $1 \leq i \leq n$, the set of rules R_i.

The application $iLabels$ takes the value $\varnothing$ for any valid combination of arguments. For $m \in L$ and $r \in rules(m)$, the function $oLabels(m, r)$ is the set of labels mentioned in target indications of the right-hand side of the rule r, excluding the label of m. The application $buildInput$ is trivially defined as $buildInput(m, r) = object(m)$.

The value $f : O \times L \to O$ of $outputBuilder(m, r)$ is defined in the following way. For every $b \in \{b \in L \mid b \in adjacent(m) \text{ and } l(b) \in oLabels(m, r)\}$, and an object $o \in O$, $f(o, b)$ will result in multiset union of $object(b)$ and the multiset of all objects of o with target indications $l(b)$. The value $f(o, m)$ will result in constructing a multiset o' by subtracting the left-hand side of r from $object(m)$ and then performing multiset union of o' and the multiset of objects of o which have no target indications or have the indication here. For all other membranes x, the value of the function is trivially defined as $f(o, x) = object(x)$. Thus, $buildObject$ distributes the symbols across the corresponding membranes.

3.3 Computation in P Systems with Static Membrane Structure

With the necessary tools set up, it is now possible to completely describe how a P system with static membrane structure, as defined in this paper, transitions from one configuration into another configuration. This will eventually make it possible to define computation.

The reasoning exposed in this section is loosely based on the considerations in [8], which provides a different approach to generalising P systems with static membrane structures, whereby the tree-like membrane structure is almost wholly dismissed.

Consider a P system Π, as defined in the previous section. Remark that different configurations of Π are given by different mappings $C = object$. To avoid confusion, as well as to specify the origin of the corresponding functions, subscripts will be henceforth supplied which show which P system and which configuration thereof is being considered.

Define the function $applyRule_{\Pi,C} : L \times R \to \hom_{\mathbf{Set}}(L, O) \cup \{\perp\}$. Its purpose is to produce a new configuration by applying a rule associated to a membrane. For $m \in L$, $r \in rules_{\Pi}(m)$, under the conditions that $buildInput_{\Pi,C}(m, r) \neq \perp$ and $\sigma(r, buildInput_{\Pi}, C(m, r)) \neq \perp$, $applyRule_{\Pi,C}$ is defined as follows:

$$\begin{aligned} applyRule_{\Pi,C}(m, r)(b) &\stackrel{def}{=} doOutput(result, b), \text{ where} \\ result &\stackrel{def}{=} \sigma(r, buildInput_{\Pi,C}(m, r)), \\ doOutput &\stackrel{def}{=} outputBuilder_{\Pi,C}(m, r). \end{aligned}$$

If the enumerated conditions are not satisfied, $applyRule_{\Pi,C}(m, r) = \perp$.

According to this definition, $applyRule_{\Pi,C}(m, r)$ is a function which maps every membrane to the objects contained within, after the application of the rule $r \in rules_{\Pi}(m)$. If applying the rule is not possible, $applyRule_{\Pi,C}(m, r)$ takes the special signal value $\perp$.

Note that, while the description of the process of applying a rule by σ is done rather generally and informally, quite a bit of effort is invested into specifying the modifications induced by the associated membrane structure in as detailed a way as possible.

Definition 3. *A rule $r \in rules_\Pi(m)$, for an $m \in L$, is said to be applicable in the configuration C if $applyRule_{\Pi,C}(m,r) \neq \perp$.*

In a given configuration given by the mapping $C = object$, the set of applicable rules is defined as

$$\begin{aligned} applicableRules(\Pi, C) &\stackrel{def}{=} \{r \in rules_\Pi(m) \mid m \in L \\ &\qquad \text{and } applyRule_{\Pi,C}(m,r) \neq \perp\}. \end{aligned}$$

It is would now be desirable to construct the analog of the marking algorithm introduced in [8]. To do this, it should be remarked that an application of a rule $r \in R$ is made possible because certain "premises" are satisfied. The action of applying r may entail removal of some of these premises. To account for this, define the application

$$premisesEraser_{\Pi,C} : L \times R \to \hom_{\mathbf{Set}}(O \times L, O) \cup \{\perp\},$$

which, in parallel to $outputBuilder_{\Pi,C}$, produces a function which removes, if possible, the premises which made the rule $r \in rules_\Pi(m)$, $m \in L$, applicable. These considerations lead to the definition of the application $erasePremises_{\Pi,C} : L \times R \to \hom_{\mathbf{Set}}(L, O) \cup \{\perp\}$, in parallel to $applyRule_{\Pi,C}$:

$$\begin{aligned} erasePremises_{\Pi,C}(m,r)(b) &\stackrel{def}{=} doErase(result, b), \text{ where} \\ result &\stackrel{def}{=} \sigma(r, buildInput_{\Pi,C}(m,r)), \\ doErase &\stackrel{def}{=} premisesEraser_{\Pi,C}(m,r). \end{aligned}$$

This definition is valid when $r \in rules_\Pi(m)$, $m \in L$. If this does not hold, or if $buildInput_{\Pi,C}(m,r) = \perp$, or if $\sigma(r, buildInput_{\Pi,C}(m,r)) = \perp$, then $erasePremises_{\Pi,C} \stackrel{def}{=} \perp$.

The are now sufficient instruments to construct the marking algorithm. Consider a multiset ρ of pairs rules and the corresponding membranes:

$$\rho = \{((m,r), n) \mid m \in L \text{ and } r \in rules_\Pi(m) \text{ and } n \in \mathbb{N}\}.$$

Define the function

$$isApplicableMultiset_{\Pi} : \hom_{\mathbf{Set}}(L, O) \times (L \times R)^* \to \{\texttt{true}, \texttt{false}\}$$

to be `true` if all rules in ρ can be applied the corresponding number of times in the supplied configuration and to be `false` otherwise:

$$\begin{aligned} isApplicableMultiset_{\Pi}(C, \lambda) &\stackrel{def}{=} \texttt{true} \\ (m, r) \in \rho \Longrightarrow & \\ isApplicableMultiset_{\Pi}(C, \rho) &\stackrel{def}{=} r \in applicableRules(\Pi, C) \\ \text{and} \quad & isApplicableMultiset_{\Pi}(C', \rho'), \\ \text{where } C' &\stackrel{def}{=} erasePremises_{\Pi,C}(m, r), \\ \rho' &\stackrel{def}{=} \rho \backslash \{(m, r)\}. \end{aligned}$$

Here λ was used to denote the empty multiset.

The function $isApplicableMultiset_{\Pi}$ essentially performs the same procedure as does the marking algorithm in [8]. It checks the applicability of every rule in the multiset ρ and removes the rules found applicable one by one. If the multiset becomes empty, the conclusion is drawn that all rules in ρ can be applied the corresponding number of times in the current configuration. Otherwise, the function is `false`.

Once the multiset of membranes and rules ρ has been decided to be applicable, the rules in ρ may obviously be applied one by one, by invoking $applyRule_{\Pi,C}$ for all of them. Thus, the basic semantics has been constructed. Further definitions provided in [8] like, for example, derivation modes, halting conditions, etc., can be easily adapted to the algorithms described in this section, which eventually completes the formalisation of P systems with static (tree-like) membrane structure.

4 Dynamic Membrane Structures

4.1 Construction of P systems with Dynamic Membrane Structure

In this section the definition of a membrane structure will be extended to cover the dynamic membrane structures arising in P systems with active membranes, for example.

Definition 4. *The following tuple will be called a (dynamic)* membrane structure*:*

$$\begin{array}{rcl} M & = & ((L,\wedge),H,l,A,a)\text{, where} \\ (L,\wedge) & & \text{is a meet-semilattice with the properties (1) and (2),} \\ H & & \text{is a set of labels,} \\ l & & \text{is a labelling of } L \text{ with } H, \\ A & & \text{is a set of attributes,} \\ a & & \text{is a labelling of } L \text{ with } A. \end{array}$$

If $A = \varnothing$, by convention, the last two components of the tuple will not be written. Thus, the definition introduced in Subsection 3.1 can be regarded as a special case of this definition.

The need for the set of attributes arises from the fact that, in P systems with active membranes, the membranes sometimes carry charge (Chapter 11 of [4]). To model this feature, one can define $A = \{0, -, +\}$; then, for a membrane $m \in L$, $a(m) \in A$ will give the charge.

In the previous parts of the paper it has been shown how the membrane structure M, together with the sets yielded by $iLabels$ and $oLabels$, directs the way rule applications happen. However, as it can be seen in Subsection 2.2, rules that influence the membrane structure itself are in very tight connection with the membranes, which makes it quite difficult to construct the parallel to the mappings $iLabels$ and $oLabels$ which would indicate how a rule acts on the membrane structure. A possible solution is to even further decouple the action of a rule on a membrane structure for the nature of the rule itself. More concretely, a rule in a P system with dynamic membrane structure will be written as two rules: a rule which works as described in the Subsection 3.3, and another rule, acting on the membrane structure. The coming paragraphs will provide further details, as well as a formal explanation.

In order to better describe the semantics of dynamic membrane structure, the reasoning will start in the frame of a P system with the (yet static) membrane structure M, as defined in Subsection 3.2. Thus,

consider the P system

$$\begin{aligned}\Pi \;=\; & (M, O, R, \sigma, object, rules, iLabels,\\ & oLabels, buildInput, outputBuilder).\end{aligned}$$

To benefit from the attributes in M, for a membrane $m \in L$ and an associated rule $r \in rules(m)$, define the application $contextChecker : L \times R \to \hom_{\mathbf{Set}}(A, \{\texttt{true}, \texttt{false}\})$ in the following way: $contextChecker(m, r)$ is a function, which checks the attributes of the membrane r, and decides whether the context is "suitable" or not. The meaning of this function will become clearer in the next section.

Fix a membrane $m \in L$ and a rule $r \in rules(m)$ associated with it. Consider the set of pairs of labels and attributes, valid in the context of the membrane m:

$$labsAttrs(m) \stackrel{def}{=} \left\{ \big(l(m'), attr\big) \mid m' \in adjacent(m) \text{ and } attr \in A \right\}.$$

Definition 5. *A* membrane structure rule *in the context of a membrane m is a multiset rewriting rule of the form $u \to v$, where $u, v \in labAttrs(m)^*$, where X^* was used to denote the set of all multisets over X.*

The set of membrane structure rules valid in the context of a given membrane m is given by the application $validMSRules : L \to labAttrs(m)^*$ naturally defined as

$$validMSRules(m) \stackrel{def}{=} \{u \to v \mid u, v \in labAttrs(m)^*\}.$$

The set of all valid membrane structure rules is defined in the following way:

$$allMSRules \stackrel{def}{=} \bigcup_{m \in L} validMSRules(m).$$

What a membrane structure rule is should become clear from an informal example.

Example 3. *Consider the construction* $[\ [\ \ ]_2^+[\ \ ]_3^-\ ]_1^+$. *Then*

$$(2,+)(3,-) \to (2,+)(3,-)(2,-)$$

is a valid membrane structure rule for the membrane with label 1. *From the notation, it should be intuitively understood that this rule produces a new membrane with label* 2 *and charge "−" if the membrane* 1 *contains a membrane* 2 *with charge "+" and a membrane* 3 *with charge "−". What exactly is the action of such a rule, in particular, how it acts upon the inner membranes of the involved membrane and the corresponding rules and objects, will be defined in the next section.*

A conclusion to this subsection is the definition of a P system with dynamic (tree-like) membrane structure.

Definition 6. *The following construct will be referred to as* P system with dynamic (tree-like) membrane structure*:*

$$\begin{aligned}\Pi \ = \ & (M, O, R, \sigma, object, rules, iLabels, oLabels, buildInput,\\ & outputBuilder, contextChecker, msRule, \lambda_O, i_0),\end{aligned}$$

where $\lambda_O \in O$ *is a "default" object to be attached to newly created membranes,* $i_0 \in H$ *is the label of the output membrane, and* $msRule : L \times R \to allMSRules \cup \{\perp\}$ *is defined to be the membrane structure rule associated with the rule* r *associated in its turn with a membrane* m.

If $r \notin rules(m)$, then $msRule(m, r) \stackrel{def}{=} \perp$. If the rule r does not influence the membrane structure, $msRule(m, r) \stackrel{def}{=} \perp$.

4.2 Computation in P Systems with Dynamic Membrane Structure

It is now possible to describe the computation in P systems with dynamic membrane structures.

Among the first things, the exact semantics of membrane structure rules should be described. Consider a P system Π with dynamic membrane structure as defined in the previous section, a membrane $m \in L$,

a rule $r \in rules_{\Pi}(m)$, and the corresponding membrane structure rule $g = msRule_{\Pi}(m, r)$ (assume that it exists, for the purposes of this explanation). Define the utility functions $lhs_{\Pi}, rhs_{\Pi} : allMSRules_{\Pi} \to (H \times A)^*$ as $lhs_{\Pi}(u \to v) = u$ and $rhs_{\Pi}(u \to v) = v$.

Before making this visible from the formal description of semantics, it will be helpful to state that a configuration of P system with dynamic membrane structure includes the mappings between membranes and objects, labels, attributes, as well as the relations between the membranes

$$C = (object, (L, \wedge), l, a).$$

Next define the function $labAttrsMultiset_{\Pi,C} : L \to (H \times A)^*$ to return the pairs of labels and attributes the number of times they occur in inner membranes of a given membrane a:

$$\begin{aligned} labAttrsMultiset_{\Pi,C}(m) &\stackrel{def}{=} buildMultiset(inner_{\Pi}(m)), \\ \text{where } buildMultiset(\varnothing) &\stackrel{def}{=} \lambda, \\ b \in adj \Longrightarrow buildMultiset(adj) &\stackrel{def}{=} (l(m), a(m)) \\ &\quad \uplus buildMultiset(adj'), \\ adj' &\stackrel{def}{=} adj \backslash \{b\}. \end{aligned}$$

Here $\uplus$ was used to denote multiset union. Note the similarity between this function and the notation $labsAttrs$, introduced in the previous section.

Proceed now with defining the function $msRuleApplicable_{\Pi,C} : L \times allMSRules \to \{\texttt{true}, \texttt{false}\}$ to decide whether a membrane structure rule g is applicable to the membrane m or not:

$$msRuleApplicable_{\Pi,C}(m, g) \stackrel{def}{=} lhs_{\Pi}(g) \subseteq labAttrsMultiset_{\Pi,C}(m),$$

where $\subseteq$ was used to denote multiset inclusion.

Now that applicability of a membrane structure rule can be decided, it is time to describe how such a rule is applied. Define the function $labelMembranesMap_{\Pi,C} : L \times H \to \hom_{\mathbf{Set}}(H, 2^L)$ to produce a mapping between some labels in H and the corresponding inner

membranes of a membrane:

$$labelMembranesMap_{\Pi,C}(m, H')(h) \stackrel{def}{=} l^{-1}(h) \cap inner(m),$$

where $l^{-1} : H \to 2^L$ provides the set of membranes labelled with a given label: $l^{-1}(h) \stackrel{def}{=} \{m \in L \mid l(m) = h\}$.

Again, consider a membrane $m \in L$, one of its rules $r \in rules_\Pi(m)$, and the corresponding membrane structure rule $g \in msRule_\Pi(m, r)$. Define the function $involvedMembranes_{\Pi,C} : L \times allMSRules \to 2^L$ to produce the set of membranes involved by the labels in left-hand side of the membrane structure rule:

$$\begin{aligned} involvedMembranes(m, g) &\stackrel{def}{=} \bigcup_{(h,attr)\in I} map(h), \text{ where} \\ map &\stackrel{def}{=} labelMembranesMap(m, labels), \\ labels &\stackrel{def}{=} \{h \in H \mid \exists attr \in A \,.\, (h, attr) \in I\}, \\ I &\stackrel{def}{=} lhs_\Pi(g). \end{aligned}$$

If g is not a membrane structure rule associated with one of the rules of the membrane m, the function is defined to take the value $\perp$:

$$\begin{aligned} (\nexists r \in rules_\Pi(m) \,.\, g = msRule(r)) \Longrightarrow \\ involvedMembranes_{\Pi,C}(m, g) \stackrel{def}{=} \perp \,. \end{aligned}$$

Suppose $msRuleApplicable_{\Pi,C}(m, g) = \texttt{true}$. In this case, define the function $applyMSRule_{\Pi,C}(m, g)$ in the following way:

$$applyMSRule_{\Pi,C}(m, g) \stackrel{def}{=} (object', (L', \wedge'), l', a') = C'.$$

If $msRuleApplicable_{\Pi,C}(m, g) = \texttt{false}$, or $\nexists r \in rules_\Pi(m) \,.\, g = msRule(r)$, $applyMSRule_{\Pi,C}(m, g) \stackrel{def}{=} \perp$.

The underlying set L' of the new semilattice $(L', wedge)$ is obtained by removing first all the membranes involved in the left-hand side of the rule g, and all their inner membranes:

$$\begin{aligned} L'_1 &\stackrel{def}{=} L \setminus (\{b \in L \mid \exists b' \in I \,.\, b' \leq b\}), \\ \wedge'_1 &\stackrel{def}{=} \wedge \setminus \{(b', b'') \mid b' \in I \text{ or } b'' \in I\}, \quad \text{where} \\ I &\stackrel{def}{=} involvedMembranes_{\Pi,C}(m, g). \end{aligned}$$

The symbol $removeMSRuleLHS$ will be used to refer to this operation, i.e.,

$$(L'_1, \wedge'_1) = removeMSRuleLHS((L, \wedge)).$$

Now, define the function $reAddMembranes$ in the following way:

$$reAddMembranes(\lambda, i, (P, \wedge)) \stackrel{def}{=} (P, \wedge),$$

and the value $i \in \mathbb{N}$ is not used in this case. If the first argument α of the function is not an empty multiset and $(h, a) \in \alpha$, then

$$\begin{array}{l} reAddMembranes(\alpha, i, (P, \wedge)) \stackrel{def}{=} \\ reAddMembranes(\alpha', i+1, (P', \wedge')), \text{where} \end{array}$$

$$\begin{array}{rcl} \alpha' & \stackrel{def}{=} & \alpha \backslash ((h, a), 1), \\ P' & \stackrel{def}{=} & P \cup S', \\ \wedge' & \stackrel{def}{=} & \wedge \cup \{(m', b) \in L \times S \mid m' \leq m\}, \\ S' & \stackrel{def}{=} & \{b'_i \in L \mid \exists b \in S \,.\, b \leq b'\}, \\ S & \stackrel{def}{=} & l^{-1}(h) \cup inner(m). \end{array}$$

Thus, according to the definition,

$$(L'_2, \wedge'_2) = reAddMembranes(rhs_\Pi(g), 0, L'_1)$$

reintroduces to the membrane structure all those membranes which have been removed during the construction of L'_1 and which are mentioned in the right-hand side of g, together with all their inner membranes. However, in the process, unique labels are attached to each of the new membranes, which makes it possible to actually duplicate a membrane together with all its inner membranes.

To keep the new labellings synchronised, along with all other identities in $reAddMembranes$, consider the following included among the

definitions in this function:

$$\begin{aligned} b_i \in S \Longrightarrow l'(b_i) &\stackrel{def}{=} h \\ \text{and } l'(b_i) &\stackrel{def}{=} a, \\ b_i \in S' \backslash S \Longrightarrow l'(b_i) &\stackrel{def}{=} l(b_i) \\ \text{and } a'(b_i) &\stackrel{def}{=} a(b_i), \\ object'(b_i) &\stackrel{def}{=} object(b). \end{aligned}$$

Thus, $reAddMembranes$ also updates the labellings for the immediately inner membranes of m which are involved with the membrane structure rule, but leaves the labellings intact for the membranes further down the tree.

Lastly, define the function $addMembranes$ in the following way:

$$\begin{aligned} addMembranes(\lambda, (P, \wedge)) &\stackrel{def}{=} (P, \wedge) \\ (h, a) \in \alpha \Longrightarrow & \\ addMembranes(\alpha, (P, \wedge)) &\stackrel{def}{=} addMembranes(\alpha', (P', \wedge')), \\ \text{where } \alpha' &\stackrel{def}{=} \alpha \backslash \{((h, l), 1)\}, \\ P' &\stackrel{def}{=} P \cup \{m_h\}, \\ \wedge' &\stackrel{def}{=} \wedge \cup \{(m', m_h) \mid m' \in L \\ & \qquad \text{and } m' \leq m\}, \\ m_h &\notin P. \end{aligned}$$

This function adds a new symbol m_h for each new label h in the right-hand side of the membrane structure rule g. Consequently,

$$(L', \wedge') = addMembranes(newMems, (L'_2, \wedge'_2))$$

is the new semilattice, representing the underlying tree of the dynamic membrane structure.

Again, to update the labellings of the membrane structure, the following definitions should be added to the definition of $addMembranes$:

$$\begin{aligned} l'(m_h) &\stackrel{def}{=} h, \\ a'(m_h) &\stackrel{def}{=} a, \\ object'(m_h) &\stackrel{def}{=} \lambda_O. \end{aligned}$$

Thus, the newly added membranes are labelled with default objects, specified in the definition of the P system.

Finally, to complete the definitions of the new labellings of the membrane structures, the following is stated:

$$m \in L_0 \Longrightarrow l'(m) \stackrel{def}{=} l(m) \text{ and } a'(m) \stackrel{def}{=} a(m) \text{ and } object'(m) \stackrel{def}{=} object(m).$$

The conclusion at this point is that $applyMSRule_{\Pi,C}$ formally describes the semantics of a membrane structure rule by constructing a new configuration C' in the context of a P system with dynamic membrane structure Π and a reference configuration C of it.

No types have been provided for $applyMSRule_{\Pi,C}$ and utilities used to construct it, because it returns functions whose domains belong to proper classes (for example, the function $object'$ whose domain is a lattice) and the notations introduced in this paper are insufficient to express this fact. This is irrelevant to the present formalisation, though.

It is now possible to move to defining an evolution step of a P system with dynamic membrane structure. As in Subsection 3.3, only the marking algorithm and one step of evolution will be described in detail. This will create sufficient foundation for continuing the reasoning along the lines shown in [8] and presents little interest in the context of this paper.

Before describing the marking algorithm itself, note that the set of rules employed in P system with dynamic membrane structure is partitioned into two sets: the rules that do not have membrane structure rules associated, and the rules that have:

$$\begin{aligned} R_{\neg\mu} &\stackrel{def}{=} \{r \in R \mid \exists m \in L \,.\, r \in rules_\Pi(m) \text{ and } msRule_\Pi(r) = \perp\}, \\ R_{\mu} &\stackrel{def}{=} \{r \in R \mid \exists m \in L \,.\, r \in rules_\Pi(m) \text{ and } msRule_\Pi(r) \neq \perp\}. \end{aligned}$$

These two types of rules will always be treated in certain order: the rules in $R_{\neg\mu}$ will always be analysed first.

Consider a multiset ρ of pairs rules and the corresponding membranes:

$$\rho = \{((m, r), n) \mid m \in L \text{ and } r \in rules_\Pi(m) \text{ and } n \in \mathbb{N}\}.$$

According to the classification of rules above, split ρ into two multisets:

$$\begin{aligned} \rho_{\neg\mu} &\stackrel{def}{=} \{((m,r),n) \in \rho \mid r \in R_{\neg\mu}\}, \\ \rho_{\mu} &\stackrel{def}{=} \{((m,r),n) \in \rho \mid r \in R_{\mu}\}. \end{aligned}$$

While it is tempting to declare that the function $isApplicableMultiset_{\Pi,C}$ can be used to decide the applicability of $\rho_{\neg\mu}$, it is not exactly so since, in P systems with dynamic membrane structures, the attributes of the membrane a rule is associated with must also be checked. Therefore, define the following simple function $ruleApplicable_{\Pi,C} : L \times R \to \{\texttt{true}, false\}$:

$$\begin{aligned} ruleApplicable_{\Pi,C}(m,r) \stackrel{def}{=} & \\ & applyRule_{\Pi,C}(m,r) \neq \perp \\ \text{and} \quad & contextChecker(m,r)(a(m)) = \texttt{true}. \end{aligned}$$

As usual, for $r \notin rules(m)$, $ruleApplicable_{\Pi,C}(m,r) = \perp$. Now, if one redefines $isApplicableMultiset_{\Pi}$ to use $ruleApplicable_{\Pi,C}$ instead of checking the condition $r \in applicableRules(\Pi, C)$, one may use $isApplicableMultiset_{\Pi}$ to check the applicability of $\rho_{\neg\mu}$ in a P system with dynamic membrane structure.

The current question is how to decide the applicability of ρ_{μ}. The answer to this question is constructed pretty much along the same line as is $isApplicableMultiset_{\Pi,C}$. Define the function $ruleApplicableG_{\Pi,C} : L \times R \to \{\texttt{true}, \texttt{false}\}$ to return `true` if, for a membrane $m \in L$ and its rule $r \in rules_{\Pi}(m)$, both r and $msRule(r)$ are applicable (here G stands for "generalised"):

$$\begin{aligned} ruleApplicableG_{\Pi,C}(m,r) \stackrel{def}{=} & \; msRuleApplicable(m, msRule(r)) \\ \text{and} \quad & ruleApplicable_{\Pi,C}(m,r). \end{aligned}$$

As usually defined in the situations when r is not a rule associated with the membrane m:

$$r \notin rules(m) \Longrightarrow ruleApplicableG_{\Pi,C}(m,r) \stackrel{def}{=} \perp .$$

Further, if $msRule(r) = \perp$, for consistency,

$$ruleApplicableG_{\Pi,C} \stackrel{def}{=} ruleApplicable_{\Pi,C}(m, r).$$

Now define the function $eraseMSRulesG$ which, quite in parallel to $eraseMSRules$ and $applyMSRule$, produces a configuration without the premises which made the rule r and the corresponding $msRule(r)$ applicable:

$$\begin{array}{rcl} eraseMSRulesG_{\Pi,C}(m, r) & \stackrel{def}{=} & (objects', (L', \wedge'), l, a), \text{ where} \\ objects' & \stackrel{def}{=} & eraseMSRules_{\Pi,C}(m, r), \end{array}$$

and $(L', \wedge')$ is defined as follows:

$$(L', \wedge') \stackrel{def}{=} removeMSRulesLHS_{\Pi,C}(m, msRule(r)).$$

Again, $r \notin rules(m) \Longrightarrow eraseMSRulesG_{\Pi,C}(m, r) \stackrel{def}{=} \perp$.

Now, define $isApplicableMSMultiset_{\Pi}$ to decide whether ρ_μ is applicable in the supplied configuration. For an empty multiset, the definition is trivial:

$$isApplicableMSMultiset_{\Pi}(C, \lambda) \stackrel{def}{=} \texttt{true}.$$

For a nonempty multiset ρ_μ and $(m, r) \in \rho_\mu$:

$$\begin{array}{l} isApplicableMSMultiset_{\Pi}(C, \rho_\mu) \stackrel{def}{=} \\ ruleApplicableG_{\Pi,C}(m, r) \text{ and } isApplicableMSMultiset_{\Pi}(C', \rho'_\mu), \\ \text{where } C' \stackrel{def}{=} eraseMSRulesG_{\Pi,C}(m, r), \rho'_\mu \stackrel{def}{=} \rho_\mu \backslash \{(m, r)\}. \end{array}$$

Finally, the function $isApplicableMultisetG_{\Pi,C}$ decides whether the multiset of rules ρ is applicable in the given configuration:

$$\begin{array}{l} isApplicableMultisetG_{\Pi,C}(\rho) \stackrel{def}{=} \\ isApplicableMultiset_{\Pi}(C, \rho_{\neg\mu}) \\ \quad \text{and } isApplicableMSMultiset_{\Pi}(C', \rho_\mu), \end{array}$$

where $C' = (object', (L, \wedge), l, a)$ and $object'$ is the labelling of the membrane structure with objects at which $isApplicableMultiset_{\Pi}(C, \lambda)$ has arrived.

Now, the application of an applicable multiset of rules ρ to configuration of P system with dynamic membrane structure is performed in two stages. First, the multiset of rules $\rho_{\neg\mu}$ is applied as described in Subsection 3.3. Then, the rules in ρ_{μ} are applied one by one, using the function $applyRuleG_{\Pi,C}(m, r)$, defined in the following way. For $r \in rules(m)$ and $g = msRule(r) \neq \perp$,

$$\begin{aligned} applyRuleG_{\Pi,C}(m,r) &\stackrel{def}{=} applyRule_{\Pi,C'}(m,r), \text{ where} \\ C' &\stackrel{def}{=} applyMSRule_{\Pi,C}(m,g). \end{aligned}$$

When $msRule(r) = \perp$, $applyRuleG_{\Pi,C} = applyRule_{\Pi,C}$. For $r \notin rules(m)$, $applyRuleG_{\Pi,C} \stackrel{def}{=} \perp$.

4.3 P Systems With Active Membranes

This section will show how the five types of rules in P systems with active membranes are translated into the suggested formalism.

The rules of type (a), $[a \to v]^e_h$, will be translated to rules in $R_{\neg\mu}$, whose context checkers will assure check the charge of the containing membrane.

The rules of type (b), $a[\,]^{e_1}_h \to [b]^{e_2}_h$, will be modelled in the following way. All parent membranes of h will have a rule which will take an instance of a and will place it into the membrane h: $a \to (a, h)$. The corresponding membrane structure rule will be $(h, e_1) \to (h, e_2)$.

Similarly, for the rules of type (c), $[a]^{e_1}_h \to [\,]^{e_2}_h b$, the parent membrane of h will contain a rule $(a, h) \to b$, with the corresponding membrane structure rule being again $(h, e_1) \to (h, e_2)$.

For the dissolution rules of type (d), $[a]^e_h \to b$, the system will include a rule $a \to b$, for which *buildInput* will fetch the whole multiset contained in the inner membrane h, so that the contents of this membrane get merged with the contents of the parent membrane. The associated membrane structure rule will be $(h, e) \to \lambda$.

Finally, for the division rules of type (e), $[a]_h^{e_1} \rightarrow [b]_h^{e_2}[c]_h^{e_3}$, the parent membrane of h will contain a rule $(a, h) \rightarrow (b, h)(c, h)$ with the corresponding membrane structure rule $(h, e_1) \rightarrow (h, e_2)(h, e_3)$. The function provided by *outputBuilder* for such a rule will take care of distributing the symbols b and c across the compartments in the correct order.

Note that, in this setup, the rules which do not have membrane structure rules associated, are applied first, just required by the definition of a P system with active membranes (Chapter 11 of [4]).

5 Conclusion

Instead of focusing on certain kinds of P systems, constructed by combining membrane structures with a certain type of rules, this paper has brought attention to the membrane structures themselves, as separate objects of study. This approach was motivated by the observation that it has become quite popular with researchers in the domain of computational devices to combine a known type of rules with membrane structures. A generalisation of membrane structures was provided in terms of algebraic structures and mappings and a number of known concrete P systems models were shown to be covered by the introduced formalisation.

Importantly enough, the constructs suggested in this paper do not focus on the nature of the rules on which the membrane structure acts. In fact, only some basic statements are made about the rules and the objects placed in the compartments of the membrane structure. This makes it possible to fit the majority of known P system models in the suggested formalisation.

Even more importantly, it turns out that membrane structures can indeed be quite easily separated from the rules associated with the membranes. Static membrane structures turned out to be simpler to factor out than dynamic membrane structures, a lot less additional constructions are required in the former case. However, as visible in Subsection 4.3, actually fitting a P system model with active membranes in the suggested formalisation is fairly straightforward. In fact,

the majority of rules shown in [9] can be fit into the constructions shown in this paper.

A remarkable feature of the formalised models suggested in the present work is that they are rather considerably narrowed down to cover as little as possible extra capabilities. As different from the powerful, generalised interaction rules shown in [8], the constructions in this paper only allow for tree-like membrane structures, with communication limited to the parent membranes and the immediately inner membranes.

While the formalisations exposed in this paper may not themselves come to know wide usage, the point of view will hopefully make more researchers consider membrane structures without the context of concrete P system models. There are two major reasons motivating such a shift of perspective. The first reason is that membrane structures are not just trees, as it has been shown in this paper, and have the full right to be studied on their own. The second reason is that such a view on membrane structures opens further possibilities for placing different types of rules in compartments and thus obtaining a potential plethora of results.

References

[1] Gh. Păun. *Computing with membranes.* TUCS Report 208, Turku Center for Computer Science, 1998.

[2] Gh. Păun. *Membrane Computing. An Introduction.* Springer-Verlag, 2002.

[3] M. J. Pérez-Jiménez, F. Sancho-Caparrini. *A formalization of transition P Systems.* Fundamenta Informaticae – Membrane computing, Volume 49 Issue 1, January 2002.

[4] Gh. Păun, G. Rozenberg, A. Salomaa, Eds. *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.

[5] A. Krassovitskiy, Yu. Rogozhin, S. Verlan. *Computational power of insertion-deletion (P) systems with rules of size two*. Journal Natural Computing, Volume 10 Issue 2, June 2011.

[6] B. A. Davey, H. A. Priestley. *Introduction to Lattices and Order (second ed.)*. Cambridge University Press, 2002.

[7] Eric W. Weisstein. *Tree*. From MathWorld – A Wolfram Web Resource, http://mathworld.wolfram.com/Tree.html.

[8] R. Freund and S. Verlan. *A Formal Framework for Static (Tissue) P Systems*. G. Eleftherakis, P. Kefalas, G. Paun, G. Rozenberg, A., Salomaa, eds., 8th International Workshop on Membrane Computing, WMC2007. vol 4860 of LNCS, 2007.

[9] E. Csuhaj-Varjú, A. Di Nola, Gh. Păun, M. J. Pérez-Jiménez, G. Vaszil. *Editing Configurations of P Systems*. Fundamenta Informaticae, Volume 82 Issue 1-2, July 2008.

[10] The P systems web page. `http://ppage.psystems.eu/`

Sergiu Ivanov,

Received July 6, 2012

Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Academiei 5, Chişinău MD-2028 Moldova
E–mail: *sivanov@math.md*

University of Academy of Sciences of Moldova
Faculty of Real Sciences
Academiei 3/2, MD-2028 Chişinău, Republic of Moldova

Abstracts of Doctor Habilitatus Thesis

(doctor habilitatus thesis in computer science, Chisinau, 2012)

Title: Models, algorithms and tools for database design and analysis
Author: Cotelea Vitalie
Date of defence: 18th of May, 2012
Place of defence: Academy of Economic Studies

The thesis is comprised of an introduction, four chapters, conclusions and recommendations, bibliography (234 titles), 7 annexes and consists of 230 pages, from which 186 pages cover the main part, including 29 figures. Obtained results are published in 72 scientific papers.

Keywords: Schema database design, functional dependencies, covers, normal forms, nonessential attributes, recoverable attributes, equivalence classes of attributes, degree of acyclicity, polynomial algorithms.

The area of study refers to the design of information systems in general, and databases in particular.

The aim of this work is to develop models and methods, techniques and efficient algorithms that could be applied to automate the design process and evaluation of the database schema. Achieving this goal involves the following key objectives: to examine and analyze the characteristics of database structures used in information systems, to determine and describe the problems which occur in the design of the database schema, investigation of the research level and available solutions for the identified problems, analysis of existing algorithms, presentation of scientific arguments, models, techniques, algorithms, their implementation and application in testing.

The scientific novelty and originality of obtained results consists of the presented models, methods, techniques and algorithms which are essentially new or are improving existing tools necessary for the design and analysis of database schemas. All these results have a direct contribution to the shaping of a direction of research - elaboration of adaptable databases, adaptable to changing environment in which it activates.

The theoretical signification of research presented in this thesis consists of the proved theoretical foundations of modeling and design techniques and analysis of schemes.

Solved scientific problems include: tools of functional dependencies efficient inference; techniques and algorithms to design schemes that satisfy a number of desirable features; model of attributes that dictate the behavior of relational schemes; techniques and polynomial algorithms for testing of the

degree of database normalization; efficient heuristic detection methods of determinants in database schemes; techniques and models for analysis of acyclic schemes and their adjustment in order to obtain desired and more efficient characteristics.

The practical value of the work: proposed algorithms in this thesis can be used to automate the design process of databases, create feasible and adjustable to changes databases. The results are of practical importance because software products are extensible and allow their integration in various application fields.

The scientific results of the work are implemented in several projects developed by the IT company Estcomputer SRL and in computer assisted training of students of the Academy of Economic Studies of Moldova and of the Technical University of Moldova.

Vitalie COTELEA is Associate Professor at Faculty of Cybernetics, Statistics and Economic Informatics from the Academy of Economic Studies of Moldova. He is the author and co-author of over 100 scientific works, including two monographs and more than 10 books. His work focuses on Databases and Information Systems Design and Declarative Programming. He has graduated the Faculty of Mathematics and Cybernetics in 1974 of State University of Moldova, Chisinau. He holds a PhD diploma in Computer Science from 1988 of Kiev State University, Ukraine. He defended the Doctor Habilitatus Thesis in Computer Science on the 18th of May 2012.