



Universiteit
Leiden
The Netherlands

Models of natural computation : gene assembly and membrane systems

Brijder, R.

Citation

Brijder, R. (2008, December 3). *Models of natural computation : gene assembly and membrane systems*. IPA Dissertation Series. Retrieved from <https://hdl.handle.net/1887/13345>

Version: Corrected Publisher's Version

License: [Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden](#)

Downloaded from: <https://hdl.handle.net/1887/13345>

Note: To cite this publication please use the final published version (if applicable).

Models of Natural Computation:
Gene Assembly and Membrane Systems

Robert Brijder



The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).



Netherlands Organisation for Scientific Research

This research was financed by the Netherlands Organisation for Scientific Research (NWO) under project 635.100.006 “VIEWS”.

Cover design: Peter Loonen.

ISBN 978-90-9023428-1

Models of Natural Computation: Gene Assembly and Membrane Systems

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus prof. mr. P.F. van der Heijden,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 3 december 2008
klokke 16.15 uur

door

Robert Brijder
geboren te Delft
in 1980

Promotiecommissie

Promotor:	Prof. Dr. G. Rozenberg	
Co-promotor:	Dr. H.J. Hoogeboom	
Referent:	Dr. G. Păun	(Romanian Academy)
Overige leden:	Prof. Dr. T.H.W. Bäck	
	Prof. Dr. T. Harju	(University of Turku)
	Prof. Dr. J.N. Kok	
	Prof. Dr. M. Koutny	(University of Newcastle)
	Prof. Dr. S.M. Verduyn Lunel	

This thesis is dedicated to my mother.

Contents

1	Introduction	1
1.1	Natural Computing	1
1.2	Background: Cells	2
1.2.1	Membranes	2
1.2.2	DNA and Cell Nucleus	2
1.3	Gene Assembly in Ciliates	3
1.4	Sorting by Reversal	6
1.5	Membrane Computing	8
1.6	Overview of the Thesis	10
	Bibliography	11
I	Gene Assembly in Ciliates	15
2	Reducibility of Gene Patterns in Ciliates using the Breakpoint Graph	17
2.1	Introduction	17
2.2	Background: Gene Assembly in Ciliates	19
2.3	Basic Notions and Notation	21
2.4	The String Pointer Reduction System	23
2.5	Pointer Removal Operation	25
2.6	Reduction Graphs	27
2.7	Reduction Function	31
2.8	Characterization of Reducibility	36
2.9	Cyclic Components	37
2.10	Successfulness of Legal Strings	42
2.10.1	Trivial Generalizations and Known Results	42
2.10.2	Non-Trivial Generalizations	43
2.11	Discussion	45
3	Strategies of Loop Recombination in Ciliates	47
3.1	Introduction	47
3.2	Basic Notions and Notation	48

3.3	String Pointer Reduction System	49
3.4	Reduction Graph	52
3.5	Pointer-Component Graphs	57
3.6	Spanning Trees in Pointer-Component Graphs	60
3.7	Merging and Splitting Components	62
3.8	Applicability of the String Negative Rule	66
3.9	The Order of Loop Recombination	69
3.10	Conclusion	72
4	The Fibers and Range of Reduction Graphs	73
4.1	Introduction	73
4.2	Mathematical Notation and Terminology	75
4.3	Legal strings	76
4.4	Reduction Graph	77
4.5	Abstract Reduction Graphs and Extensions	79
4.6	Back to Legal Strings	83
4.7	Flip Edges	85
4.8	Merging and Splitting Connected Components	87
4.9	Connectedness of Pointer-Component Graph	90
4.10	Flip and the Underlying Legal String	92
4.11	Dual String Pointer Rules	94
4.12	Discussion	97
5	How Overlap Determines Reduction Graphs for Gene Assembly	99
5.1	Introduction	99
5.2	Notation and Terminology	101
5.3	Gene Assembly in Ciliates	102
5.4	The Reduction Graph	104
5.5	The Reduction Graph of Realistic Strings	108
5.6	Compressing the Reduction Graph	113
5.7	From Overlap Graph to Reduction Graph	114
5.8	Consequences	119
5.9	Discussion	120
	Bibliography	121
II	Membrane Computing	125
6	Membrane Systems with Proteins Embedded in Membranes	127
6.1	Introduction	127
6.2	Preliminaries	129
6.3	Operations for Marked Membranes	131
6.4	Membrane Systems with Marked Membranes	132
6.5	Preliminary Results	134

6.6	Membrane Systems Using Protein-Membrane Rules	135
6.7	Using Protein-Membrane and Protein Movement Rules	139
6.8	Decision Problems	143
6.9	Concluding Remarks	149
7	Membrane Systems with External Control	153
7.1	Introduction	153
7.2	Preliminaries	155
7.3	String-Controlled P Systems	157
7.4	Fully-Promoted SC P Systems	159
7.5	The Influence of the Control Program	163
7.6	Fully-Promoted SC P Systems: Universality	168
7.7	Concluding Remarks and Open Problems	170
8	Communication Membrane Systems with Active Symports	173
8.1	Introduction	173
8.2	Preliminaries	175
8.2.1	Matrix Grammars	175
8.2.2	Register Machines	176
8.3	Communication Membrane Systems with Active Symports	177
8.4	Alphabetic Restriction	180
8.5	The Sequential Mode	182
8.6	Unary Rules	185
8.7	Unidirectional Membranes	187
8.8	Noncooperative Rules	189
8.9	Deciding Boundness	190
8.10	Discussion	192
	Bibliography	195
	Nederlandse Samenvatting	199
	Curriculum Vitae	203
	Publication List	205

Chapter 1

Introduction

The two main topics of this thesis, gene assembly in ciliates and membrane computing, are representatives of the broad research field of natural computing. Membrane computing is a computational model inspired by the functioning of membranes in living cells, and gene assembly is a complex biological process occurring in unicellular organisms called ciliates.

1.1 Natural Computing

Natural computing is a broad and diverse research discipline residing on the boundary of computer science and natural sciences. Therefore, by its very nature, natural computing is interdisciplinary and it builds bridges between computer science and natural sciences – here computer science is meant as a broadly understood science of information processing. In natural computing one can distinguish two main research directions. On one hand, it considers processes taking place in nature as (some sort of) computation, while on the other hand it is concerned with developing and analyzing computational methods inspired by nature [16].

This thesis considers two research areas within natural computing: gene assembly in ciliates, representing the first research direction given above, and membrane computing, representing the second research direction.

In the following section we recall some very basic cell biology underlying the theory presented in this thesis. Then, in Section 1.3 we provide a basic description of the gene assembly process, and in Section 1.4 we discuss sorting by reversal which is strongly related to our theoretical model of gene assembly. In Section 1.5 we discuss the generic membrane computing model. We conclude this chapter with an outline of the thesis.

1.2 Background: Cells

Each organism consists of one or more cells. The tiniest organisms are unicellular, they consist of just one cell, while, e.g., the number of cells in humans is of the order 10^{14} . On one hand cells can be seen as building blocks for complex organisms such as human beings – cells in such organisms have their own function, and together they form more complex organizations such as tissues, organs, etc. On the other hand, cells themselves are amazingly complex – they have an involved internal structure. Two substructures of (eukaryotic) cells will be most relevant for us: cell membranes and the cell nucleus. A standard text concerning the molecular biology of the cell is [1]. A more accessible text for a computer scientist is Chapter 1 of [10].

1.2.1 Membranes

Membranes separate cells from their environment, but membranes also divide a cell into compartments. Each compartment may have its own structure and function, and either requires or avoids the presence of certain ions and molecules. Communication between compartments of cells or between a cell and its outside environment is facilitated by various kinds of channels. They allow for controlled passage of ions and molecules from one compartment to another, or between the cell and its outside environment. Different channels may control the passage of different molecules or ions.

1.2.2 DNA and Cell Nucleus

A single-stranded DNA molecule (where DNA stands for deoxyribonucleic acid) is a chain of basic components (monomers) called nucleotides. There are four types of nucleotides: adenine, cytosine, guanine, and thymine, abbreviated as *A*, *C*, *G*, and *T*, respectively. A DNA molecule can thus be represented as a sequence of symbols *A*, *C*, *G*, and *T*. For example, the sequence (string) *GACGT* represents a single-stranded DNA molecule, which is the chain of nucleotides *G*, *A*, *C*, *G*, *T* (in this order).

A single stranded DNA molecule has a natural orientation, meaning that one end of it is (chemically) distinguishable from the other – one of the ends is called 5' and the other one 3'. Almost all information processing of DNA molecules in nature happens in the direction from 5' to 3', and for this reason the reading of the sequence of the nucleotides comprising a DNA molecule goes from its 5' end to its 3' end. Hence, DNA molecule *GACGT* is *not* equal to its reverse *TGCAG*.

A basic feature of single stranded DNA molecules is that each such molecule has a complementary single stranded DNA molecule. Together they can form a double stranded DNA molecule. Here, two complementary single stranded DNA molecules bind together by weak hydrogen bonds between complementary nucleotides: nucleotides *A* and *T* are complementary, and *C* and *G* are complemen-



Figure 1.1: Two single-stranded DNA molecules forming a double-stranded DNA molecule.

tary. Moreover, the two complementary single stranded DNA molecules bind in their opposite orientation – meaning that the first (second, ..., resp.) nucleotide on the 5' end of one molecule sticks to the first (second, ..., resp.) nucleotide on the 3' end of the other molecule. This is illustrated in Figure 1.1 with the complementary DNA molecules $GACGT$ and $ACGTC$. We use the arrows as the standard notation for indicating the 5'-3' orientation of a single-stranded DNA molecule. Since strings are used to denote/specify single-stranded DNA molecules, the double string notation is very natural for denoting double-stranded DNA molecules. Thus, the double-stranded DNA molecule in the Figure 1.1 is denoted by either

$\begin{array}{c} GACGT \\ CTGCA \end{array}$ or $\begin{array}{c} ACGTC \\ TGCAG \end{array}$, since double-stranded DNA molecules do not have an orientation. Of course, segments within a double-stranded DNA molecule α do have an orientation (w.r.t. α), e.g., although double-stranded DNA molecule $\begin{array}{c} AC \\ TG \end{array}$ can also be represented as $\begin{array}{c} GT \\ CA \end{array}$, only $\begin{array}{c} AC \\ TG \end{array}$ appears in $\begin{array}{c} GACGT \\ CTGCA \end{array}$ which

is *not* equal to $\begin{array}{c} GGTGT \\ CCACA \end{array}$. For this reason, we sometimes fix an orientation of a double-stranded DNA molecule, i.e., choose one of the two representations of the molecule. If we let M be a double-stranded DNA molecule with a fixed orientation, then we define the *inversion* of M , denoted by $\bar{M}$, to be the same double-stranded DNA molecule with the other orientation, i.e., M rotated 180 degrees.

The cell nucleus is a substructure of the cell holding the genome. The genome is divided into a number of chromosomes, e.g., the human genome consists of 46 chromosomes. Each chromosome contains one double-stranded DNA molecule. These DNA molecules contain genes which are segments containing “instructions” for the production (expression) of proteins. The genetic part of chromosomal DNA may be very small (e.g., in humans only about 2%-5% is genetic). It also contains regulatory information (when and how much of specific proteins should be produced), but the role of the non-genetic part of chromosomal DNA is not yet well understood.

1.3 Gene Assembly in Ciliates

Ciliates (ciliated protozoa) are a group of ancient unicellular organisms. The name ciliates is due to the hair-like structure, called cilia, present on their external surface. Ciliates are different from other organisms in that they have two kinds of

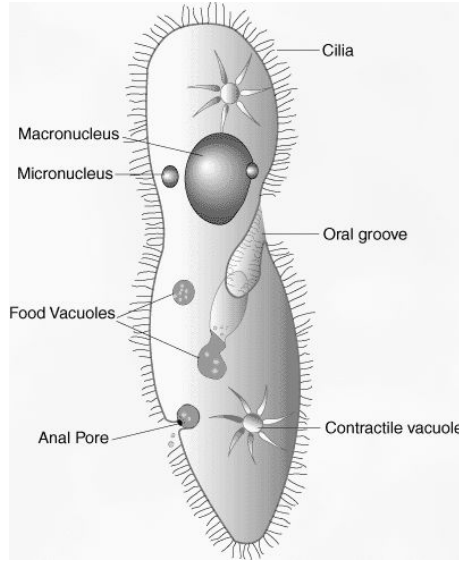


Figure 1.2: Schematic image of a ciliate, copyright SparkNotes.

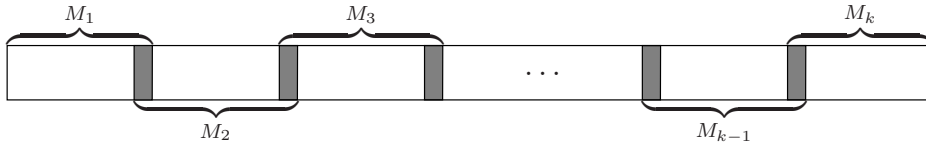


Figure 1.3: The structure of a MAC gene consisting of κ MDSs.

nuclei that are radically different, both functionally and physically. The two kinds of nuclei (which both can be present in various multiplicities) are called micronucleus (MIC) and macronucleus (MAC) – the former is used only in mating, while the latter is used for producing RNA needed for cell maintenance and reproduction. A schematic image of a ciliate is given in Figure 1.2.

The number of chromosomes in a MIC is similar to that of other eukaryotes (say about 100), while there are very many (millions) of minichromosomes in the MAC. Also, the MIC chromosomes are very long (as is generally the case for eukaryotes) and for the most part (more than 95%) non-genetic. The MAC minichromosomes are very short (on average about 2000 base pairs but may be as small as 300 base pairs), and for the most part (about 85%) genetic.

All the genes occur in both the MIC and the MAC, but in very different forms. The relationship between the form of MIC and MAC genes can be described as follows. For each gene, one can distinguish a number of double stranded DNA molecules $M_1, \dots, M_\kappa$ with a fixed orientation, called MDSs (macronuclear destined



Figure 1.4: The structure of the MIC gene encoding for the actin protein in *sterkiella nova*.

segments), appearing in both the MIC and MAC form of that gene. The MAC form is a sequence of overlapping MDSs in their orthodox order, i.e., $M_1, \dots, M_\kappa$ – this is illustrated in Figure 1.3. The gray areas in the figure indicate the overlaps of MDSs – these overlaps are called pointers. In the MIC form the MDSs are separated by non-coding segments, called IESs (internal eliminated segments). The MDSs either occur in orthodox order or in a different order, and the MDSs can occur inverted (no MDS can occur twice). As an example, Figure 1.4 shows the MIC form of the gene that encodes for the actin protein in a ciliate called *sterkiella nova*. This gene consists of nine segments, where the enumeration $M_1, M_2, \dots, M_9$ refers to the orthodox order of the MDSs in the MAC form of the gene. Note that MDS M_2 occurs inverted in the MIC form of the gene. It is important to realize that the number of MDSs, the specific permutation of the MDSs and the possible inversions are fixed for a given gene and given species, but they can be very different for different genes and for the same gene in different species.

The process of gene assembly transforms a MIC into a MAC. This process occurs during sexual reproduction of two ciliates where first a MIC is formed holding half of the genetic information of each parent, and then a MAC is constructed from this newly formed MIC. During gene assembly, each of the about 25,000 genes in MIC form are transformed into the corresponding gene in MAC form. The transformation of a single gene from MIC form to MAC form is complex: all MDSs must be ‘sorted’ in the right order and must have the right orientation, and all IESs must be spliced out from between the MDSs. This transformation process involves quite a number of “cutting and gluing” of DNA. Pointers in the MIC form indicate how this cutting and gluing, called recombination, is done. Indeed, each overlapping segment of two MDSs in the MAC form appears in two places in the MIC form and this in turn indicates where the DNA segments are to be cut and glued together. The differences in the genetic material of the MIC and the MAC discussed above are particularly pronounced in the stichotrichs group of ciliates. For this reason, a lot of literature, including this thesis, concerns this group of ciliates. We refer to [10] for an in-depth treatment of the biology of gene assembly.

In one possible modeling of the assembly process, the MIC form of a gene is transformed into the MAC form through three types of recombination operations that operate on the pointers. These types of operations are called: loop recombination, hairpin recombination, and double-loop recombination. Each of these recombinations can only take place on pointers of the gene in MIC form (or an intermediate product) provided that these pointers fulfill specific conditions. The

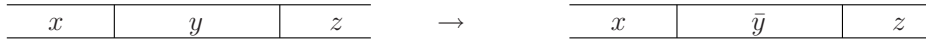


Figure 1.5: Inversion within a chromosome.

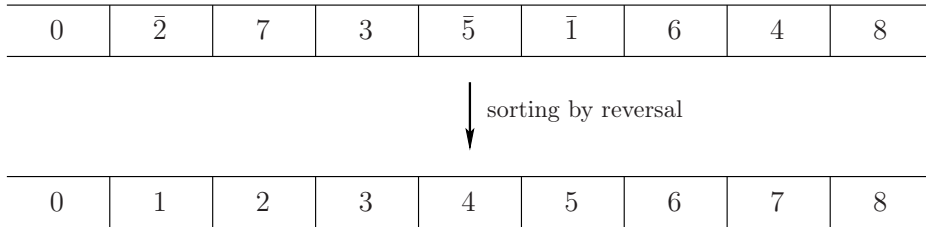


Figure 1.6: Two chromosomes of different species and their common contiguous segments.

operations are defined in [10] and we will recall them in Part 1 of this thesis.

1.4 Sorting by Reversal

During evolution the genomes of species change. One such change is inversion, and is illustrated in Figure 1.5. The result is that a segment y is inverted (rotated 180 degrees) – this is indicated by $\bar{y}$ in the figure. In this way, two different species can have several contiguous segments in their genome that are very similar, although their relative order (and orientation) may differ in both genomes. For example, consider the two chromosomes in Figure 1.6. Both chromosomes have 9 segments in common, however their relative order and orientation differs. The breakpoints of a chromosome are the borders of each two consecutive segments. Figure 1.7 shows the application of an inversion, called reversal, on the breakpoint between segments 0 and $\bar{2}$ and the breakpoint between segments $\bar{1}$ and 6 (these two breakpoints are indicated by two small arrows in the figure).

In the theory of sorting by reversal, initiated by S. Hannenhalli and P.A. Pevzner in [11], one tries to determine the minimal number of reversals needed to convert the genome of one species into that of the other. The smaller this number, the more likely it is that their common ancestor is relatively young in evolution. Thus, this number can aid in constructing an ancestor tree of species, called a phylogenetical tree.

Note that sorting by reversal differs from gene assembly in ciliates in several aspects. First, it is an evolutionary process from one species to another; there are no pointers that indicate where recombination should take place. Second, recombination takes place on the scale of complete chromosomes, while in gene

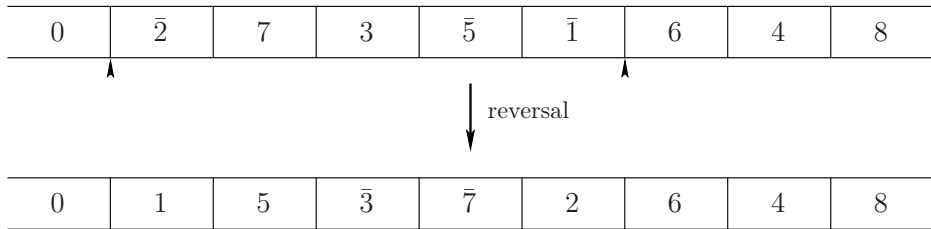


Figure 1.7: Applying a reversal on the chromosome.

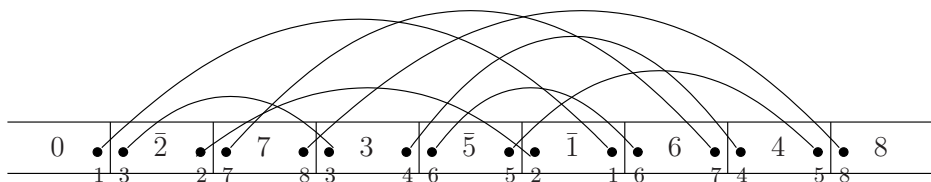


Figure 1.8: The breakpoint graph of the given chromosome.

assembly it is on the level of individual genes. And finally, instead of three types of recombination operations there is only one type: the reversal.

An essential tool in the theory of sorting by reversal is the breakpoint graph (also called reality and desire diagram) which is used to capture both the present situation, the genome of the first species, and the desired situation, the genome of the second species. For each breakpoint, we assign two vertices in the graph representing both sides of that breakpoint. These vertices are labeled such that segment i has vertices labelled by i and $i + 1$. Then i represents the left-hand side and $i + 1$ the right-hand side of segment i . If segment i appears inverted in the genome then, w.r.t. the chromosome, i appears on the right-hand side and $i + 1$ on the left-hand side. Moreover, there are edges, called desire edges, that connect vertices with the same label. In Figure 1.8 these vertices and edges are depicted for our example*.

In addition to the desire edges, the breakpoint graph has a second set of edges, called reality edges. These edges connect each two vertices belonging to the same breakpoint. Thus, in Figure 1.8, the left-most two vertices labeled by 1 and 3 are connected by a reality edge, and similarly for the next two vertices labeled by 2 and 7, etc. The linear order of the vertices in the figure is therefore partially captured by the reality edges. However, the complete linear order of the vertices remains important, and therefore the breakpoint graph should not be seen as a graph, but

*It is customary for breakpoint graphs to let $2i - 1$ represent the left-hand side and $2i$ the right-hand side of segment i – in this way eliminating the need for labels. However, we choose this notation to make comparison with reduction graphs (defined in the next chapter) easier.

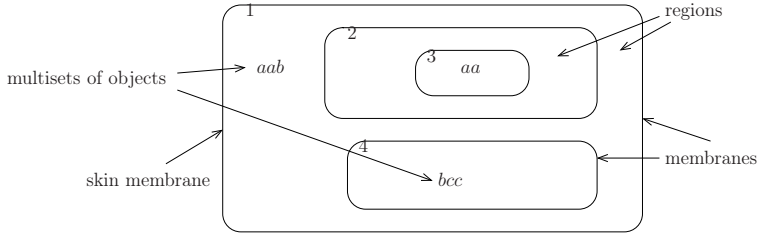


Figure 1.9: Example membrane system.

as a diagram where the vertices are drawn in this linear order. Therefore reality and desire *diagram* is arguably a more appropriate name for this concept. One could extend the breakpoint graph with a third set of edges, for example called segment edges, connecting each two consecutive vertices belonging to the same segment. Thus, e.g., in Figure 1.8 the two vertices labeled by 3 and 2 of segment $\bar{2}$ are then connected by such a segment edge. In this way, we obtain a graph which retains the linear order of the vertices, and hence need not be seen as a diagram. We will introduce these additional sets of edges in the context of gene assembly in this thesis. Given only the breakpoint graph it is possible to deduce, in a computationally efficient way, the minimal number of reversals needed to convert the genome from one species into that of the other.

1.5 Membrane Computing

Membrane computing studies a range of computational models inspired by the functioning of membranes in cells. This research area was initiated by Gh. Păun in 1998 (see [13]). Membrane systems are therefore also often called P systems after its inventor. Membrane computing has in a short time attracted a large research community. Many classes of membrane systems exist, but in this section we consider a ‘typical/generic’ membrane system; for an in-depth introduction to membrane computing we refer to [14], and for an easier-to-read overview we refer to [15].

Such a membrane system consists of a hierarchical membrane structure where each membrane, except for the outer membrane (called the skin membrane), is fully contained in another membrane (called its parent). The compartments enclosed by (situated in-between) the membranes are called regions. An example of a membrane structure is given in Figure 1.9.

Each region contains zero or more objects, and each object is of a certain type. In the figure the region enclosed by the skin membrane (called skin region) contains two objects of type a , and one object of type b . To make the system evolve/compute there are evolution rules assigned to the regions that in some way transform, create, delete, or move the objects (between regions – moving

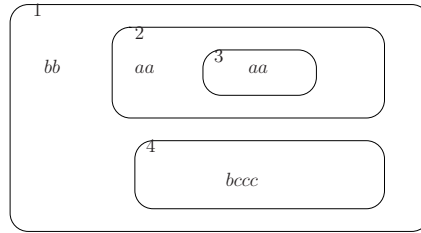


Figure 1.10: A possible state of the membrane system in Figure 1.9 after one time step.

objects between adjacent regions is referred to as communication).

In each time step (there is a global clock) during the evolution of a membrane system many such evolution rules can be applied in parallel. In fact, in each time step the evolution rules are applied in a *maximal* parallel manner: the multiset of evolution rules that is applied cannot be extended by any evolution rule – no subset of the objects that remain unused in a given time step can evolve using any evolution rule.

For example, there could be two evolution rules in the skin region: one that transforms one object of type a and one of type b into two objects of type a , both of which cross membrane 2, *and* one that transforms one object of type a into two objects of type b (both staying in the skin region). In addition there could be an evolution rule in the region enclosed by membrane 4 transforming one object of type b and one of type c into one object of type b and two objects of type c . Then, there are two possible maximal parallel ways to transform this membrane system. In the next time step, the state of the membrane system is either the one given in Figure 1.10 or the one given in Figure 1.11.

A membrane system computes by iteratively applying the evolution rules in a maximal parallel manner until no evolution rule can be applied anymore. Then, the contents of a preselected membrane, called the output membrane, is the result of the computation. In this way, the language of a given membrane system is defined to be the set of results of all computations of the membrane system.

A well-studied class of membrane systems called symport/antiport P systems involves only communication and no transformation, see [12]. Here, the rules are assigned to membranes instead of regions, and they allow movement of objects from a region on one side of the membrane to the region on the other side. The movement of objects is also synchronized. For example, an object a may only move together with object b to the other side of the membrane. Or, for example, an object a may only move through the membrane if simultaneously an object b from the other side of the membrane moves through the membrane in the opposite direction. The former type of movement is described by the so-called symport rules, while the latter type of movement is described by antiport rules. Note that

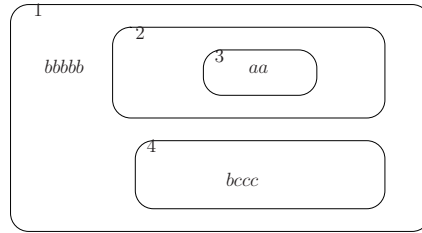


Figure 1.11: A possible state of the membrane system in Figure 1.9 after one time step.

these rules cannot change either the type of an individual object or the quantity of objects present in the system.

1.6 Overview of the Thesis

This thesis consists of two parts.

The first part, consisting of Chapters 2 through 5, is devoted to gene assembly in ciliates. The central notion of this part is the reduction graph – it is inspired by the breakpoint graph discussed in Section 1.4. The concept of reality and desire remains in place, but the notion of reduction graph is specifically tailored for the theory of gene assembly. Given the MIC form (reality) of a gene, the reduction graph describes the end result (desire) after gene assembly for this gene is completed. This includes the MAC form of a gene, but it also describes the “end structure” of all the IESs. The reduction graph however is defined in a more general fashion: it deals with arbitrary recombination using pointers. In the model we use, the MIC form of the gene is represented by a string, called legal string, and the reduction graph is defined for each such legal string. In Chapter 2 we introduce the reduction graph, and then use it to characterize the intermediate gene patterns that may occur during the transformation of a MIC form of a gene to its MAC form. We also show that for legal strings in general the number of loop recombination operations in each possible strategy transforming the MIC form into the MAC form is fixed and directly determinable through the reduction graph. This chapter is based on [7]. In Chapter 3 we strengthen these results in order to obtain a characterization of loop recombination that allows one to determine which loop recombination operations can be applied in such strategies and also in which order they can be applied. This is done by using the notion of pointer-component graph (defined “on top of” the reduction graph) that identifies the relationship of pointers on the connected components of the reduction graph. This chapter is based on [6]. Since the reduction graph is the main notion of the first part of the thesis, it is certainly natural to ask which graphs are reduction graphs. Such a characterization of reduction graphs is given in Chapter 4. Also,

in Chapter 4 we consider the problem of equivalence for MIC genes: we characterize which genes in MIC form (formally legal strings) yield the same end result after gene assembly is accomplished. This characterization is given in terms of string rewriting rules (applied to legal strings) that correspond to recombination operations which, surprisingly, are very similar to the recombination operations defining gene assembly. This chapter is based on [5]. The MIC forms of genes can be represented both as strings, called legal strings, and as graphs, called signed overlap graphs. Both representations lead to two almost equivalent models of gene assembly. The definition of reduction graph in Chapter 2 relies on string representations. In Chapter 5 we define the reduction graph directly for overlap graphs, and show that this graph is identical to the reduction graph of every “realistic” legal string corresponding to that overlap graph. This allows one to carry over the results of Chapters 2, 3, and 4 to the graph based model of gene assembly. This chapter is based on [9] (see [8] for an extended abstract).

The second part of this thesis, consisting of Chapters 6 to 8, is devoted to membrane computing. Chapter 6 considers membrane systems that can have objects not only within the regions (which is standard in membrane systems) but also on/within the membranes themselves. Such objects allow for both controlled movement of objects through the membranes and controlled evolution of the membranes. These systems are biologically motivated by the fact that some proteins (represented by objects) residing on/within membranes control the movement of ions/molecules through membranes. This chapter is based on [4]. Chapter 7 considers membrane systems where the evolution of the system depends on external signals. Each signal is represented by a sequence (string) of objects which enters the system from outside and during the evolution of the system moves through the regions. Here the first object of the signal has influence on the system and this object is removed when passing through a membrane until finally the whole “string signal” has disappeared. This chapter is based on [3]. Chapter 8 focusses on membrane systems with symports and antiports where we relax the condition that symports only move objects – we allow now that during the crossing of a membrane the objects themselves can change in both type and quantity. The intuitive interpretation is that objects can engage in (biochemical) reactions while crossing the membrane. This chapter is based on [2]. The central unifying research topic (question) which we consider in Part 2 is the computational power (including decidability results) of the various classes of membrane systems described above.

Bibliography

- [1] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *The Molecular Biology of the Cell*. Garland Publ. Inc., London, 4th edition, 2002.
- [2] R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, and D. Sburlan. Communication membrane systems with active symports. *Journal of Automata, Languages and Combinatorics*, 11(3):241–261, 2006.

- [3] R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, and D. Sburlan. Membrane systems with external control. In H.J. Hoogeboom, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Workshop on Membrane Computing*, volume 4361 of *Lecture Notes in Computer Science*, pages 215–232. Springer, 2006.
- [4] R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, and D. Sburlan. Membrane systems with proteins embedded in membranes. *Theoretical Computer Science*, 404:26–39, 2008.
- [5] R. Brijder and H.J. Hoogeboom. The fibers and range of reduction graphs in ciliates. *Acta Informatica*, 45:383–402, 2008.
- [6] R. Brijder, H.J. Hoogeboom, and M. Muskulus. Strategies of loop recombination in ciliates. *Discrete Applied Mathematics*, 156:1736–1753, 2008.
- [7] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. Reducibility of gene patterns in ciliates using the breakpoint graph. *Theoretical Computer Science*, 356:26–45, 2006.
- [8] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. From micro to macro: How the overlap graph determines the reduction graph in ciliates. In E. Csuhaj-Varjú and Z. Ésik, editors, *Fundamentals of Computation Theory (FCT) 2007*, volume 4639 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 2007.
- [9] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. How overlap determines the macronuclear genes in ciliates. Submitted, also LIACS Technical Report 2007-02, [arXiv:cs.LO/0702171], 2008.
- [10] A. Ehrenfeucht, T. Harju, I. Petre, D.M. Prescott, and G. Rozenberg. *Computation in Living Cells – Gene Assembly in Ciliates*. Springer Verlag, 2004.
- [11] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *J. ACM*, 46(1):1–27, 1999.
- [12] A. Păun and Gh. Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–305, 2002.
- [13] Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000. Also, Turku Center for Computer Science-TUCS Report No. 208, 1998.
- [14] Gh. Păun. *Membrane Computing. An Introduction*. Springer, Berlin, 2002.
- [15] Gh. Păun and G. Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287(1):73–100, 2002.

-
- [16] G. Rozenberg. Computer science, informatics, and natural computing - personal reflections. In S.B. Cooper, B. Löwe, and A. Sorbi, editors, *New Computational Paradigms - Changing Conceptions of What is Computable*, pages 373–379. Springer, 2007.

Part I

Gene Assembly in Ciliates

Chapter 2

Reducibility of Gene Patterns in Ciliates using the Breakpoint Graph

Abstract

Gene assembly in ciliates is one of the most involved DNA processings going on in any organism. This process transforms one nucleus (the micronucleus) into another functionally different nucleus (the macronucleus). We continue the development of the theoretical models of gene assembly, and in particular we demonstrate the use of the concept of the breakpoint graph, known from another branch of DNA transformation research. More specifically: (1) we characterize the *intermediate* gene patterns that can occur during the transformation of a *given* micronuclear gene pattern to its macronuclear form; (2) we determine the number of applications of the loop recombination operation (the most basic of the three molecular operations that accomplish gene assembly) needed in this transformation; (3) we generalize previous results (and give elegant alternatives for some proofs) concerning characterizations of the micronuclear gene patterns that can be assembled using a specific subset of the three molecular operations.

2.1 Introduction

Ciliates are single cell organisms that have two functionally different nuclei, one called micronucleus and the other called macronucleus (both of which can occur in various multiplicities). At some stage in sexual reproduction a micronucleus is transformed into a macronucleus in a process called gene assembly. This is the most involved DNA processing in living organisms known today. The reason that gene assembly is so involved is that the genome of the micronucleus may be dramatically different from the genome of the macronucleus — this is particularly

true in the stichotrichs group of ciliates, which we consider in this chapter. The investigation of gene assembly turns out to be very exciting from both biological and computational points of view.

Another research area concerned with transformations of DNA is *sorting by reversal*, see, e.g., [23, 21, 1]. Two different species can have several contiguous segments in their genome that are very similar, although their relative order (and orientation) may differ in both genomes. In the theory of sorting by reversal one tries to determine the number of operations needed to reorder such a series of genomic ‘blocks’ from one species into that of another. An essential tool is the *breakpoint graph* (or reality and desire diagram) which is used to capture both the present situation, the genome of the first species, and the desired situation, the genome of the second species.

Motivated by the breakpoint graph, we introduce the notion of *reduction graph* into the theory of gene assembly. The intuition of ‘reality and desire’ remains in place, but the technical details are different. Instead of one operation, the reversal, we have three operations. Furthermore, these operations are irreversible and can only be applied on special positions in the string, called *pointers*. Also, instead of two different species, we deal with two different nuclei — the reality is a gene in its micronuclear form, and desire is the same gene but in its macronuclear form. Surprisingly, where the breakpoint graph in the theory of sorting by reversal is mostly useful to determine the number of needed operations, the reduction graph has different uses in the theory of gene assembly, providing valuable insights into the gene assembly process. Adapted from the theory of sorting by reversal, and applied to the theory of gene assembly in ciliates, we hope the reduction graph can serve as a ‘missing link’ to connect the two fields.

For example, the reduction graph allows for a direct characterization of the *intermediate* strings that may be constructed during the transformation of a given gene from its micronuclear form to its macronuclear form (Theorem 11). Also, it makes the number of loop recombination operations (see Figure 2.3 below) needed in this transformation quite explicit as the number of cyclic (connected) components in the reduction graph (Theorem 18).

Each micronuclear form of a gene defines a sequence of (oriented) segments, the boundaries of which define the pointers where splicing takes place. In abstract representation, the gene defines a so-called *realistic* string in which every pointer is denoted by a single symbol. Each pointer occurs twice (up to inversion) in that string. Not every string in which each symbol has two occurrences (up to inversion) can be obtained as the representation of a micronuclear gene. Our results are obtained in the larger context, i.e., they are not only valid for realistic strings, but for *legal* strings in general.

The chapter is organized as follows. In Section 2.2 we briefly discuss the basics of gene assembly in ciliates, and describe three molecular operations stipulated to accomplish gene assembly. The reader is referred to monograph [12] for more background information. In Section 2.3 we recall some basic notions and notation concerning strings and graphs, and then in Section 2.4 we recall the string

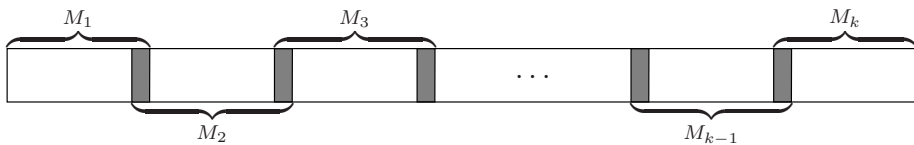


Figure 2.1: The MAC form of genes.

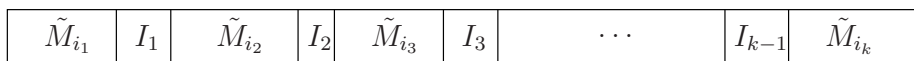


Figure 2.2: The MIC form of genes.

pointer reduction system, which is a formal model of gene assembly. This model is used throughout the rest of this chapter. In Section 2.5 we introduce the operation of pointer removal, which forms a useful formal tool in this chapter. Then in Sections 2.6 and 2.7 we introduce our main construct, the reduction graph, and discuss the transformations of it that correspond to the three molecular operations. In Section 2.8 we provide a characterization of intermediate forms of a gene resulting from its assembly to the macronuclear form — then, in Section 2.9 we determine the number of loop recombination operations required in this assembly. As an application of this last result, in Section 2.10 we generalize some well-known results from [13] (and Chapter 13 in [12]) as well as give elegant alternatives for these proofs. A conference edition of this chapter, containing selected results without proofs, was presented at CompLife [5].

2.2 Background: Gene Assembly in Ciliates

This section discusses the biological origin for the string pointer reduction system, the formal model we discuss in Section 2.4 and use throughout this chapter. Let us recall that the *inversion* of a double stranded DNA sequence M , denoted by $\bar{M}$, is the point rotation of M by 180 degrees. For example, if $M = \begin{smallmatrix} GACGT \\ CTGCA \end{smallmatrix}$,

then $\bar{M} = \begin{smallmatrix} ACGTC \\ TGCAG \end{smallmatrix}$.

Ciliates are unicellular organisms (eukaryotes) that have two kinds of functionally different nuclei: the micronucleus (MIC) and the macronucleus (MAC). All the genes occur in both MIC and MAC, but in very different forms. For a given individual gene (in given species) the relationship between its MAC and MIC form can be described as follows.

The MAC form G of a given gene can be represented as the sequence $M_1, M_2, \dots, M_k$ of overlapping segments (called MDSs) which form G in the way shown in Figure 2.1 (where the overlaps are given by the shaded areas). The MIC form g of the same gene is formed by a specific permutation $M_{i_1}, \dots, M_{i_k}$ of $M_1, \dots, M_k$ in the way shown in Figure 2.2, where $I_1, I_2, \dots, I_{k-1}$ are segments of DNA (called

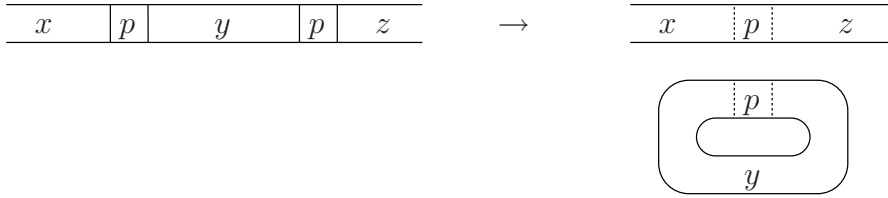


Figure 2.3: The loop recombination operation.

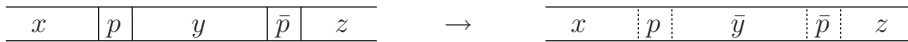


Figure 2.4: The hairpin recombination operation.

IESs) inserted in-between segments $\tilde{M}_{i_1}, \dots, \tilde{M}_{i_k}$ with each $\tilde{M}_i$ equal to either M_i or $\bar{M}_i$ (the inversion of M_i). As clear from Figure 2.1, each MDS M_i except for M_1 and M_k (the first and the last one) begins with the overlap with M_{i-1} and ends with the overlap with M_{i+1} — these overlap areas are called pointers; the former is the incoming pointer of M_i denoted by p_i , and the latter is the outgoing pointer of M_i denoted by p_{i+1} . Then M_1 has only the outgoing pointer p_2 , and M_k has only the incoming pointer p_k .

The MAC is the (standard eukaryotic) ‘household’ nucleus that provides RNA transcripts for the expression of proteins — hence MAC genes are functional expressible genes. On the other hand the MIC is a dormant nucleus where no production of RNA transcripts occurs. As a matter of fact MIC becomes active only during sexual reproduction. Within a part of sexual reproduction in a process called *gene assembly*, MIC genes are transformed into MAC genes (as MIC is transformed into MAC). In this transformation the IESs from the MIC gene g (see Figure 2.2) must be excised and the MDSs must be spliced (overlapping on pointers) in their order $M_1, \dots, M_k$ to form the MAC gene G (see Figure 2.1).

The gene assembly process is accomplished through the following three molecular operations, which through iterative applications beginning with the MIC form g of a gene, and going through intermediate forms, lead to the formation of the MAC form G of the gene.

Loop recombination The effect of the loop recombination operation is illustrated in Figure 2.3. The operation is applicable to a gene pattern (i.e., MIC or an intermediate form of a gene) which has two identical pointers p , p separated by a single IES y . The application of this operation results in the excision from the DNA molecule of a circular molecule consisting of y (and a copy of the involved pointer) only.

Hairpin recombination The effect of the hairpin recombination operation is

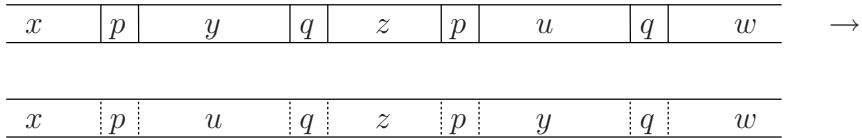


Figure 2.5: The double-loop recombination operation.

illustrated in Figure 2.4. The operation is applicable to a gene pattern containing a pair of pointers $p, \bar{p}$ in which one pointer is an inversion of the other. The application of this operation results in the inversion of the DNA molecule segment that is contained between the mentioned pair of pointers.

Double-loop recombination The effect of the double-loop recombination operation is illustrated in Figure 2.5. The operation is applicable to a gene pattern containing two identical pairs of pointers for which the segment of the molecule between the first pair of pointers overlaps with the segment of the molecule between the second pair of pointers. The application of this operation results in interchanging the segment of the molecule between the first two (of the four) pointers in the gene pattern and the segment of the molecule between the last two (of the four) pointers in the gene pattern.

For a given MIC gene g , a sequence of (applications of) these molecular operations is *successful* if it transforms g into its MAC form G . The gluing of MDS M_j with MDS M_{j+1} on the common pointer p_{j+1} results in a composite MDS. This means that after gluing, the outgoing pointer of M_j and the incoming pointer of M_{j+1} are not pointers anymore, because pointers are always positioned on the boundary of MDSs (hence they are adjacent to IESs). Therefore, the molecular operations can be seen as operations that remove pointers. This is an important property of gene assembly which is crucial in the formal models of the gene assembly process (see [12]).

2.3 Basic Notions and Notation

In this section we recall some basic notions concerning functions, strings, and graphs. We do this mainly to set up the basic notation and terminology for this chapter.

The empty set will be denoted by $\emptyset$. The composition of functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ is the function $gf : X \rightarrow Z$ such that $(gf)(x) = g(f(x))$ for every $x \in X$. The restriction of f to a subset A of X is denoted by $f|A$.

We will use λ to denote the empty string. For strings u and v , we say that v is a *substring* of u if $u = w_1vw_2$, for some strings w_1, w_2 ; we also say that v *occurs in* u . For a string $x = x_1x_2 \dots x_n$ over Σ with $x_1, x_2, \dots, x_n \in \Sigma$, we say

that substrings $x_{i_1} \cdots x_{j_1}$ and $x_{i_2} \cdots x_{j_2}$ of x overlap in x if $i_1 < i_2 < j_1 < j_2$ or $i_2 < i_1 < j_2 < j_1$.

For alphabets Σ and Δ , a *homomorphism* is a function $\varphi : \Sigma^* \rightarrow \Delta^*$ such that $\varphi(xy) = \varphi(x)\varphi(y)$ and for all $x, y \in \Sigma^*$. Let $\varphi : \Sigma^* \rightarrow \Delta^*$ be a homomorphism. If there is a $\Gamma \subseteq \Sigma$ such that

$$\varphi(x) = \begin{cases} x & x \notin \Gamma \\ \lambda & x \in \Gamma \end{cases},$$

then φ is denoted by erase_Γ .

We move now to graphs. A *labelled graph* is a 4-tuple $G = (V, E, f, \Psi)$, where V is a finite set, Ψ is an alphabet, E is a finite subset of $V \times \Psi^* \times V$, and $f : D \rightarrow \Gamma$, for some $D \subseteq V$ and some alphabet Γ , is a partial function on V . The elements of V are called *vertices*, and the elements of E are called *edges*. Function f is the *vertex labelling function*, the elements of Γ are the *vertex labels*, and the elements of Ψ^* are the *edge labels*.

For $e = (x, u, y) \in V \times \Psi^* \times V$, x is called the *initial vertex* of e , denoted by $\iota(e)$, y is called the *terminal vertex* of e , denoted by $\tau(e)$, and u is called the *label* of e , denoted by $\ell(e)$. Labelled graph $G' = (V', E', f|_{V'}, \Psi)$ is an *induced subgraph* of G if $V' \subseteq V$ and $E' = E \cap (V' \times \Psi^* \times V')$. We also say that G' is the *subgraph of G induced by V'* .

A *walk* in G is a string $\pi = e_1 e_2 \cdots e_n$ over E with $n \geq 1$ such that $\tau(e_i) = \iota(e_{i+1})$ for $1 \leq i < n$. The *label* of π is the string $\ell(\pi) = \ell(e_1)\ell(e_2) \cdots \ell(e_n)$. Vertex $\iota(e_1)$ is called the *initial vertex* of π , denoted by $\iota(\pi)$, vertex $\tau(e_n)$ is called the *terminal vertex* of π , denoted by $\tau(\pi)$ and we say that π is a *walk between $\iota(\pi)$ and $\tau(\pi)$* (or that π is a *walk from $\iota(\pi)$ to $\tau(\pi)$*). We say that G is *weakly connected* if for every two vertices v_1 and v_2 of G with $v_2 \neq v_1$, there is string $e_1 e_2 \cdots e_n$ over $E \cup \{(\tau(e), \ell(e), \iota(e)) \mid e \in E\}$ with $n \geq 1$, $\iota(e_1) = v_1$, $\tau(e_n) = v_2$, and $\tau(e_i) = \iota(e_{i+1})$ for $1 \leq i < n$. A subgraph H of G induced by $V_H \subseteq V$ is a *component* of G if H is weakly connected, and for every edge $e \in E$ either $\iota(e), \tau(e) \in V_H$ or $\iota(e), \tau(e) \in V \setminus V_H$.

The isomorphism between two labelled graphs is defined in the usual way. Two labelled graphs $G = (V, E, f, \Psi)$ and $G' = (V', E', f', \Psi)$ are *isomorphic*, denoted by $G \approx G'$, if there is a bijection $\alpha : V \rightarrow V'$ such that $f(v) = f'(\alpha(v))$ for all $v \in V$, and

$$(x, u, y) \in E \text{ iff } (\alpha(x), u, \alpha(y)) \in E',$$

for all $x, y \in V$ and $u \in \Psi^*$. The bijection α is then called an *isomorphism from G to G'* .

In this chapter we will consider walks in labelled graphs that often originate in a fixed source vertex and will end in a fixed target vertex. Therefore, we need the following notion.

A *two-ended graph* is a 6-tuple $G = (V, E, f, \Psi, s, t)$, where (V, E, f, Ψ) is a labelled graph, f is a function on $V \setminus \{s, t\}$ and $s, t \in V$ where $s \neq t$. Vertex s is called the *source vertex* of G and vertex t is called the *target vertex* of G . The

basic notions and notation for labelled graphs carry over to two-ended graphs. However, for the notion of isomorphism, care must be taken that the two ends are preserved. Thus, if G and G' are two-ended graphs, and α is an isomorphism from G to G' , then $\alpha(s) = s'$ and $\alpha(t) = t'$, where s (s' , resp.) is the source vertex of G (G' , resp.) and t (t' , resp.) is the target vertex of G (G' , resp.).

2.4 The String Pointer Reduction System

In this chapter we consider the string pointer reduction system, which we will recall now (see also [11] and Chapter 9 in [12]).

We fix $\kappa \geq 2$, and define the alphabet $\Delta = \{2, 3, \dots, \kappa\}$. For $D \subseteq \Delta$, we define $\bar{D} = \{\bar{a} \mid a \in D\}$ and $\Pi_D = D \cup \bar{D}$; also $\Pi = \Pi_\Delta$. We will use the alphabet Π to formally denote the pointers — the intuition is that the pointer p_i will be denoted by either i or $\bar{i}$. Accordingly, elements of Π will also be called *pointers*.

We use the ‘bar operator’ to move from Δ to $\bar{\Delta}$ and back from $\bar{\Delta}$ to Δ . Hence, for $p \in \Pi$, $\bar{\bar{p}} = p$. For a string $u = x_1 x_2 \cdots x_n$ with $x_i \in \Pi$, the *inverse* of u is

the string $\bar{u} = \bar{x}_n \bar{x}_{n-1} \cdots \bar{x}_1$. For $p \in \Pi$, we define $\mathbf{p} = \begin{cases} p & \text{if } p \in \Delta \\ \bar{p} & \text{if } p \in \bar{\Delta} \end{cases}$, i.e., $\mathbf{p}$ is

the ‘unbarred’ variant of p . The *domain* of a string $v \in \Pi^*$ is $\text{dom}(v) = \{\mathbf{p} \mid p \text{ occurs in } v\}$. A *legal string* is a string $u \in \Pi^*$ such that for each $p \in \Pi$ that occurs in u , u contains exactly two occurrences from $\{p, \bar{p}\}$.

We define the alphabet $\Theta_\kappa = \{M_i, \bar{M}_i \mid 1 \leq i \leq \kappa\}$ — these symbols denote the MDSs and their inversions. With each string over Θ_κ , we associate a unique string over Π through the homomorphism $\pi_\kappa : \Theta_\kappa^* \rightarrow \Pi^*$ defined by:

$$\pi_\kappa(M_1) = 2, \quad \pi_\kappa(M_\kappa) = \kappa, \quad \pi_\kappa(M_i) = i(i+1) \quad \text{for } 1 < i < \kappa,$$

and $\pi_\kappa(\bar{M}_j) = \overline{\pi_\kappa(M_j)}$ for $1 \leq j \leq \kappa$. A permutation of the string $M_1 M_2 \cdots M_\kappa$, with possibly some of its elements inverted, is called a *micronuclear pattern* since it can describe the MIC form of a gene. String u is *realistic* if there is a micronuclear pattern δ such that $u = \pi_\kappa(\delta)$.

Example 1

The MIC form of the gene that encodes the actin protein in the stichotrich *Sterkiella nova* is described by micronuclear pattern

$$\delta = M_3 M_4 M_6 M_5 M_7 M_9 \bar{M}_2 M_1 M_8$$

(see [22, 12]). The associated realistic string is $\pi_9(\delta) = 34456756789\bar{3}\bar{2}289$.

Note that every realistic string is legal, but a legal string need not be realistic. For example, a realistic string cannot have ‘gaps’ (missing pointers): thus 2244 is not realistic while it is legal. It is also easy to produce examples of legal strings which do not have gaps but still are not realistic — 3322 is such an example. For a pointer p and a legal string u , if both p and $\bar{p}$ occur in u then we say that both p

and $\bar{p}$ are *positive* in u ; if on the other hand only p or only $\bar{p}$ occurs in u , then both p and $\bar{p}$ are *negative* in u . So, every pointer occurring in a legal string is either positive or negative in it. A nonempty legal string with no proper nonempty legal substrings is called *elementary*. For example, the legal string 234324 is elementary, while the legal string 234342 is not (because 3434 is a proper legal substring).

Definition 1

Let $u = x_1x_2 \cdots x_n$ be a legal string with $x_i \in \Pi$ for $1 \leq i \leq n$. For a pointer $p \in \Pi$ such that $\{x_i, x_j\} \subseteq \{p, \bar{p}\}$ and $1 \leq i < j \leq n$, the p -interval of u is the substring $x_i x_{i+1} \cdots x_j$. Two distinct pointers $p, q \in \Pi$ *overlap* in u if the p -interval of u overlaps with the q -interval of u . ■

The string pointer reduction system consists of three types of reduction rules operating on legal strings. For all $p, q \in \Pi$ with $\mathbf{p} \neq \mathbf{q}$, we define:

- the *string negative rule* for p by $\mathbf{snr}_p(u_1 p p u_2) = u_1 u_2$,
- the *string positive rule* for p by $\mathbf{spr}_p(u_1 p u_2 \bar{p} u_3) = u_1 \bar{u}_2 u_3$,
- the *string double rule* for p, q by $\mathbf{sdr}_{p,q}(u_1 p u_2 q u_3 p u_4 q u_5) = u_1 u_4 u_3 u_2 u_5$,

where $u_1, u_2, \dots, u_5$ are arbitrary strings over Π .

Note that each of these rules is defined only on legal strings that satisfy the given form. For example, $\mathbf{snr}_2$ is not defined on legal string 2323. It is important to realize that for every non-empty legal string there is at least one reduction rule applicable. Indeed, every legal string for which no string positive rule and no string double rule is applicable must have only nonoverlapping, negative pointers and thus a string negative rule is applicable.

We also define $\mathbf{Snr} = \{\mathbf{snr}_p \mid p \in \Pi\}$, $\mathbf{Spr} = \{\mathbf{spr}_p \mid p \in \Pi\}$ and $\mathbf{Sdr} = \{\mathbf{sdr}_{p,q} \mid p, q \in \Pi, \mathbf{p} \neq \mathbf{q}\}$ to be the sets containing all the reduction rules of a specific type.

The string negative rule corresponds to the loop recombination operation, the string positive rule corresponds to the hairpin recombination operation, and the string double rule corresponds to the double-loop recombination operation. Note that the fact (pointed out at the end of Section 2.2) that the molecular operations remove pointers is explicit in the string pointer reduction system — indeed when a string rule for a pointer p (or pointers p and q) is applied, then all occurrences of p and $\bar{p}$ (or $p, \bar{p}, q$ and $\bar{q}$) are removed.

Definition 2

The *domain* $\text{dom}(\rho)$ of a reduction rule ρ equals the set of unbarred variants of the pointers the rule is applied to, i.e., $\text{dom}(\mathbf{snr}_p) = \text{dom}(\mathbf{spr}_p) = \{\mathbf{p}\}$ and $\text{dom}(\mathbf{sdr}_{p,q}) = \{\mathbf{p}, \mathbf{q}\}$ for $p, q \in \Pi$. For a composition $\varphi = \varphi_1 \varphi_2 \cdots \varphi_n$ of reduction rules $\varphi_1, \varphi_2, \dots, \varphi_n$, the *domain* $\text{dom}(\varphi)$ is the union of the domains of its constituents, i.e., $\text{dom}(\varphi) = \text{dom}(\varphi_1) \cup \text{dom}(\varphi_2) \cup \cdots \cup \text{dom}(\varphi_n)$. ■

Definition 3

Let u and v be legal strings and $S \subseteq \{Snr, Spr, Sdr\}$. Then a composition φ of reduction rules from S is called an $(S\text{-})$ reduction of u , if φ is applicable to (defined on) u . A *successful reduction* φ of u is a reduction of u such that $\varphi(u) = \lambda$. We then also say that φ is *successful for* u . We say that u is *reducible to* v in S if there is a S -reduction φ of u such that $\varphi(u) = v$. We simply say that u is *reducible to* v if u is reducible to v in $\{Snr, Spr, Sdr\}$. We say that u is *successful in* S if u is reducible to λ in S . ■

Note that if φ is a reduction of u , then $dom(\varphi) = dom(u) \setminus dom(\varphi(u))$. Because (as pointed out already) for every non-empty legal string there is at least one reduction rule applicable, we easily obtain Theorem 9.1 in [12] which states that every legal string is successful in $\{Snr, Spr, Sdr\}$.

Example 2

Let $S = \{Snr, Spr\}$, $u = 3245\bar{4}5\bar{3}2$, and $v = \bar{5}4\bar{5}\bar{4}$. Then u is reducible to v in S , because $(\mathbf{snr}_3 \mathbf{spr}_2)(u) = v$. Since applying $\varphi = \mathbf{spr}_5 \mathbf{spr}_4 \mathbf{snr}_2 \mathbf{spr}_3$ to u yields λ , φ is successful for u . On the other hand, $u = 3232$ is not reducible to any v in S , because none of the rules in Snr and none of the rules in Spr is applicable for this u .

Referring to the Introduction, in Theorem 11 we present a characterization of the intermediate strings that may be constructed during the transformation of a given gene from its micronuclear form to its macronuclear form. Formally, this is a characterization of reducibility, which allows one to determine for any given legal strings u and v and $S \subseteq \{Snr, Spr, Sdr\}$, whether or not u is reducible to v in S . This result can be seen as a generalization of the results from Chapter 13 in [12], which provide a characterization of successfulness for realistic strings, that is, for the case where u is realistic and $v = \lambda$.

2.5 Pointer Removal Operation

Let φ be a reduction of a legal string u . If we let u' be the legal string obtained from u by deleting all pointers from $\Pi_{dom(\varphi(u))}$, then it turns out that φ is also a reduction of u' . In fact, φ is a successful reduction of u' . This is formalized in Theorem 6, and thus it states a necessary condition for reducibility. In the following sections we will strengthen Theorem 6 to obtain a characterization of reducibility.

Definition 4

For a subset $D \subseteq \Delta$, the D -removal operation, denoted by rem_D , is defined by $rem_D = erase_{D \cup \bar{D}}$. We also refer to rem_D operations, for all $D \subseteq \Delta$, as *pointer removal operations*. ■

Example 3

Let $u = 3245\bar{4}5\bar{3}\bar{2}$ and $D = \{4, 5\}$. Then $rem_D(u) = 32\bar{3}\bar{2}$. Note that $2, 3 \notin D$. Note also that $\varphi = \mathbf{snr}_3 \mathbf{spr}_2$ is applicable to both u and $rem_D(u)$, but for $rem_D(u)$, φ is also successful.

The following easy to verify lemma formalizes the essence of the above example.

Lemma 5

Let u be a legal string and $D \subseteq dom(u)$. Let φ be a composition of reduction rules.

1. If φ is applicable to $rem_D(u)$ and φ does not contain string negative rules, then φ is applicable to u .
2. If φ is applicable to u and $dom(\varphi) \subseteq dom(u) \setminus D$, then φ is applicable to $rem_D(u)$.
3. If φ is applicable to both u and $rem_D(u)$, then $\varphi(rem_D(u)) = rem_D(\varphi(u))$.

Note that the first statement of Lemma 5 may not be true when φ is allowed to contain string negative rules. The obvious reason for this is that two identical occurrences of a pointer p may end up to be next to each other only if some pointers in between those occurrences are first removed by rem_D . This is illustrated in the following example.

Example 4

Let $u = 3245\bar{4}5\bar{3}66\bar{2}$, $v = 5\bar{4}\bar{5}466$ and $D = dom(v)$. Then $rem_D(u) = 32\bar{3}\bar{2}$. Note that although $\varphi = \mathbf{snr}_3 \mathbf{spr}_2$ is a successful reduction of $rem_D(u)$, φ is not applicable to u .

The following theorem is an immediate consequence of the previous lemma.

Theorem 6

Let $S \subseteq \{Snr, Spr, Sdr\}$. For legal strings u and v , if u is reducible to v in S and $D = dom(v)$, then $rem_D(u)$ is successful in S .

Proof

Let u be reducible to v in S . Then there is an S -reduction φ such that $\varphi(u) = v$. By Lemma 5, φ is an S -reduction of $rem_D(u)$ and $\varphi(rem_D(u)) = rem_D(\varphi(u)) = rem_D(v) = \lambda$. Hence, φ is a successful S -reduction of $rem_D(u)$. ■

The proof of the above result observes that any reduction of u into v must be a successful reduction of $rem_D(u)$ where $D = dom(v)$. Referring to Example 4, we now note that u is not reducible to v , because $rem_D(u)$ has two successful reductions and neither is applicable to u . In fact, there is no v' with $D = dom(v')$ such that u is reducible to v' .



Figure 2.6: Part of a genome with three pointer pairs corresponding to the same gene.

2.6 Reduction Graphs

The main purpose of this section is to define the notion of reduction graph. A reduction graph represents some key aspects of reductions from a legal string u to a legal string v : it provides the additional requirements on u and v to make the reverse implication of Theorem 6 hold. In addition, it allows one to easily determine the number of string negative rules needed to successfully reduce u . We will first define the notion of a 2-edge coloured graph.

Definition 7

A *2-edge coloured graph* is a 7-tuple

$$G = (V, E_1, E_2, f, \Psi, s, t),$$

where both (V, E_1, f, Ψ, s, t) and (V, E_2, f, Ψ, s, t) are two-ended graphs. Note that E_1 and E_2 are not necessary disjoint. ■

The terminology and notation for the two-ended graph carries over to 2-edge coloured graphs. However, for the notion of isomorphism, care must be taken that the two sorts of edges are preserved. Thus, if $G = (V, E_1, E_2, f, \Psi, s, t)$ and $G' = (V', E'_1, E'_2, f', \Psi, s', t')$ are two-ended graphs, then it must hold that for any isomorphism α from G to G' ,

$$(x, u, y) \in E_i \text{ iff } (\alpha(x), u, \alpha(y)) \in E'_i$$

for all $x, y \in V$, $u \in \Psi$ and $i \in \{1, 2\}$.

We say that edges e_1 and e_2 have the *same colour* if either $e_1, e_2 \in E_1$ or $e_1, e_2 \in E_2$, otherwise they have *different colours*. An *alternating walk* in G is a walk $\pi = e_1 e_2 \cdots e_n$ in G such that e_i and e_{i+1} have different colours for $1 \leq i < n$. For each edge e with $\ell(e) \in \Pi^*$, we define $(\tau(e), \overline{\ell(e)}, \iota(e))$, denoted by $\bar{e}$, as the *reverse* of e .

We are ready now to define the notion of a reduction graph, the main technical notion of this chapter. The reduction graph is a 2-edge coloured graph and it is defined for a legal string u and a set of pointers $D \subseteq \text{dom}(u)$. The intuition behind it is as follows.

Figure 2.6 depicts a part of a genome with three pointer pairs corresponding to the same gene g . The reduction graph introduces two vertices for each pointer and two special vertices s and t representing the ends. It connects adjacent pointers through *reality edges* and connects pointers corresponding to the same pointer

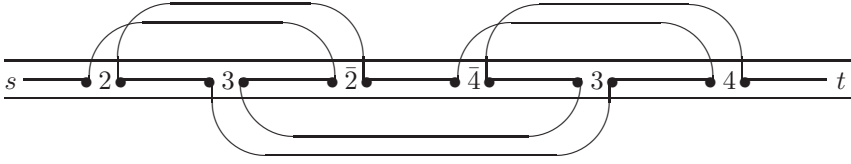


Figure 2.7: The reduction graph corresponding to the underlying genome.

pair through *desire edges* in a way that reflects how the parts will be glued after a molecular operation is applied on that pointer. The resulting reduction graph is depicted in Figure 2.7. Thus, every reality edge corresponds to a certain DNA segment. If such a DNA segment contains other pointers of g , then these pointers form the label of that reality edge.

By definition a realistic string has a physical interpretation. It shows the boundaries of the MDSs, and how these should be recombined (following their orientation). Considering a subset of these pointers, we still have the physical interpretation, although the other pointers are hidden in the segments. Technically, however, removing a subset of the pointers may change a realistic string into a legal one that is no longer realistic or even realizable (by renaming pointers we cannot obtain a realistic string). An example of such a case is given in the introduction of Section 2.10. In fact, each legal string has a physical interpretation with pointers indicating how parts of the string are to be reconnected, cf. Figure 2.7, where no use is made of any MDS-IES segmentation. Thus our definition of reduction graph works for legal strings in general, rather than only for realistic ones. The intuition of a reduction graph is similar to the intuition behind a reality and desire diagram (or breakpoint graph) from [16, 21].

Formally, the reduction graph of legal string u with respect to $D \subseteq \text{dom}(u)$ shows how u is reduced to a legal string v with $\text{dom}(v) = D$ by any possible reduction φ . The vertices of the graph correspond to (two copies of each of) the pointers that are removed during the reduction (those in $\Pi_{\text{dom}(u) \setminus D}$). As illustrated above, we have two types of edges. The desire edges are unlabelled and connect the pointer pairs in $\Pi_{\text{dom}(u) \setminus D}$, while reality edges connect the successive pointers in $\Pi_{\text{dom}(u) \setminus D}$ and are labelled by the strings over Π_D^* that are in between these pointers in u .

Definition 8

Let $D \subseteq \Delta$ and let u be a legal string, such that $u = \delta_0 p_1 \delta_1 p_2 \dots p_n \delta_n$ where $\delta_0, \dots, \delta_n \in \Pi_D^*$ and $p_1, \dots, p_n \in \Pi_{\text{dom}(u) \setminus D}$. The *reduction graph of u with respect to D* , denoted by $\mathcal{R}_{u,D}$, is a 2-edge coloured graph $(V, E_1, E_2, f, \Pi, s, t)$, where

$$V = \{I_1, I_2, \dots, I_n\} \cup \{I'_1, I'_2, \dots, I'_n\} \cup \{s, t\},$$

$$E_1 = E_{1,r} \cup E_{1,l}, \text{ where}$$

$$E_{1,r} = \{e_0, e_1, \dots, e_n\} \text{ with } e_i = (I'_i, \delta_i, I_{i+1}) \text{ for } 1 \leq i \leq n-1,$$

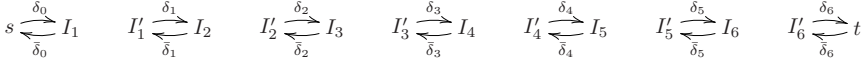


Figure 2.8: The part of the reduction graph of the legal string u with respect to D as defined in Example 5 which involves only reality edges (the vertex labels are omitted).

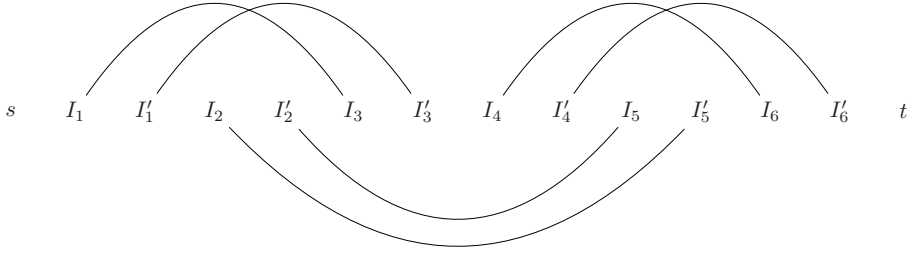


Figure 2.9: The part of the reduction graph of the legal string u with respect to D as defined in Example 5, where only desire edges are shown (the vertex labels are omitted). Crossing edges correspond to positive pointers.

$$e_0 = (s, I_1), e_n = (I'_n, t),$$

$$E_{1,l} = \{\bar{e} \mid e \in E_{1,r}\},$$

$$E_2 = \{(I'_i, \lambda, I_j), (I_i, \lambda, I'_j) \mid i, j \in \{1, 2, \dots, n\} \text{ with } i \neq j \text{ and } p_i = p_j\} \cup \\ \{(I_i, \lambda, I_j), (I'_i, \lambda, I'_j) \mid i, j \in \{1, 2, \dots, n\} \text{ and } p_i = \bar{p}_j\}, \text{ and}$$

$$f(I_i) = f(I'_i) = \mathbf{p}_i \text{ for } 1 \leq i \leq n.$$

■

The edges of E_1 are called the *reality edges*, and the edges of E_2 are called the *desire edges*. Note that E_1 and E_2 are not necessary disjoint. The components of $\mathcal{R}_{u,D}$ that do not contain s and t are called *cyclic components*. When $D = \emptyset$, we simply refer to $\mathcal{R}_{u,D}$ as the *reduction graph of u* .

Thus the reduction graph is a ‘superposition’ of two graphs on the same set of vertices V : one graph with edges from E_1 (reality edges), and one graph with edges from E_2 (desire edges). The following example should make the notion of reduction graph more clear.

Example 5

Let $u = 526883\bar{2}5\bar{4}37746$ be a legal string and $D = \{5, 6, 7, 8\} \subseteq \text{dom}(u)$. Thus, $\{2, 3, 4\} = \text{dom}(u) \setminus D$, and

$$u = \delta_0 \ 2 \ \delta_1 \ 3 \ \delta_2 \ \bar{2} \ \delta_3 \ \bar{4} \ \delta_4 \ 3 \ \delta_5 \ 4 \ \delta_6$$

with $\delta_0 = 5$, $\delta_1 = 688$, $\delta_2 = \lambda$, $\delta_3 = 5$, $\delta_4 = \lambda$, $\delta_5 = 77$ and $\delta_6 = 6$. Notice that $\delta_1, \delta_2, \dots, \delta_6 \in \Pi_D^*$. This example corresponds to the situation in Figure 2.6.

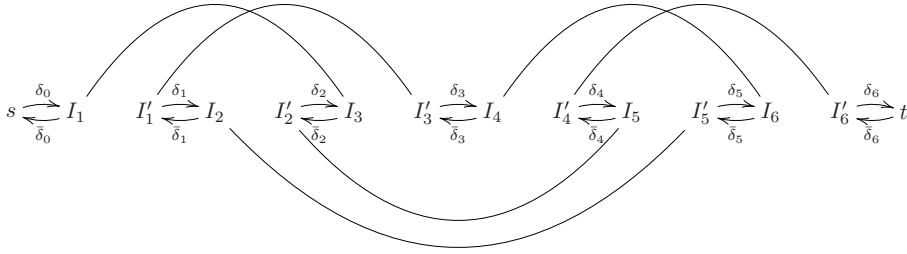


Figure 2.10: The reduction graph $\mathcal{R}_{u,D}$ as defined in Example 5 (the vertex labels are omitted).

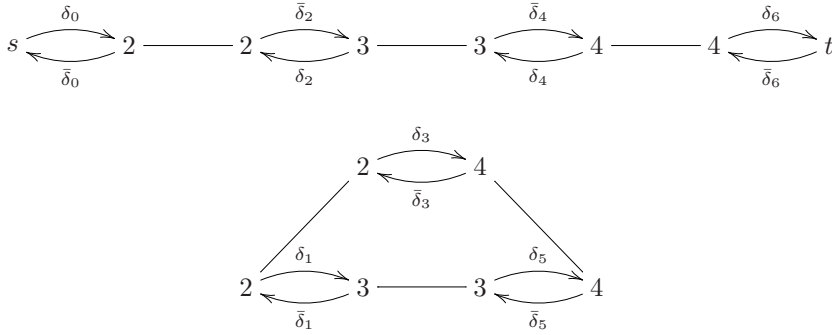


Figure 2.11: The reduction graph of Figure 2.10 where every vertex (except s and t) is represented by its label.

The reduction graph $\mathcal{R}_{u,D}$ of u with respect to D is given in Figure 2.10. It is the union of the graphs in Figure 2.8 and Figure 2.9. Note that for every desire edge e , we represent both e and $\bar{e}$ by a single unlabelled, undirected edge. The graphs are drawn in a form that closely relates to the linear ordering of u . The desire edges that cross correspond to positive pointers, and the desire edges that do not cross correspond to negative pointers.

Since the exact identity of the vertices in a reduction graph is not essential for the problems considered in this chapter (we need only to know, modulo ‘bar’, which pointer is represented by a given vertex), in order to simplify the pictorial notation of reduction graphs we will replace the vertices (except for s and t) by their labels. Figure 2.11 gives $\mathcal{R}_{u,D}$ in this way. In this figure we have reordered the vertices, making it transparent that $\mathcal{R}_{u,D}$ has a single cyclic component (the figure illustrates why the adjective ‘cyclic’ was added).

Note that a reduction graph is an undirected graph in the sense that if $e \in E_1$ ($e \in E_2$, resp.) then also $\bar{e} \in E_1$ ($\bar{e} \in E_2$, resp.). If we think of a reduction graph as an undirected graph by considering edges e and $\bar{e}$ as one undirected edge, then both s and t are connected to exactly one (undirected) edge, and every other vertex is connected to exactly two (undirected) edges. As a corollary to Euler’s theorem, a reduction graph has exactly one component that has a linear structure with s and t as endpoints and possibly one or more components that have a cyclic structure (the cyclic components). Thus, there is a unique alternating walk from s to t in every reduction graph.

If a 2-edge coloured graph G has a unique alternating walk from s to t , then the label of this walk is called the *reduct of G* , denoted by $\text{red}(G)$. We know now that if $\mathcal{R}_{u,D}$ is a reduction graph of a legal string u with respect to $D \subseteq \text{dom}(u)$, then the reduct exists. It is then also called the *reduct of u to D* , and denoted by $\text{red}(u, D)$. Since $\mathcal{R}_{u, \text{dom}(u)}$ consists of the vertices s and t connected by a (reality) edge labelled by u (and by $\bar{u}$ in the reverse direction), we have $\text{red}(u, \text{dom}(u)) = u$. Also, it is clear that if 2-edge coloured graphs G_1 and G_2 are isomorphic, then $\text{red}(G_1) = \text{red}(G_2)$.

Example 6

If we take u and D from Example 5, then

$$\text{red}(u, D) = \delta_0 \bar{\delta}_2 \bar{\delta}_4 \delta_6 = 56,$$

which is easy to see in Figure 2.11.

2.7 Reduction Function

Before we can prove (in the next section) our main theorem on reducibility, we need to define reduction functions. A reduction function operates on reduction graphs. As we will see, these functions simulate the effect (up to isomorphism) of each of the three string pointer reduction rules on a reduction graph. For a vertex

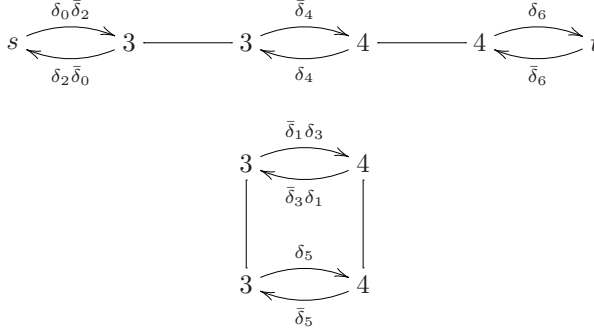


Figure 2.12: The reduction graph obtained when applying rf_2 to the reduction graph of Figure 2.11.

label p , the p -reduction function merges edges that form a walk ‘over’ vertices labelled by p and removes all vertices labelled by p .

Definition 9

For each vertex label p , we define the p -reduction function rf_p , which constructs for every 2-edge coloured graph $G = (V, E_1, E_2, f, \Psi, s, t)$, the 2-edge coloured graph

$$rf_p(G) = (V', (E_1 \setminus E_{rem}) \cup E_{add}, E_2 \setminus E_{rem}, f|V', \Psi, s, t),$$

with

$$\begin{aligned} V' &= \{s, t\} \cup \{v \in V \setminus \{s, t\} \mid f(v) \neq p\}, \\ E_{rem} &= \{e \in E_1 \cup E_2 \mid f(\iota(e)) = p \text{ or } f(\tau(e)) = p\}, \text{ and} \\ E_{add} &= \{(\iota(\pi), \ell(\pi), \tau(\pi)) \mid \pi = e_1 e_2 \cdots e_n \text{ with } n > 2 \text{ is an alternating walk} \\ &\quad \text{in } G \text{ with } f(\iota(\pi)) \neq p, f(\tau(\pi)) \neq p, \text{ and } f(\tau(e_i)) = p \text{ for } 1 \leq i < n\}. \end{aligned}$$

■

Example 7

If we take the reduction graph $\mathcal{R}_{u,D}$ from Example 5, cf. Figure 2.11, then $rf_2(\mathcal{R}_{u,D})$ is given in Figure 2.12.

It is easy to see that the following property holds for each reduction graph $\mathcal{R}_{u,D}$ and all $p \in \text{dom}(u) \setminus D$:

$$\text{red}(\mathcal{R}_{u,D}) = \text{red}(rf_p(\mathcal{R}_{u,D})).$$

Also, reduction functions commute under composition. Thus, if moreover there is a $q \in \text{dom}(u) \setminus D$ such that $p \neq q$, then

$$(rf_q rf_p)(\mathcal{R}_{u,D}) = (rf_p rf_q)(\mathcal{R}_{u,D}).$$

The main property of reduction functions is that they simulate the effect (up to isomorphism) of each of the three string pointer reduction rules on a reduction graph.

Theorem 10

Let u be a legal string, let $D \subseteq \text{dom}(u)$, and let φ be a reduction of u such that $\text{dom}(\varphi) = \{p_1, p_2, \dots, p_n\} \subseteq \text{dom}(u) \setminus D$. Then

$$(rf_{p_n} \cdots rf_{p_2} rf_{p_1})(\mathcal{R}_{u,D}) \approx \mathcal{R}_{\varphi(u),D},$$

and $\text{red}(u, D) = \text{red}(\varphi(u), D)$.

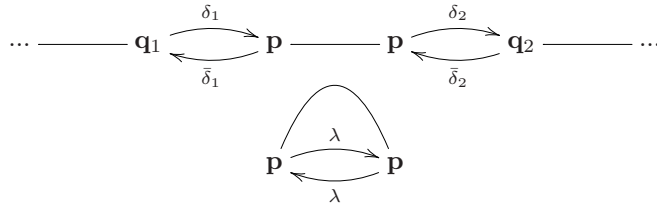
Proof

To prove the first statement, it suffices to prove the cases where $\varphi = \mathbf{snr}_p$, $\varphi = \mathbf{spr}_p$ and $\varphi = \mathbf{sdr}_{p,q}$ for $p, q \in \Pi_{\text{dom}(u) \setminus D}$.

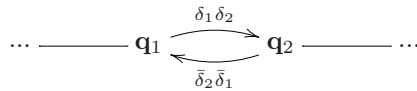
We first prove the **snr** case. Assume **snr** _{p} is applicable to u . We consider the general case

$$u = u_1 q_1 \delta_1 p p \delta_2 q_2 u_2$$

for some $\delta_1, \delta_2 \in \Pi_D^*$, $q_1, q_2 \in \Pi_{\text{dom}(u) \setminus D}$ and $u_1, u_2 \in \Pi^*$. In the special case where q_1 (q_2 , resp.) does not exist, the vertex labelled by $\mathbf{q}_1$ ($\mathbf{q}_2$, resp.) in the graphs below equals the source vertex s (target vertex t , resp.). We will first prove that $rf_{\mathbf{p}}(\mathcal{R}_{u,D}) = \mathcal{R}_{\mathbf{snr}_p(u),D}$. Because $u = u_1 q_1 \delta_1 p p \delta_2 q_2 u_2$, the reduction graph $\mathcal{R}_{u,D}$ is



where we omitted the parts of the graph that remain the same after applying $rf_{\mathbf{p}}$. Now, the graph $rf_{\mathbf{p}}(\mathcal{R}_{u,D})$ is given below.



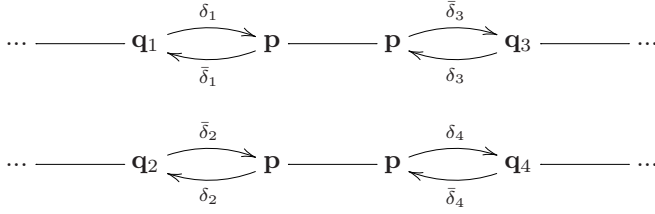
This is clearly the reduction graph of $\mathbf{snr}_p(u) = u_1 q_1 \delta_1 \delta_2 q_2 u_2$ with respect to D . Thus, indeed $rf_{\mathbf{p}}(\mathcal{R}_{u,D}) \approx \mathcal{R}_{\mathbf{snr}_p(u),D}$.

We now prove the **spr** case. Assume **spr** _{p} is applicable to u . We may distinguish three cases, which differ in the number of elements of $\Pi_{\text{dom}(u) \setminus D}$ in between p and $\bar{p}$ in u :

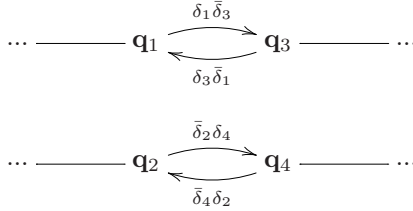
1. $u = u_1 q_1 \delta_1 p \delta_2 \bar{p} \delta_4 q_4 u_3$
2. $u = u_1 q_1 \delta_1 p \delta_2 q_2 \delta_3 \bar{p} \delta_4 q_4 u_3$

$$3. \ u = u_1 q_1 \delta_1 p \delta_2 q_2 u_2 q_3 \delta_3 \bar{p} \delta_4 q_4 u_3$$

for some $\delta_1, \dots, \delta_4 \in \Pi_D^*$, $q_1, \dots, q_4 \in \Pi_{\text{dom}(u) \setminus D}$, and $u_1, u_2, u_3 \in \Pi^*$. Note that we have assumed that p is preceded and that $\bar{p}$ is followed by an element from $\Pi_{\text{dom}(u) \setminus D}$. The special cases where q_1 or q_4 do not exist, can be handled in the same way as we did for the **snr** case (by setting them equal to s and t , resp.). In each of the three cases, one can prove that $rf_{\mathbf{p}}(\mathcal{R}_{u,D}) \approx \mathcal{R}_{\mathbf{spr}_p(u),D}$. We will discuss it in detail only for the third case. The reduction graph $\mathcal{R}_{u,D}$ is



where we again omitted the parts of the graph that remain the same after applying $rf_{\mathbf{p}}$. Now, the graph $rf_{\mathbf{p}}(\mathcal{R}_{u,D})$ is given below.



This graph is clearly isomorphic to the reduction graph of

$$\mathbf{spr}_p(u) = u_1 q_1 \delta_1 \bar{\delta}_3 \bar{q}_3 \bar{u}_2 \bar{q}_2 \bar{\delta}_2 \delta_4 q_4 u_3$$

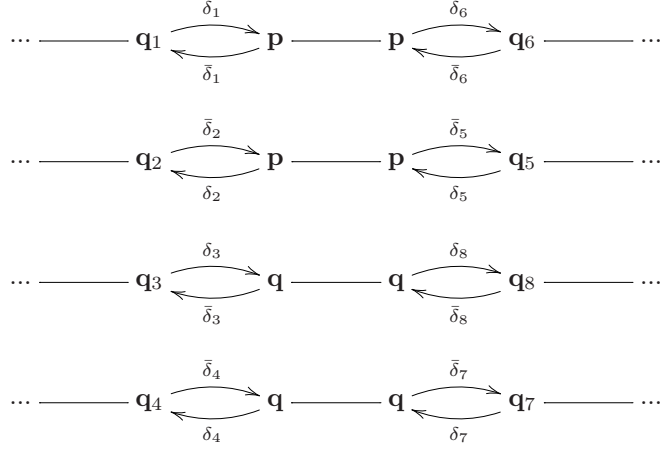
with respect to D . Thus, indeed $rf_{\mathbf{p}}(\mathcal{R}_{u,D}) \approx \mathcal{R}_{\mathbf{spr}_p(u),D}$.

Finally, we prove the **sdr** case. Assume **sdr** _{p,q} is applicable to u . We only consider the general case (the other cases are proved similarly):

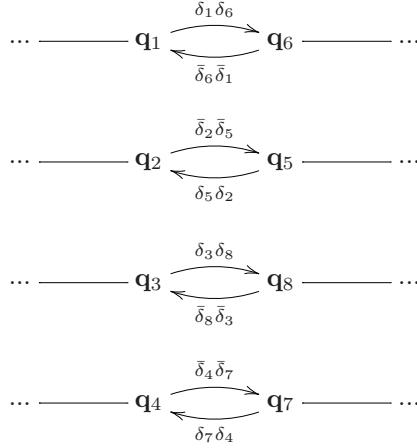
$$u = u_1 \ q_1 \delta_1 p \delta_2 q_2 \ u_2 \ q_3 \delta_3 q \delta_4 q_4 \ u_3 \ q_5 \delta_5 p \delta_6 q_6 \ u_4 \ q_7 \delta_7 q \delta_8 q_8 \ u_5$$

for some $\delta_1, \dots, \delta_8 \in \Pi_D^*$, $q_1, \dots, q_8 \in \Pi_{\text{dom}(u) \setminus D}$, and $u_1, \dots, u_5 \in \Pi^*$. The

reduction graph $\mathcal{R}_{u,D}$ is



where we omitted the parts of the graph that remain the same after applying $(rf_{\mathbf{q}} rf_{\mathbf{p}})$. Now, the graph $rf_{\mathbf{q}}(rf_{\mathbf{p}}(\mathcal{R}_{u,D}))$ is given below.



This graph is clearly isomorphic to the reduction graph of

$$\mathbf{sdr}_{p,q}(u) = u_1 q_1 \delta_1 \delta_6 q_6 u_4 q_7 \delta_7 \delta_4 q_4 u_3 q_5 \delta_5 \delta_2 q_2 u_2 q_3 \delta_3 \delta_8 q_8 u_5$$

with respect to D . Thus, indeed $rf_{\mathbf{q}}(rf_{\mathbf{p}}(\mathcal{R}_{u,D})) \approx \mathcal{R}_{\mathbf{sdr}_{p,q}(u),D}$. This proves the first statement.

Now, by the fact that the reduction function does not change the reduct of the graph, and by the first statement, we have

$$red(\mathcal{R}_{u,D}) = red((rf_{p_1} rf_{p_2} \cdots rf_{p_n})(\mathcal{R}_{u,D})) = red(\mathcal{R}_{\varphi(u),D}).$$

Thus, $red(u,D) = red(\varphi(u),D)$ and this proves the second statement. $\blacksquare$

2.8 Characterization of Reducibility

We are now ready to prove our main theorem on reducibility. In Theorem 6 we have shown that if u is reducible to v in S , then $rem_{dom(v)}(u)$ is successful in S . Here we strengthen this theorem into an iff statement by additionally requiring that v equals the reduct of u to $dom(v)$. The resulting characterization is independent of the chosen set of reduction rules $S \subseteq \{Snr, Spr, Sdr\}$.

Theorem 11

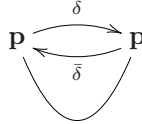
Let u and v be legal strings, $D = dom(v) \subseteq dom(u)$ and $S \subseteq \{Snr, Spr, Sdr\}$. Then u is reducible to v in S iff $rem_D(u)$ is successful in S and $red(u, D) = v$.

Proof

Let u be reducible to v in S . Therefore, there is an S -reduction φ of u such that $\varphi(u) = v$. Also, $rem_D(u)$ is successful in S by Theorem 6. By Theorem 10, we have $red(u, D) = red(\varphi(u), D)$. Now, $red(\varphi(u), D) = \varphi(u) = v$, because $D = dom(\varphi(u))$.

To prove the reverse implication, let $rem_D(u)$ be successful in S and $red(u, D) = v$. We have to prove that u is reducible to v in S . Clearly, there is a successful S -reduction φ of $rem_D(u)$.

Assume that φ is not applicable to u . Since φ is applicable to $rem_D(u)$, we know from Lemma 5 that $\varphi = \varphi_2 \mathbf{snr}_p \varphi_1$ for some φ_1 , φ_2 and p , where φ_1 is applicable to u and $\mathbf{snr}_p$ is not applicable to $\varphi_1(u)$. Thus, $p\delta p$ is a substring of $\varphi_1(u)$ with $\delta \in \Pi_D^* \setminus \{\lambda\}$. Therefore the following graph



must be isomorphic to a cyclic component of the reduction graph $\mathcal{R}_{\varphi_1(u), D}$ of $\varphi_1(u)$ with respect to D . Because $v = red(u, D) = red(\varphi_1(u), D)$ is a legal string and $dom(v) = D$, the labels of the reality edges of $\mathcal{R}_{\varphi_1(u), D}$ belonging to cyclic components are empty. This is a contradiction and therefore φ is applicable to u . Now, we have $\varphi(u) = red(\varphi(u), D) = red(u, D) = v$, because $D = dom(\varphi(u))$. Thus, u is reducible to v in S . ■

Note that the proof of Theorem 11 even proves a stronger fact. The S -reduction φ of u with $\varphi(u) = v$ can be taken to be same as the (successful) S -reduction φ of $rem_D(u)$. The following corollary follows directly from the previous theorem and the fact that every legal string is successful in $\{Snr, Spr, Sdr\}$.

Corollary 12

Let u and v be legal strings and $D = dom(v) \subseteq dom(u)$. Then u is reducible to v iff $red(u, D) = v$.

The previous corollary shows that reducibility can be checked quite efficiently. Since the reduction graph of a legal string u has $2|u| + 2$ vertices and $8|u| + 4$ edges (counting an undirected desire edge as two (directed) edges), it takes only linear time $O(|u|)$ to generate $\mathcal{R}_{u,\emptyset}$ using the adjacency lists representation. Also, generating $\mathcal{R}_{u,D}$ for any $D \subseteq \text{dom}(u)$ is of at most the same complexity as $\mathcal{R}_{u,\emptyset}$. Now, since the walk from s to t does not contain vertices more than once, it takes only linear time to determine $\text{red}(u, D) = v$, and therefore, by the previous corollary, it takes linear time to determine whether or not u is reducible to v .

The next corollary illustrates that the function of the reduct is twofold: it does not only determine, given u and $D \subseteq \text{dom}(u)$, *which* legal string is obtained by applying a reduction φ of u with $\text{dom}(\varphi(u)) = D$, but also *whether or not* there is such a φ .

Corollary 13

Let u be a legal string and $D \subseteq \text{dom}(u)$. Then u there is a reduction φ of u with $\text{dom}(\varphi(u)) = D$ iff $\text{red}(u, D)$ is legal and $\text{dom}(\text{red}(u, D)) = D$.

Proof

We first prove the forward implication. If we let $v = \varphi(u)$, then v is a legal string, u is reducible to v , and $D = \text{dom}(v)$. By Corollary 12, $\text{red}(u, D) = v$ and therefore $\text{red}(u, D)$ is legal and $\text{dom}(\text{red}(u, D)) = D$.

We now prove the reverse implication. If we let $v = \text{red}(u, D)$, then v is legal and $\text{dom}(v) = D$. By Corollary 12, u is reducible to v . ■

Example 8

Let u and D be as in Example 5. By Example 6, $\text{red}(u, D) = 56$. Therefore by Corollary 13, there is no reduction φ of u with $\text{dom}(\varphi(u)) = D$. Thus, there is no reduction φ of u with $\text{dom}(\varphi) = \{2, 3, 4\}$.

2.9 Cyclic Components

In this section we consider the cyclic components of the ‘full’ reduction graph $\mathcal{R}_{u,\emptyset}$ of a legal string u . We show that if $\mathbf{snr}_p$ is applicable to u for some pointer p , then the number of cyclic components of $\mathcal{R}_{\mathbf{snr}_p(u),\emptyset}$ is exactly one less than the number of cyclic components of $\mathcal{R}_{u,\emptyset}$. On the other hand, if either $\mathbf{spr}_p$ or $\mathbf{sdr}_{p,q}$ is applicable to u for some pointer p, q , then the number of cyclic components remains the same. Before we state this result (Theorem 17), we will prepare for its proof by studying some elementary connections between u and the structures in $\mathcal{R}_{u,\emptyset}$. Since all the edges of $\mathcal{R}_{u,\emptyset}$ are labelled λ , we will omit the labels of the edges in the figures.

Because desire edges in a reduction graph connect vertices that are of the same label, for every label $\mathbf{p}$, there are exactly 0, 2 or 4 vertices labelled by $\mathbf{p}$ in every cyclic component of a reduction graph. The following lemma establishes an additional property of the number of vertices of a single label in a cyclic component.

Lemma 14

Let u be a legal string, and let P be a cyclic component in $\mathcal{R}_{u,\emptyset}$. Let p (q , resp.) be the first (last, resp.) pointer (from left to right) in u such that there is a vertex in P with label $\mathbf{p}$ ($\mathbf{q}$, resp.). Then there are exactly two vertices of P labelled by $\mathbf{p}$ and there are exactly two vertices of P labelled by $\mathbf{q}$.

Proof

Assume that all four vertices labelled by $\mathbf{p}$ are in P . Then these vertices are I_i , I'_i , I_j and I'_j for some i and j with $i < j$. By the definition of reduction graph, there is a reality edge from vertex I_i to vertex I'_{i-1} . But by the definition of p , vertex I'_{i-1} cannot belong to P , which is a contradiction. Therefore, there are only two vertices labelled by $\mathbf{p}$ in P . The second claim is proved analogously. ■

Note that in the previous lemma, $\mathbf{p}$ and $\mathbf{q}$ need not be distinct. Note also that if all the vertices of a cyclic component have the same label, then the cyclic component has exactly two vertices.

Lemma 15

Let u be a legal string, and let $p \in \Pi$. Then $\mathcal{R}_{u,\emptyset}$ has a cyclic component consisting of exactly two vertices, which are both labelled by $\mathbf{p}$ iff either pp or $\bar{p}\bar{p}$ is a substring of u .

Proof

Let either pp or $\bar{p}\bar{p}$ be a substring of u . Then



is a cyclic component of $\mathcal{R}_{u,\emptyset}$ consisting of exactly two vertices, both labelled by $\mathbf{p}$.

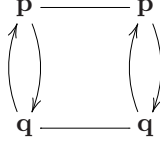
To prove the forward implication, let $\mathcal{R}_{u,\emptyset}$ have a cyclic component P consisting of exactly two vertices, both labelled by $\mathbf{p}$. Clearly, every vertex of a cyclic component has exactly one incoming and one outgoing edge in each colour. Because there is a reality edge between the two vertices of P , I'_i and I_{i+1} are the vertices of P for some i . Now, since there is a desire edge (I'_i, I_{i+1}) in P , either p or $\bar{p}$ occurs twice in u . As reality edges in $\mathcal{R}_{u,\emptyset}$ connect adjacent pointers in u , either pp or $\bar{p}\bar{p}$ is a substring of u . ■

Lemma 16

Let u be a legal string, let p and q be negative pointers occurring in u . Then $\mathcal{R}_{u,\emptyset}$ has a cyclic component consisting of exactly two vertices labelled by $\mathbf{p}$ and two vertices labelled by $\mathbf{q}$ iff either $u = u_1 p q u_2 q p u_3$ or $u = u_1 q p u_2 p q u_3$ for some strings $u_1, u_2, u_3 \in \Pi^*$.

Proof

Let either $u = u_1pqu_2qpu_3$ or $u = u_1qpu_2pqu_3$ for some strings $u_1, u_2, u_3 \in \Pi^*$. Then



is a cyclic component of $\mathcal{R}_{u,\emptyset}$ consisting of exactly two vertices labelled by **p** and two vertices labelled by **q**.

To prove the forward implication, let $\mathcal{R}_{u,\emptyset}$ have a cyclic component P consisting of exactly two vertices labelled by **p** and two vertices labelled by **q**. Since each cyclic component ‘is’ a cycle of edges of alternating colour, and since desire edges connect only vertices with the same label, the component looks like the figure above. Since reality edges in $\mathcal{R}_{u,\emptyset}$ connect adjacent pointers in u and since p and q are negative, either $u = u_1pqu_2qpu_3$ or $u = u_1pqu_2pqu_3$ with $u_i \in \Pi^*$ (with possibly p and q interchanged). Assume that $u = u_1pqu_2pqu_3$ (with possibly p and q interchanged). Then there must be vertices I'_i and I'_j labelled by **p** with a desire edge (I'_i, I'_j) in P . But this is impossible since p is negative. Consequently, $u = u_1pqu_2qpu_3$ (with possibly p and q interchanged). ■

The following theorem states that only the string negative rules can remove cyclic components. This is consistent with the fact that only loop recombination introduces a new (cyclic) molecule, cf. Figure 2.3. Clearly, by the definition of reduction function, a cyclic component is removed by simply removing its vertices and edges and not by merging with another component.

Theorem 17

Let u be a legal string, let N be the number of cyclic components of $\mathcal{R}_{u,\emptyset}$, and let $p \in \Pi$ with $\mathbf{p} \in \text{dom}(u)$.

- If $\mathbf{snr}_p$ is applicable to u , then the reduction graph of $\mathbf{snr}_p(u)$ has exactly $N - 1$ cyclic components.
- If $\mathbf{spr}_p$ is applicable to u , then the reduction graph of $\mathbf{spr}_p(u)$ has exactly N cyclic components.

Now let $q \in \Pi$ with $\mathbf{q} \in \text{dom}(u)$ and $\mathbf{p} \neq \mathbf{q}$.

- If $\mathbf{sdr}_{p,q}$ is applicable to u , then the reduction graph of $\mathbf{sdr}_{p,q}(u)$ has exactly N cyclic components.

Proof

First note that by the definition of reduction function and Theorem 10 the number of cyclic components cannot increase when applying reduction rules.

Let $\mathbf{snr}_p$ be applicable to u . By Lemma 15, $\mathcal{R}_{u,\emptyset}$ has a cyclic component consisting of exactly two vertices, which are both labelled by **p**. It follows then from

Theorem 10 that the reduction graph of $\mathbf{snr}_p(u)$ has at most $N - 1$ cyclic components. The other two vertices labelled by $\mathbf{p}$ are connected by reality edges to vertices that are not labelled by $\mathbf{p}$, and therefore this component does not disappear. Hence, the reduction graph of $\mathbf{snr}_p(u)$ has exactly $N - 1$ cyclic components.

Let $\mathbf{spr}_p$ be applicable to u . Assume that the reduction graph of $\mathbf{spr}_p(u)$ has less than N cyclic components. Then by Theorem 10, there exist a cyclic component P of $\mathcal{R}_{u,\emptyset}$ consisting of only vertices labelled by $\mathbf{p}$. By Lemma 14, P consists of only two vertices. By Lemma 15, either pp or $\bar{p}\bar{p}$ is a substring of u and thus $\mathbf{spr}_p$ is not applicable to u . This is a contradiction. Consequently, the reduction graph of $\mathbf{spr}_p(u)$ has exactly N cyclic components.

Let $\mathbf{sdr}_{p,q}$ be applicable to u . Assume that the reduction graph of $\mathbf{sdr}_{p,q}(u)$ has less than N cyclic components. Then there exist a cyclic component P in $\mathcal{R}_{u,\emptyset}$ consisting only of vertices labelled by $\mathbf{p}$ and $\mathbf{q}$. Assume that all vertices of P are labelled by $\mathbf{p}$. Then, analogously to the previous case, we deduce that either pp or $\bar{p}\bar{p}$ is a substring of u . Thus $\mathbf{sdr}_{p,q}$ is not applicable to u . This is a contradiction. Similarly, P cannot consist only of vertices labelled by $\mathbf{q}$. Assume then that P consists of vertices that are labelled by both $\mathbf{p}$ and $\mathbf{q}$. By Lemma 14 and the fact that pointers p and q overlap, there are only two vertices labelled by $\mathbf{p}$ in P and two vertices labelled by $\mathbf{q}$ in P . By Lemma 16, either $u = u_1pqu_2qpu_3$ or $u = u_1qpu_2pqu_3$ for some strings $u_1, u_2, u_3 \in \Pi^*$. Thus $\mathbf{sdr}_{p,q}$ is not applicable to u . This is a contradiction. Therefore, such a component P cannot exist and so the reduction graph of $\mathbf{sdr}_{p,q}(u)$ has exactly N cyclic components. ■

The previous theorem can be reformulated as follows, yielding a key property of reduction graphs.

Theorem 18

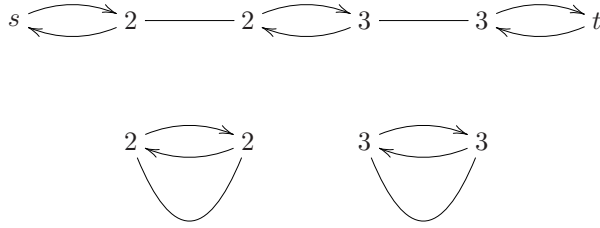
Let N be the number of cyclic components of the reduction graph of legal string u . Then every successful reduction of u has exactly N string negative rules.

The Invariant Theorem [14] (and Chapter 12 in [12]) shows that all successful reductions of a *realistic* string u have the same number of string negative rules. Therefore, Theorem 18 can be seen as a generalization of this result, since it holds for *legal* strings in general. Indeed, the technical framework used in [14] is the MDS descriptor reduction system which is only suited to model realistic strings.

Moreover, Theorem 18 shows that this number N is an elegant graph theoretical property of the reduction graph. As a consequence, it can be efficiently obtained. Since it takes $O(|u|)$ to generate $\mathcal{R}_{u,\emptyset}$, and again $O(|u|)$ to determine the number of connected components of $\mathcal{R}_{u,\emptyset}$, the previous theorem implies that it takes only linear time to determine how many string negative rules are needed to successfully reduce legal string u . Theorem 18 will be used in the next section, when we characterize successfulness in $S \subseteq \{Spr, Sdr\}$.

Example 9

Let $u = 23\bar{2}\bar{4}34$ be a legal string. The reduction graph of u is depicted in Figure 2.11, where $\delta_i = \lambda$ for all $i \in \{0, 1, \dots, 6\}$. By Theorem 18 every reduction of u

Figure 2.13: The reduction graph of $u = 2233$.

has exactly one string negative rule. There are exactly four successful reductions of u , these are $\mathbf{snr}_2 \mathbf{spr}_3 \mathbf{spr}_{\bar{4}}$, $\mathbf{snr}_{\bar{3}} \mathbf{spr}_2 \mathbf{spr}_{\bar{4}}$, $\mathbf{snr}_{\bar{3}} \mathbf{spr}_{\bar{4}} \mathbf{spr}_2$ and $\mathbf{snr}_4 \mathbf{spr}_3 \mathbf{spr}_2$. Notice that each of these reductions has exactly one string negative rule.

Remark

Results in [13] (and Chapter 13 in [12]) show that a successful reduction of a realistic string u has at least one string negative rule if the string has a disjoint cycle. Clearly, the notions of disjoint cycle and (cyclic) component are related. It is easy to verify that every disjoint cycle of a string can be found as a connected component of the reduction graph of the string, although that might be the linear component. As an example, consider the realistic string $u = \pi_3(M_1 M_2 M_3) = 2233$. This realistic string has three disjoint cycles $\{22\}$, $\{33\}$, and $\{23, 32\}$ corresponding to the connected components of the reduction graph of u , see Figure 2.13. This correspondence is not a bijection for all legal strings, not even for realistic ones. E.g., realistic string $u = \pi_3(M_3 \bar{M}_1 M_2) = 3\bar{2}23$ has only a single disjoint cycle $\{33\}$ whereas its reduction graph has two components, one linear and one cyclic. Hence, the number of disjoint cycles cannot be used to characterize the number of string negative rules present in every successful reduction of u .

It is easy to see that for legal string u and $D \subseteq \text{dom}(u)$, $\mathcal{R}_{\text{rem}_D(u), \emptyset}$ is isomorphic to $\mathcal{R}_{u, D}$ modulo the labels of the edges. Now, we have the following corollary to Theorems 18.

Corollary 19

Let u be a legal string, $D \subseteq \text{dom}(u)$, and let N be the number of cyclic components of $\mathcal{R}_{u, D}$. Then every reduction φ of u with $\text{dom}(\varphi(u)) = D$ has exactly N string negative rules.

Proof

Let φ be a reduction of u with $\text{dom}(\varphi(u)) = D$. Then by Theorem 6, φ is a successful reduction of $\text{rem}_D(u)$. Since $\mathcal{R}_{u, D}$ is isomorphic to $\mathcal{R}_{\text{rem}_D(u), \emptyset}$ modulo the labels of the edges, $\mathcal{R}_{\text{rem}_D(u), \emptyset}$ has N cyclic components. By Theorem 18, φ has exactly N string negative rules. ■

2.10 Successfulness of Legal Strings

In [13] (and Chapter 13 in [12]) an elementary characterization of the *realistic* strings that are successful in any given $S \subseteq \{Snr, Spr, Sdr\}$ is presented. This is helpful in applying Theorem 11, where reducibility of legal string u into legal string v is translated into successfulness of $rem_D(u)$ with $D = dom(v)$. Unfortunately, even when u is a realistic string, $rem_D(u)$ for some $D \subseteq dom(u)$ is not necessary a realistic string. For example, $u = \pi_5(M_1M_2\bar{M}_3M_4M_5) = 2234\bar{3}455$ is realistic, while $rem_{\{4\}}(u) = 223\bar{3}55$ is not. As a matter of fact, it can be shown that this legal string is not even *realizable*, that is, the legal string can not be transformed into a realistic string by renaming pointers. Formally, legal string v is *realizable* if there exists a homomorphism $h : \Pi \rightarrow \Pi$ with $h(\bar{p}) = \bar{h(p)}$ for all $p \in \Pi$ such that $h(v)$ is realistic. Thus, e.g., $223\bar{3}44$ and $\bar{2}\bar{2}\bar{3}344$ are also not realistic.

In this section we generalize the results from [13], and give a characterization of the *legal* strings that are successful in any given $S \subseteq \{Snr, Spr, Sdr\}$. Theorems 22, 23, and 25 are the ‘legal counterparts’ of Theorems 8, 9, and 6 in [13], respectively. These results are independent of the results in the previous sections of this chapter. On the other hand, Theorems 27, 28, and 30 (the ‘legal counterparts’ of Theorems 14, 11, and 13 in [13], respectively) rely heavily on Theorem 18.

2.10.1 Trivial Generalizations and Known Results

In the cases of $\{Snr, Spr\}$, $\{Snr, Sdr\}$, and $\{Snr, Spr, Sdr\}$, the characterizations from [13] (and Chapter 13 in [12]) and their proofs, although stated in terms of realistic strings, are valid for legal strings in general. The results are given below for completeness. First we restate Lemma 4 and Lemma 7 from [13] respectively, which will be used in our considerations below.

Lemma 20

Let $u = \alpha v \beta$ be a legal string such that v is also a legal string, and let $S \subseteq \{Snr, Spr, Sdr\}$. Then u is successful in S iff both v and $\alpha\beta$ are successful in S .

Lemma 21

Let u be an elementary legal string. Then u is successful in $\{Snr, Spr\}$ iff either u contains at least one positive pointer or $u = pp$ for some $p \in \Pi$.

The following result follows directly from Lemma 20 and Lemma 21. It is the ‘legal version’ of Theorem 8 in [13], which can be taken almost verbatim.

Theorem 22

Let u be a legal string. Then u is successful in $\{Snr, Spr\}$ iff for all legal substrings v of u , if $v = v_1u_1v_2 \cdots v_ju_jv_{j+1}$, where each u_i is a legal substring, then $v_1v_2 \cdots v_{j+1}$ either contains a positive pointer or is successful in $\{Snr\}$.

The previous theorem can be stated more elegantly in terms of connected components of the overlap graph of u , see [12, p.141]. Note that characterization

for case $\{Snr, Spr\}$ refers to the case of $\{Snr\}$. The latter case does differ from the realistic characterization in [13], and is treated later.

Theorem 23

Let u be a legal string. Then u is successful in $\{Snr, Sdr\}$ iff all the pointers in u are negative.

We give now the legal version of Theorem 9.1 in [12] — it is a direct consequence of Theorems 22 and 23. Without restrictions on the types of reduction rules used, every legal string is successful, cf. the remark below the definition of the reduction rules, in Section 2.4.

Theorem 24

Every legal string is successful in $\{Snr, Spr, Sdr\}$.

2.10.2 Non-Trivial Generalizations

The following theorem is the legal counterpart of Theorem 6 in [13]. It turns out to be much less restrictive than the original realistic version.

Theorem 25

Let u be a legal string. Then u is successful in $\{Snr\}$ iff u consists of negative pointers only and no two pointers overlap in u .

Proof

The condition from the statement of the lemma is obviously necessary, because **snr** cannot resolve overlapping or positive pointers. We will now prove that this condition is also sufficient. If no two pointers overlap in u , then there must be a substring pp or $p\bar{p}$ of u for some pointer p . If moreover u consists of negative pointers only, then pp is a substring of u . So **snr** _{p} is applicable to u . Now, again no two pointers overlap in legal string **snr** _{p} (u), and **snr** _{p} (u) consists of negative pointers only. By iteration of this argument we conclude that u is successful in $\{Snr\}$. ■

Observe that the $\{Snr\}$ case is referred to in the characterization of $\{Snr, Spr\}$ in Theorem 22. With the above result we can rephrase the latter result as follows.

Corollary 26

Let u be a legal string. Then u is successful in $\{Snr, Spr\}$ iff for all legal substrings v of u , if $v = v_1u_1v_2 \cdots v_ju_jv_{j+1}$, where each u_i is a legal substring, then, if $v_1v_2 \cdots v_{j+1}$ consists of negative pointers only, they are nonoverlapping.

The following result follows directly from Theorem 18; a successful reduction without string negative rules means that the reduction graph has a single (linear) connected component.

Theorem 27

Let u be a legal string. Then u is successful in $\{Spr, Sdr\}$ iff the reduction graph of u has no cyclic component.

Theorem 14 in [13] is the realistic predecessor of this result, but instead of cyclic components it uses disjoint cycles, cf. Remark 1. The latter notion cannot be used in the general case, as, e.g., the legal string $23\bar{3}24\bar{4}$ has no disjoint cycle, but its reduction graph has one cyclic component. Obviously, the only way to reduce this string is to apply $\mathbf{spr}_3$ and $\mathbf{spr}_4$ (in either order) and then to apply $\mathbf{snr}_2$. In particular, the converse of Corollary 13.1 in [12] does not hold.

In the same way as Theorem 27 relates to Theorem 14 in [13], the following theorem and lemma relate to Theorem 11 and Lemma 12 from [13], respectively.

Theorem 28

Let u be a legal string. Then u is successful in $\{Sdr\}$ iff u consists of negative pointers only and $\mathcal{R}_{u,\emptyset}$ has no cyclic component.

Proof

The forward implication follows directly from Theorem 18 and the fact that $\mathbf{sdr}$ cannot resolve positive pointers. To prove the reverse implication, let u consist of negative pointers only, and let the corresponding reduction graph $\mathcal{R}_{u,\emptyset}$ have no cyclic component. By Theorem 27, there is a successful $\{Spr, Sdr\}$ -reduction φ of u . Since u consists of negative pointers only, φ is a successful $\{Sdr\}$ -reduction of u (as applications of string double rules do not introduce positive pointers). ■

Lemma 29

Let u be an elementary legal string. Then u is successful in $\{Spr\}$ iff u contains a positive pointer and $\mathcal{R}_{u,\emptyset}$ has no cyclic component.

Proof

The forward implication follows directly from Theorem 18. To prove the reverse implication, let u contain a positive pointer and let $\mathcal{R}_{u,\emptyset}$ have no cyclic component. By Lemma 21, there is a successful $\{Snr, Spr\}$ -reduction φ of u . By Theorem 18, φ is a $\{Spr\}$ -reduction of u . ■

The following result follows directly from Lemmas 20 and 29 — it relates to Theorem 13 in [13].

Theorem 30

Let u be a legal string. Then u is successful in $\{Spr\}$ iff for all legal substrings v of u , if $v = v_1u_1v_2 \cdots v_ju_jv_{j+1}$, where each u_i is a legal substring, then $v_1v_2 \cdots v_{j+1}$ either is λ or contains a positive pointer and its reduction graph has no cyclic component.

Similarly to Theorem 22, the previous theorem can be stated in terms of connected components of the overlap graph of u .

Recall that for legal string u and $D \subseteq \text{dom}(u)$, $\mathcal{R}_{\text{rem}_D(u),\emptyset}$ is isomorphic to $\mathcal{R}_{u,D}$ modulo the labels of the edges. Then, by Theorems 11 and 27, we have the following corollary. In this result it is especially apparent that both the linear component *and* the cyclic components of reduction graphs reveal crucial properties concerning reducibility.

Corollary 31

Let u and v be legal strings with $D = \text{dom}(v) \subseteq \text{dom}(u)$. Then u is reducible to v in $\{\text{Spr}, \text{Sdr}\}$ iff $\mathcal{R}_{u,D}$ has no cyclic component and $\text{red}(\mathcal{R}_{u,D}) = v$.

2.11 Discussion

This chapter introduces the concept of breakpoint graph (or reality and desire diagram) into gene assembly models, through the notion of reduction graph. The reduction graph provides surprisingly valuable insights into the gene assembly process. First, it allows one to characterize which gene patterns can occur during the transformation of a given gene from its MIC form to its MAC form. Formally, in the string pointer reduction system we characterize whether a legal string u is reducible to a legal string v for a given set of reduction rule types. The characterization is independent from the chosen subset of the three types of string pointer rules, and it allows us to determine whether a legal string u is reducible to a legal string v in linear time. This generalizes the characterization of successfulness in [13], since the reduced string need not be the empty string. Secondly, the reduction graph allows one to determine the number of loop recombination operations that are necessary in the transformation of a given gene from its MIC form to its MAC form. This result allows for a second generalization of the characterization of successfulness, since we consider legal strings instead of realistic strings.

Reduction graphs are defined for legal strings, the basic notion of the string pointer reduction system that represents the genes. Future research could focus on the possibility of defining a similar notion for overlap graphs, which are used in the the graph pointer reduction system — a model (almost) equivalent to the string pointer reduction system. This would allow results in this chapter to be carried over to the graph pointer reduction system.

Chapter 3

Strategies of Loop Recombination in Ciliates

Abstract

The concept of breakpoint graph, known from the theory of sorting by reversal, has been successfully applied in the theory of gene assembly in ciliates. We further investigate its usage for gene assembly, and show that the graph allows for an efficient characterization of the possible orders of loop recombination operations (one of the three types of molecular operations that accomplish gene assembly) for a given gene during gene assembly. The characterization is based on spanning trees within a graph built upon the connected components in the breakpoint graph. We work in the abstract and more general setting of so-called legal strings.

3.1 Introduction

Gene assembly is an involved DNA transformation process in ciliates (a large group of single cell organisms) which transforms a nucleus (the micronucleus) into a functionally different nucleus (the macronucleus). The process is accomplished using three types of DNA splicing operations, which operate on special DNA sequences called pointers. Each pointer can be seen as a breakpoint with a ‘tag’ which specifies how the splicing should be done, ensuring that the end result is fixed. The process however is not deterministic: for every gene in its micronuclear form, there can be several sequences of operations, called strategies, to transform this gene to its macronuclear form. For a given micronuclear gene, strategies may differ in the number of operations. It has been shown however that the number of loop recombination operations is independent of the chosen strategy [14, 12], and that this number can be efficiently calculated [6, 5].

In this chapter we characterize for a given set of pointers D , whether or not there is a strategy that applies loop recombination (called string negative rule in the formal model that we use) on exactly these pointers. This result depends

heavily on the reduction graph, which is motivated by the breakpoint graph in the theory of sorting by reversal [21, 23, 1] since it adopts the concept of reality-and-desire for DNA sequences with breakpoints. More specifically, we define a graph, called the pointer-component graph, ‘on top of’ the reduction graph, thereby depicting the distribution of pointers over the connected components of the reduction graph [6, 5]. We show that one can apply loop recombination on the pointers in D exactly when D forms a spanning tree in the pointer-component graph. This characterization implies an efficient algorithm. Also, we characterize in which order the pointers of D can possibly be applied in strategies.

This chapter is organized as follows. In Section 3.2 we recall basic notions and terminology mainly concerning strings and graphs, and in Section 3.3 we recall a formal model of the gene assembly process: the string pointer reduction system. In Section 3.4 we recall the notion of reduction graph and some theorems related to this notion. In Section 3.5 we define the pointer-component graph, a graph that depends on the reduction graph, and we discuss an operation on this graph that captures the application of string pointer rules. In Section 3.6 we show that the spanning trees of the pointer-component graphs reveal crucial information concerning applicability of string negative rules. Section 3.7 shows that merging and splitting of vertices in pointer-component graphs relate to the removal of pointers. Using the results of Sections 3.6 and 3.7, we characterize in Section 3.8 for a given set of pointers D , whether or not there is a strategy that applies string negative rules on exactly these pointers. Section 3.9 strengthens results of Section 3.8 by also characterizing in which order the string negative rules can be applied on the pointers. We conclude this chapter with Section 3.10. A conference edition of this chapter, containing selected results without proofs, was presented at CompLife ’06 [3].

3.2 Basic Notions and Notation

In this section we recall some basic notions concerning functions, strings, and graphs. We do this mainly to fix the basic notation and terminology.

The restriction of f to a subset A of X is denoted by $f|_A$, and for $D \subseteq X$ we denote by $f(D)$ the set $\{f(x) \mid x \in D\}$. Let $\prec$ be a binary relation over a finite set X . Then x_i and x_j in X are called *independent*, if they are incomparable in the transitive closure of $\prec$. A *topological ordering* of $\prec$ is a linear ordering $(x_1, \dots, x_n)$ of X such that if $x_i \prec x_j$, then $i < j$.

We will use λ to denote the empty string. For strings u and v , we say that v is a *substring* of u if $u = w_1vw_2$, for some strings w_1, w_2 ; we also say that v *occurs in* u .

Let Σ and Δ be alphabets. For $\Gamma \subseteq \Sigma$, we denote by $\text{eraser} : \Sigma^* \rightarrow \Delta^*$ the homomorphism defined by

$$\varphi(a) = \begin{cases} a & \text{if } a \notin \Gamma \\ \lambda & \text{if } a \in \Gamma \end{cases},$$

for all $a \in \Sigma$.

We now turn to graphs. We will only consider undirected graphs. A *graph* is a tuple $G = (V, E)$, where V is a finite set and $E \subseteq \{\{x, y\} \mid x, y \in V\}$. The elements of V are called *vertices* and the elements of E are called *edges*. We also write $o(G) = |V|$. We allow $x = y$, and therefore edges can be of the form $\{x, x\} = \{x\}$ — an edge of this form should be seen as an edge connecting x to x , i.e., a ‘loop’ for x . Vertex x is *isolated* in G if there is no edge e of G with $x \in e$. The *restriction of G to $E' \subseteq E$* , denoted by $G|_{E'}$, is (V, E') .

A *multigraph* is a graph $G = (V, E, \epsilon)$, where parallel edges are possible. Therefore, E is a finite set of edges and $\epsilon : E \rightarrow \{\{x, y\} \mid x, y \in V\}$ is the *endpoint mapping*. Clearly, if ϵ is injective, then such a multigraph is equivalent to a graph. We let MGr denote the set of all multigraphs.

A *2-edge coloured graph* is a graph $G = (V, E_1, E_2, f, s, t)$ where E_1 and E_2 are two finite (not necessarily disjoint) sets of edges, $s, t \in V$ are two distinct vertices called the *source vertex* and the *target vertex*, respectively, and there is a vertex labelling function $f : V \setminus \{s, t\} \rightarrow \Gamma$ for some finite set Γ . The elements of Γ are the *vertex labels*. We use 2EGr to denote the set of all 2-edge coloured graphs.

Notions such as isomorphisms, paths, connectedness, and trees for graphs carry over to these two types of graphs. For example, for a multigraph $G = (V, E, \epsilon)$ and $E' \subseteq E$, we have $G|_{E'} = (V, E', \epsilon|_{E'})$. Care must be taken for isomorphisms. Multigraphs $G = (V, E, \epsilon)$ and $G' = (V', E', \epsilon')$ are *isomorphic* if there is a bijection $\alpha : V \rightarrow V'$ such that $\alpha\epsilon = \epsilon'$, or more precisely, for $e \in E$, $\epsilon(e) = \{v_1, v_2\}$ implies $\epsilon'(e) = \{\alpha(v_1), \alpha(v_2)\}$. Note that the sets of edges of G and G' are identical. Also, 2-edge coloured graphs $G = (V, E_1, E_2, f, s, t)$ and $G' = (V', E'_1, E'_2, f', s', t')$ are *isomorphic* if there is a bijection $\alpha : V \rightarrow V'$ such that $\alpha(s) = s'$, $\alpha(t) = t'$, $f(v) = f'(\alpha(v))$ for all $v \in V$, and $\{x, y\} \in E_i$ iff $\{\alpha(x), \alpha(y)\} \in E'_i$, for all $x, y \in V$, and $i \in \{1, 2\}$.

For 2-edge coloured graphs G , we say that a path $\pi = e_1 e_2 \cdots e_n$ in G is an *alternating path in G* if, for $1 \leq i < n$, both $e_i \in E_1$ and $e_{i+1} \in E_2$, or the other way around.

3.3 String Pointer Reduction System

Three (almost) equivalent formal models for gene assembly were considered in [15, 11, 12]. In this section we briefly recall the one that we will use in this chapter: the string pointer reduction system. For a detailed motivation and other results concerning this model we refer to [12].

We fix $\kappa \geq 2$, and define the alphabet $\Delta = \{2, 3, \dots, \kappa\}$. For $D \subseteq \Delta$, we define $\bar{D} = \{\bar{a} \mid a \in D\}$ with $D \cap \bar{D} = \emptyset$, and we define $\Pi = \Delta \cup \bar{\Delta}$. The elements of Π will be called *pointers*. Since we work in the general framework of legal strings, the exact identities of the elements in Δ is irrelevant, in fact, any finite set Δ would suffice. However, we respect the convention of denoting a pointer by an integer larger than 1 with possibly a bar. The name ‘pointer’ is lent from computer science due to its similarities with pointers defined here (see, e.g., Chapter 16 in [12]).

	5		4		3		7		2		5		6		2		7		3		4		6	
--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--

Figure 3.1: Sequence of pointers represented by the legal string $u = 54372562\bar{7}346$.

We use the ‘bar operator’ to move from Δ to $\bar{\Delta}$ and back from $\bar{\Delta}$ to Δ . Hence, for $p \in \Pi$, $\bar{\bar{p}} = p$. For a string $u = x_1x_2 \cdots x_n$ with $x_i \in \Pi$, the *inverse* of u is the string $\bar{u} = \bar{x}_n\bar{x}_{n-1} \cdots \bar{x}_1$. For $p \in \Pi$, we define $\mathbf{p}$ to be p if $p \in \Delta$, and $\bar{p}$ if $p \in \bar{\Delta}$, i.e., $\mathbf{p}$ is the ‘unbarred’ variant of p . The *domain* of a string $v \in \Pi^*$ is $\text{dom}(v) = \{\mathbf{p} \mid p \text{ occurs in } v\}$. A *legal string* is a string $u \in \Pi^*$ such that for each $p \in \Pi$ that occurs in u , u contains exactly two occurrences from $\{p, \bar{p}\}$. For a pointer p and a legal string u , if both p and $\bar{p}$ occur in u then we say that both p and $\bar{p}$ are *positive* in u ; if on the other hand only p or only $\bar{p}$ occurs in u , then both p and $\bar{p}$ are *negative* in u .

Let $u = x_1x_2 \cdots x_n$ be a legal string with $x_i \in \Pi$ for $1 \leq i \leq n$. For a pointer $p \in \Pi$ such that $\{x_i, x_j\} \subseteq \{p, \bar{p}\}$ and $1 \leq i < j \leq n$, the *p-interval* of u is the substring $x_ix_{i+1} \cdots x_j$. Two distinct pointers $p, q \in \Pi$ *overlap* in u if both $\mathbf{q} \in \text{dom}(I_p)$ and $\mathbf{p} \in \text{dom}(I_q)$, where I_p (I_q , resp.) is the *p-interval* (*q-interval*, resp.) of u .

Example 1

String $u = 4\bar{3}7\bar{7}\bar{4}3$ is a legal string. However, $v = 424$ is not a legal string. Also, $\text{dom}(u) = \{3, 4, 7\}$ and $\bar{u} = \bar{3}477\bar{3}4$. The 3-interval of u is $37\bar{7}43$, and pointers 3 and 4 overlap in u .

A legal string is a representation of a sequence of pointers. The legal string $u = 54372562\bar{7}346$ corresponds to the sequence of pointers in Figure 3.1. Each gene in the micronucleus in ciliates can be represented by such a legal string. For example, the legal string $34456756789\bar{3}2289$ corresponds to the micronuclear form of the gene that corresponds to the actin protein in the stichotrich *Sterkiella nova* (see [22, 12, 8]). Gene assembly transforms each gene in micronuclear form to its macronuclear form by three splicing operations which operate on the pointers. These three operations are formally defined on legal strings through the string pointer reduction system, where each is defined on a specific pattern of the pointers.

The string pointer reduction system consists of three types of reduction rules operating on legal strings. For all $p, q \in \Pi$ with $\mathbf{p} \neq \mathbf{q}$, we define:

- the *string negative rule* for p by $\mathbf{snr}_p(u_1ppu_2) = u_1u_2$,
- the *string positive rule* for p by $\mathbf{spr}_p(u_1pu_2\bar{p}u_3) = u_1\bar{u}_2u_3$,
- the *string double rule* for p, q by $\mathbf{sdr}_{p,q}(u_1pu_2qu_3pu_4qu_5) = u_1u_4u_3u_2u_5$,

where $u_1, u_2, \dots, u_5$ are arbitrary (possibly empty) strings over Π . We also define $\mathbf{Snr} = \{\mathbf{snr}_p \mid p \in \Pi\}$, $\mathbf{Spr} = \{\mathbf{spr}_p \mid p \in \Pi\}$ and $\mathbf{Sdr} = \{\mathbf{sdr}_{p,q} \mid p, q \in \Pi, \mathbf{p} \neq \mathbf{q}\}$ to be the sets containing all the reduction rules of a specific type.

Note that each of these rules is defined only on legal strings that satisfy the given form. For example, $\mathbf{spr}_2$ is defined on the legal string $\bar{2}323$, however $\mathbf{spr}_2$ is not defined on this legal string. Also note that for every non-empty legal string there is at least one reduction rule applicable. Indeed, every non-empty legal string for which no string positive rule and no string double rule is applicable must have only non-overlapping negative pointers, thus there is a string negative rule which is applicable. This is formalized in Theorem 1 below.

The *domain* of a reduction rule ρ , denoted by $\text{dom}(\rho)$, is defined by $\text{dom}(\mathbf{snr}_p) = \text{dom}(\mathbf{spr}_p) = \{\mathbf{p}\}$ and $\text{dom}(\mathbf{sdr}_{p,q}) = \{\mathbf{p}, \mathbf{q}\}$ for $p, q \in \Pi$. For a composition $\varphi = \varphi_n \cdots \varphi_2 \varphi_1$ of reduction rules $\varphi_1, \varphi_2, \dots, \varphi_n$, the *domain*, denoted by $\text{dom}(\varphi)$, is defined by $\text{dom}(\varphi) = \text{dom}(\varphi_1) \cup \text{dom}(\varphi_2) \cup \cdots \cup \text{dom}(\varphi_n)$.

Example 2

The domain of $\varphi = \mathbf{snr}_2 \mathbf{spr}_{\bar{4}} \mathbf{sdr}_{7,5} \mathbf{snr}_{\bar{9}}$ is $\text{dom}(\varphi) = \{2, 4, 5, 7, 9\}$.

Let $S \subseteq \{Snr, Spr, Sdr\}$. Then a composition φ of reduction rules from S is called an (S -)*reduction*. Let u be a legal string. We say that φ is a *reduction of u* , if φ is a reduction and φ is applicable to (i.e., defined on) u . A *successful reduction of u* is a reduction of u such that $\varphi(u) = \lambda$. We then also say that φ is *successful for u* . We say that u is *successful in S* if there is a successful S -reduction of u . Note that if φ is a reduction of u , then $\text{dom}(\varphi) = \text{dom}(u) \setminus \text{dom}(\varphi(u))$.

Example 3

Again let $u = \bar{4}37\bar{7}\bar{4}3$. Then $\varphi_1 = \mathbf{sdr}_{\bar{4},3} \mathbf{spr}_7$ is a successful $\{Spr, Sdr\}$ -reduction of u . However, both $\varphi_2 = \mathbf{snr}_3 \mathbf{spr}_7$ and $\varphi_3 = \mathbf{snr}_8$ are *not* reductions of u .

Example 4

If we again consider the legal string $34456756789\bar{3}\bar{2}289$, which represents the micronuclear form of the gene corresponding to the actin protein in the stichotrich *Sterkiella nova*, then $\mathbf{spr}_3 \mathbf{sdr}_{8,9} \mathbf{snr}_7 \mathbf{sdr}_{5,6} \mathbf{snr}_4 \mathbf{spr}_2$ is a successful reduction of this legal string. Therefore, this sequence of operations transforms the gene from its micronuclear form to its macronuclear form.

We say that a linear ordering $L = (p_1, \dots, p_n)$ of a subset of $\text{dom}(\varphi)$ is the *Snr-order of φ* , if $\varphi = \varphi_{n+1} \mathbf{snr}_{\tilde{p}_n} \varphi_n \mathbf{snr}_{\tilde{p}_{n-1}} \cdots \varphi_2 \mathbf{snr}_{\tilde{p}_1} \varphi_1$ for some (possible empty) $\{Spr, Sdr\}$ -reductions $\varphi_1, \varphi_2, \dots, \varphi_{n+1}$ and $\tilde{p}_i \in \{p_i, \bar{p}_i\}$ for $1 \leq i \leq n$ and $n \geq 0$. Moreover, we define $\text{snrdom}(\varphi) = \{p_1, \dots, p_n\}$.

Example 5

The *Snr-order* of $\varphi = \mathbf{snr}_2 \mathbf{spr}_{\bar{4}} \mathbf{sdr}_{7,5} \mathbf{snr}_{\bar{9}}$ is $(9, 2)$, and $\text{snrdom}(\varphi) = \{2, 9\}$.

Since for every (non-empty) legal string there is an applicable reduction rule, by iterating this argument, we have the following well-known result.

Theorem 1

For every legal string u there is a successful reduction of u .

3.4 Reduction Graph

In this section we recall the definition of reduction graph and some results concerning this graph. First we give the definition of pointer removal operations on strings, see also [6].

Definition 2

For a subset $D \subseteq \Delta$, the D -removal operation, denoted by rem_D , is defined by $\text{rem}_D = \text{erase}_{D \cup \bar{D}}$. We also refer to rem_D operations, for all $D \subseteq \Delta$, as *pointer removal operations*. ■

Note that for each legal string u , $\text{rem}_D(u)$ is a legal string.

Example 6

Let $u = 54372562\bar{7}346$ be a legal string. We will use this legal string as our running example for this chapter. For $D = \{4, 6, 7, 9\}$, we have $\text{rem}_D(u) = 532523$.

The next lemma is an easy consequence of Lemma 8 from [6]. It cannot be extended to Snr rules: if $\mathbf{snr}_p$ is applicable to $\text{rem}_D(u)$, then it is not necessarily applicable to u .

Lemma 3

Let u be a legal string, let φ be a composition of reduction rules that does not contain string negative rules, and let $D = \text{dom}(u) \setminus \text{dom}(\varphi)$. Then φ is a reduction of u iff φ is a (successful) reduction of $\text{rem}_D(u)$.

The string negative rules in a reduction can be ‘postponed’ without affecting the applicability. More precisely, if $\varphi = \varphi_2 \rho \mathbf{snr}_p \varphi_1$ is a reduction of a legal string u , with $p \in \Pi$, ρ a string positive rule or string double rule, and φ_1, φ_2 arbitrary compositions of reduction rules, then there is a $\tilde{p} \in \{p, \bar{p}\}$ such that $\varphi_2 \mathbf{snr}_{\tilde{p}} \rho \varphi_1$ is a reduction of u . Thus we can separate each reduction into a sequence without Snr rules, and a tail of Snr rules. We often use this ‘normal form’.

Example 7

We continue the example. Since $\varphi = \mathbf{snr}_6 \mathbf{snr}_2 \mathbf{spr}_7 \mathbf{snr}_4 \mathbf{sdr}_{5,3}$ is a successful reduction of u , it follows that $\varphi' = \mathbf{snr}_6 \mathbf{snr}_2 \mathbf{snr}_{\tilde{4}} \mathbf{spr}_7 \mathbf{sdr}_{5,3}$ is also a successful reduction of u for some $\tilde{4} \in \{4, \bar{4}\}$. One can verify that we can take $\tilde{4} = 4$. However, $\varphi'' = \mathbf{snr}_4 \mathbf{snr}_6 \mathbf{snr}_2 \mathbf{spr}_7 \mathbf{sdr}_{5,3}$ is *not* a successful reduction of u . If we consider the legal string $7\bar{2}\bar{2}\bar{7}$, for which $\mathbf{spr}_7 \mathbf{snr}_2$ is a successful reduction, then by postponing the string negative rule, $\mathbf{snr}_2 \mathbf{spr}_7$ is also a successful reduction of this legal string.

Figure 3.2 illustrates Lemma 3 when φ is in this normal form: $\varphi = \varphi_2 \varphi_1$ is a successful reduction of u , where φ_1 is a $\{Spr, Sdr\}$ -reduction and φ_2 is a $\{Snr\}$ -reduction with $\text{dom}(\varphi_2) = D$.

We are now ready to recall the definition of reduction graph. It was introduced in [6], and we restate it here in a less general form. A reduction graph is a 2-edge

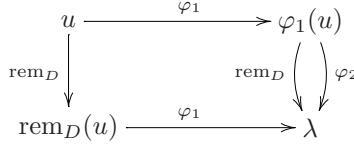


Figure 3.2: An illustration of Lemma 3: $\varphi = \varphi_2$ φ_1 is a successful reduction of u , where φ_1 is a $\{Spr, Sdr\}$ -reduction and φ_2 is a $\{Snr\}$ -reduction with $\text{dom}(\varphi_2) = D$.

coloured graph where the two types of edges are called *reality edges* and *desire edges*. Moreover, all vertices, except for two distinct vertices s and t , are labelled by an element from Δ . Recall that the physical representation of our running example $u = 54372562\bar{7}346$ is given in Figure 3.1. The reduction graph is defined in such a way that (1) each (occurrence of a) pointer of u appears twice (in unbarred form) as a vertex in the graph to represent both sides of the pointer in Figure 3.1, (2) the reality edges (depicted as ‘double edges’ to distinguish them from the desire edges) represent the segments between the pointers, (3) the desire edges represent which segments should be glued to each other when operations are applied on the corresponding pointers. Positive pointers are connected by crossing desire edges (cf. pointer 7 in Figure 3.3), while negative pointers are connected by parallel desire edges. We refer to [6] for a more elaborate motivation and for more examples and results concerning this graph. The notion is similar to the breakpoint graph (or reality-and-desire diagram) known from another branch of DNA processing theory called sorting by reversal, see e.g. [23] and [21].

Definition 4

Let $u = p_1 p_2 \cdots p_n$ with $p_1, \dots, p_n \in \Pi$ be a legal string. The *reduction graph* of u , denoted by $\mathcal{R}_u$, is a 2-edge coloured graph (V, E_1, E_2, f, s, t) , where

$$V = \{I_1, I_2, \dots, I_n\} \cup \{I'_1, I'_2, \dots, I'_n\} \cup \{s, t\},$$

$$E_1 = \{e_0, e_1, \dots, e_n\} \text{ with } e_i = \{I'_i, I_{i+1}\} \text{ for } 1 < i < n, e_0 = \{s, I_1\}, e_n = \{I'_n, t\},$$

$$E_2 = \{ \{I'_i, I_j\}, \{I_i, I'_j\} \mid i, j \in \{1, 2, \dots, n\} \text{ with } i \neq j \text{ and } p_i = p_j \} \cup \\ \{ \{I_i, I_j\}, \{I'_i, I'_j\} \mid i, j \in \{1, 2, \dots, n\} \text{ and } p_i = \bar{p}_j \}, \text{ and}$$

$$f(I_i) = f(I'_i) = \mathbf{p}_i \text{ for } 1 \leq i \leq n.$$

■

The edges of E_1 are called *reality edges*, and the edges of E_2 are called *desire edges*. Notice that for each $p \in \text{dom}(u)$, the reduction graph of u has exactly two desire edges containing vertices labelled by p .

In depictions of reduction graphs, we will represent the vertices (except for s and t) by their labels, because the exact identities of the vertices are not essential

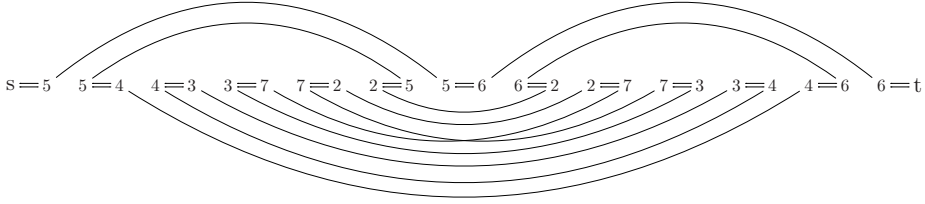


Figure 3.3: The reduction graph $\mathcal{R}_u$ of u from Example 8.

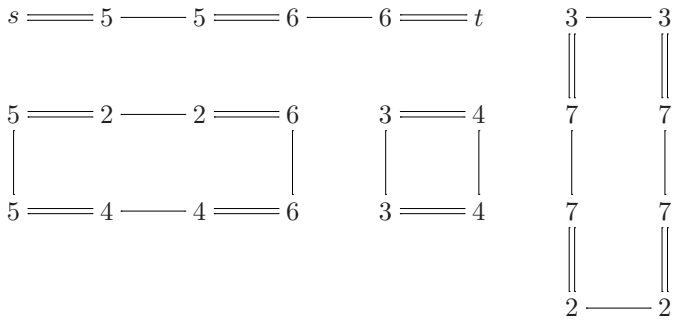


Figure 3.4: The reduction graph of Figure 3.3.

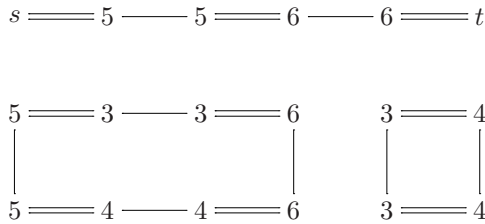


Figure 3.5: The reduction graph $\mathcal{R}_{rem_{\{2,7\}}(u)}$ from the Example.

for the problems considered in this chapter. We will also depict reality edges as ‘double edges’ to distinguish them from the desire edges.

Example 8

We continue the example. Reduction graph $\mathcal{R}_u$ is given in Figure 3.3. The same graph is again depicted in Figure 3.4 – we have only rearranged the vertices. Also, $\mathcal{R}_{rem_{\{2,7\}}(u)}$ is given in Figure 3.5.

Each reduction graph has a connected component with a linear structure containing both the source and the target vertex [6]. This connected component is called the *linear component* of the reduction graph. The other connected components are called *cyclic components* because of their structure.

The definition of reduction functions and the remaining results are also taken from [6]. The p -reduction function removes vertices labelled by p and ‘contracts’ alternating paths via these vertices into a single edge.

Definition 5

For each vertex label p , we define the p -reduction function $rf_p : 2\text{EGr} \rightarrow 2\text{EGr}$, for $G = (V, E_1, E_2, f, s, t) \in 2\text{EGr}$, by

$$rf_p(G) = (V', (E_1 \setminus E_{rem}) \cup E_{add}, E_2 \setminus E_{rem}, f|_{V'}, s, t),$$

with

$$\begin{aligned} V' &= \{s, t\} \cup \{v \in V \setminus \{s, t\} \mid f(v) \neq p\}, \\ E_{rem} &= \{e \in E_1 \cup E_2 \mid f(x) = p \text{ for some } x \in e\}, \text{ and} \\ E_{add} &= \{\{y_1, y_2\} \mid e_1 e_2 \cdots e_n \text{ with } n > 2 \text{ is an alternating path in } G \\ &\quad \text{with } y_1 \in e_1, y_2 \in e_n, f(y_1) \neq p \neq f(y_2), \text{ and} \\ &\quad f(x) = p \text{ for all } x \in e_i, 1 < i < n\}. \end{aligned}$$

■

Reduction functions commute under composition. Thus, for a reduction graph $\mathcal{R}_{rem_D(u)}$ and pointers p and q , we have

$$(rf_q rf_p)(\mathcal{R}_u) = (rf_p rf_q)(\mathcal{R}_u).$$

Any reduction can be simulated, on the level of reduction graphs, by a sequence of reduction functions with the same domain, cf. [6, Theorem 17].

Theorem 6

Let u be a legal string, and let φ be a reduction of u . Then

$$(rf_{p_n} \cdots rf_{p_2} rf_{p_1})(\mathcal{R}_u) \approx \mathcal{R}_{\varphi(u)},$$

where $\text{dom}(\varphi) = \{p_1, p_2, \dots, p_n\}$.

The next lemma is an easy consequence from Lemma 22 and Lemma 23 in [6].

$$s \equiv \mathbf{p} \text{ --- } \mathbf{p} \equiv \mathbf{q} \text{ --- } \mathbf{q} \equiv \mathbf{p} \text{ --- } \mathbf{p} \equiv \mathbf{q} \text{ --- } \mathbf{q} \equiv t$$

Figure 3.6: The reduction graph of $pq\bar{p}q$ (and $pqpq$).

Lemma 7

Let u be a legal string and let $p \in \Pi$. Then $\mathcal{R}_u$ has a cyclic component C consisting of only vertices labelled by $\mathbf{p}$ iff either pp or $\bar{p}\bar{p}$ is a substring of u . Moreover, if C exists, then it has exactly two vertices.

One of the motivations for the reduction graph is the easy determination of the number of string negative rules needed in each successful reduction [6, Theorem 26].

Theorem 8

Let N be the number of cyclic components in the reduction graph of legal string u . Then every successful reduction of u has exactly N string negative rules.

Example 9

We continue the example. Since $\mathcal{R}_u$ has three cyclic components, by Theorem 8, every successful reduction φ of u has exactly three string negative rules. For example $\varphi = \mathbf{snr}_6 \mathbf{snr}_2 \mathbf{spr}_7 \mathbf{snr}_4 \mathbf{sdr}_{5,3}$ is a successful reduction of u . Indeed, φ has exactly three string negative rules. Alternatively, $\mathbf{snr}_6 \mathbf{snr}_4 \mathbf{snr}_3 \mathbf{spr}_2 \mathbf{spr}_5 \mathbf{spr}_7$ is also a successful reduction of u , with a different number of ($\mathbf{spr}$ and $\mathbf{sdr}$) operations.

The previous theorem and example should clarify that the reduction graph reveals crucial properties concerning the string negative rule. We now further investigate the string negative rule, and show that many more properties of this rule can be revealed using the reduction graph.

However, the reduction graph does not seem to be well suited to prove properties of the string positive rule and string double rule. If we for example consider legal strings $u = pq\bar{p}q$ and $v = pqpq$ for some distinct $p, q \in \Pi$, then u has a unique successful reduction $\varphi_1 = \mathbf{spr}_{\bar{q}} \mathbf{spr}_p$ and v has a unique successful reduction $\varphi_2 = \mathbf{sdr}_{p,q}$. Thus u must necessarily be reduced by string positive rules, while v must necessarily be reduced by a string double rule. However, the reduction graph of u and the reduction graph of v are isomorphic, as shown in Figure 3.6. Also, whether or not pointers overlap is not preserved by reduction graphs. For example, the reduction graphs of legal strings $pqr\bar{p}qr$ and $pqr\bar{p}qr$ for distinct pointers p, q and r are isomorphic, however p and r do not overlap in the first legal string, but they do overlap in the latter legal string.

The next lemma is an easy consequence of Lemma 3 and Theorem 8.

Lemma 9

Let u be a legal string, and let $D \subseteq \text{dom}(u)$. There is a $\{\mathbf{Spr}, \mathbf{Sdr}\}$ -reduction φ of u with $\text{dom}(\varphi(u)) = D$ iff $\mathcal{R}_{\text{rem}_D(u)}$ does not contain cyclic components.

Proof

There is a $\{Spr, Sdr\}$ -reduction φ of u with $\text{dom}(\varphi(u)) = D$ iff there is a successful $\{Spr, Sdr\}$ -reduction of $\text{rem}_D(u)$ (by Lemma 3) iff $\mathcal{R}_{\text{rem}_D(u)}$ does not contain cyclic components (by Theorem 8). ■

Using Theorem 8 and Lemma 9 we obtain a first characterization of the sets of pointers that are used in string negative rules in successful reductions.

Lemma 10

Let u be a legal string, and let $D \subseteq \text{dom}(u)$. There is a successful reduction φ of u with $\text{snrdom}(\varphi) = D$ iff $\mathcal{R}_{\text{rem}_D(u)}$ and $\mathcal{R}_u$ have 0 and $|D|$ cyclic components, respectively.

Proof

We first prove the forward implication. Since we can postpone the string negative rules, there is a successful reduction $\varphi' = \varphi'_2 \varphi'_1$ of u , where φ'_1 is a $\{Spr, Sdr\}$ -reduction and φ'_2 is a $\{Snr\}$ -reduction with $\text{dom}(\varphi'_2) = D$. By Lemma 9, $\mathcal{R}_{\text{rem}_D(u)}$ does not contain cyclic components. By Theorem 8, $\mathcal{R}_u$ has $|D|$ cyclic components.

We now prove the reverse implication. By Lemma 9, there is a successful reduction $\varphi = \varphi_2 \varphi_1$ of u , where φ_1 is a $\{Spr, Sdr\}$ -reduction and $\text{dom}(\varphi_2) = D$. Since $\mathcal{R}_u$ has $|D|$ cyclic components, by Theorem 8, every pointer in D is used in a string negative rule, and thus φ_2 is a $\{Snr\}$ -reduction. ■

3.5 Pointer-Component Graphs

If it is clear from the context which legal string u is meant, we will denote by ζ the set of connected components of the reduction graph of u . We now define a graph on ζ that we will use throughout the rest of this chapter. The graph represents how the labels of a reduction graph are distributed among its connected components. This graph is particularly useful in determining which sets D of pointers correspond to strategies that apply loop recombination operations on exactly the pointers of D .

Definition 11

Let u be a legal string. The *pointer-component graph of u (or of $\mathcal{R}_u$)*, denoted by $\mathcal{PC}_u$, is a multigraph (ζ, E, ϵ) , where $E = \text{dom}(u)$ and ϵ is, for $e \in E$, defined by $\epsilon(e) = \{C \in \zeta \mid C \text{ contains vertices labelled by } e\}$. ■

Note that for each $e \in \text{dom}(u)$, there are exactly two desire edges connecting vertices labelled by e , thus $1 \leq |\epsilon(e)| \leq 2$, and therefore ϵ is well defined.

Example 10

We continue the example. Consider $\mathcal{R}_u$ shown in Figure 3.4. Let us define C_1 to be the cyclic component with a vertex labelled by 7, C_2 to be the cyclic component with a vertex labelled by 5, C_3 to be the third cyclic component, and R to be the linear component. Then $\zeta = \{C_1, C_2, C_3, R\}$. The pointer-component graph $\mathcal{PC}_u = (\zeta, \text{dom}(u), \epsilon)$ of u is given in Figure 3.7. As C_1 contains all four vertices labelled by 7, this results in a loop for C_1 in $\mathcal{PC}_u$.

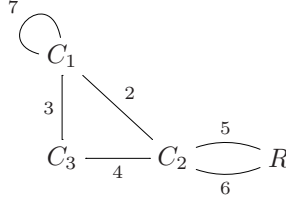


Figure 3.7: The graph $\mathcal{PC}_u$ from the Example.

By the definition of pointer-component graph and Theorem 8, every successful reduction of a legal string u has exactly $o(\mathcal{PC}_u) - 1$ string negative rules (recall that $o(\mathcal{PC}_u)$ denotes the number of vertices of $\mathcal{PC}_u$). Let ρ be a reduction rule applicable to u . Then by Theorem 1, there is a successful reduction φ' of $\rho(u)$. Hence, $\varphi'\rho$ is a successful reduction of u . Thus, if ρ is a string positive rule or string double rule, then $o(\mathcal{PC}_{\rho(u)}) = o(\mathcal{PC}_u)$, and if ρ is a string negative rule, then $o(\mathcal{PC}_{\rho(u)}) = o(\mathcal{PC}_u) - 1$. Thus we have the following result.

Theorem 12

Let φ be a reduction of a legal string u with $N = |\text{snrdom}(\varphi)|$. Then $o(\mathcal{PC}_{\varphi(u)}) = o(\mathcal{PC}_u) - N$.

For a reduction φ of a legal string u , the difference between $\mathcal{R}_u$ and $\mathcal{R}_{\varphi(u)}$ is formulated in Theorem 6 in terms of reduction functions. We now reformulate this result for pointer-component graphs. The difference (up to isomorphism) between the pointer-component graph PC_1 of $\mathcal{R}_u$ and the pointer-component graph PC_2 of $\text{rf}_p(\mathcal{R}_u)$ (assuming rf_p is applicable to $\mathcal{R}_u$) is as follows: in PC_2 edge p is removed and also those vertices v that become isolated, except when v is the linear component (since the linear component always contains the source and target vertex). Since the only legal string u for which the linear component in $\mathcal{PC}_u$ is isolated is the empty string, in this case we obtain a graph containing only one vertex. This is formalized as follows. By abuse of notation we will also denote these functions as reduction functions rf_p .

Definition 13

For each edge p , we define the p -reduction function $\text{rf}_p : \text{MGr} \rightarrow \text{MGr}$, for $G = (V, E, \epsilon) \in \text{MGr}$, by

$$\text{rf}_p(G) = (V', E', \epsilon|E'),$$

where $E' = E \setminus \{p\}$ and $V' = \{v \in V \mid v \in \epsilon(e) \text{ for some } e \in E'\}$ if $E' \neq \emptyset$, and $V' = \{\emptyset\}$ otherwise. ■

Therefore, these reduction functions correctly simulate (up to isomorphism) the effect of applications of a reduction functions on the underlying reduction graph when the reduction functions correspond to an applicable reduction. Note however, when these reduction functions do not correspond to an applicable reduction,

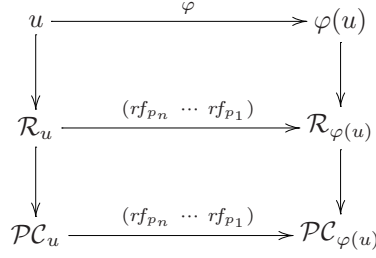


Figure 3.8: An illustration of Theorems 6 and 14 as a commutative diagram.

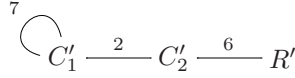


Figure 3.9: Pointer-component graph PC_1 from the Example.

the linear component may become isolated while there are still edges present. Thus in general the reduction functions for pointer-component graphs do not faithfully simulate the reduction functions for reduction graphs.

As a consequence of Theorem 6 we now obtain the following result.

Theorem 14

Let u be a legal string, and let φ be a reduction of u . Then

$$(rf_{p_n} \cdots rf_{p_2} rf_{p_1})(\mathcal{PC}_u) \approx \mathcal{PC}_{\varphi(u)},$$

where $\text{dom}(\varphi) = \{p_1, p_2, \dots, p_n\}$.

Thus, $\mathcal{PC}_{\varphi(u)}$ is obtained from $\mathcal{PC}_u$ (up to isomorphism) by iteratively removing the edges p_i and any isolated vertices that may appear after removing the edges. Thus the only difference between $\mathcal{PC}_{\varphi(u)}$ and $\mathcal{PC}_u|_D$ with $D = \text{dom}(\varphi(u))$ is the possible existence of isolated vertices in $\mathcal{PC}_u|_D$. The only exception is the case $\varphi(u) = \lambda$, since we may not end up with the empty graph (without vertices), and thus one vertex should always remain. Figure 3.8 illustrates Theorems 6 and 14.

Example 11

We continue the example. We have $(\mathbf{snr}_4 \mathbf{sdr}_{5,3})(u) = 627726$. The pointer-component graph PC_1 of this legal string is shown in Figure 3.9. It is easy to see that the graph obtained by applying $(rf_5 rf_4 rf_3)$ to $\mathcal{PC}_u$ (Figure 3.7) is isomorphic to PC_1 .

3.6 Spanning Trees in Pointer-Component Graphs

In this section we consider spanning trees in pointer-component graphs, and we show that there is an intimate connection between these trees and the *Snr*-orders of successful reductions. First we separate loops from other edges in pointer-component graphs.

Definition 15

Let u be a legal string and let $\mathcal{PC}_u = (V, E, \epsilon)$. We define $\text{bridge}(u) = \{e \in E \mid |\epsilon(e)| = 2\}$. ■

Thus, $\text{bridge}(u)$ is the set of vertex labels p for which there are vertices labelled by p in *different* connected components of $\mathcal{R}_u$.

Example 12

We continue the example. We have $\text{bridge}(u) = \{2, 3, 4, 5, 6\}$, and $\text{dom}(u) \setminus \text{bridge}(u) = \{7\}$. Indeed, the only loop in Figure 3.7 is 7, indicating that this pointer occurs only in one connected component of $\mathcal{R}_u$.

The following corollary to Theorem 14 observes that an edge in $\text{dom}(\varphi(u))$ is a loop in $\mathcal{PC}_{\varphi(u)}$ iff it is a loop in $\mathcal{PC}_u$.

Corollary 16

Let u be a legal string and φ a reduction of u . Then $\text{bridge}(\varphi(u)) = \text{dom}(\varphi(u)) \cap \text{bridge}(u) = \text{bridge}(u) \setminus \text{dom}(\varphi)$.

We now characterize $\{Spr, Sdr\}$ -reductions in terms of pointer-component graphs.

Theorem 17

Let u be a legal string, and φ a reduction of u with $D = \text{dom}(\varphi(u))$. Then the following statements are equivalent:

1. φ is a $\{Spr, Sdr\}$ -reduction,
2. $o(\mathcal{PC}_{\varphi(u)}) = o(\mathcal{PC}_u)$,
3. $\mathcal{PC}_{\varphi(u)} \approx \mathcal{PC}_u|_D$,
4. either $o(\mathcal{PC}_u|_D) = 1$ or $\mathcal{PC}_u|_D$ has no isolated vertices.

Proof

Statements (1) and (2) are equivalent by Theorem 12. If (3) holds, then clearly (2) holds. Assume now that (2) holds. Since an application of rf_p that does not remove vertices, only removes edge p , we have, by Theorem 14, $\mathcal{PC}_{\varphi(u)} \approx \mathcal{PC}_u|_D$. Thus (3) holds. Assume now that (3) holds. Since the pointer-component graph of a legal string u does not have isolated vertices except when $u = \lambda$, it follows that (4) holds. Finally, assume that (4) holds. If $o(\mathcal{PC}_u|_D) = 1$, then $o(\mathcal{PC}_u) = 1$ and therefore $1 \leq o(\mathcal{PC}_{\varphi(u)}) \leq o(\mathcal{PC}_u) = 1$. Consequently, $o(\mathcal{PC}_{\varphi(u)}) = o(\mathcal{PC}_u)$,

and (2) holds. On the other hand, if $\mathcal{PC}_u|_D$ has no isolated vertices, then by Theorem 14 the reduction functions corresponding to φ do not remove vertices, and hence (2) holds. ■

Thus, by Theorems 14 and 17, the effect of a rf_p operation where $p \in \text{dom}(\rho)$ for some applicable Spr or Sdr rule ρ is the removal of edge p . We now discuss the Snr case. By Lemma 7, we have the following result. Here v is identical to C in Lemma 7.

Lemma 18

Let u be a legal string and $p \in \text{dom}(u)$. Then $\mathbf{snr}_p$ or $\mathbf{snr}_{\bar{p}}$ is applicable to u iff $p \in \text{bridge}(u)$ and edge p in $\mathcal{PC}_u$ has an endpoint v such that (1) v is not the linear component and (2) p is the only edge with v as an endpoint (v is of degree 1).

Thus the effect of a rf_p operation where $p \in \text{dom}(\rho)$ for some applicable Snr rule ρ is the removal of edge p and the removal of vertex v as in Lemma 18. Vertex v is unique, otherwise $\mathcal{PC}_{\rho(u)}$ would have two vertices less than $\mathcal{PC}_u$ – a contradiction with Theorem 12.

The examples so far have shown connected pointer-component graphs. It turns out that these graphs are *always* connected.

Theorem 19

The pointer-component graph of any legal string is connected.

Proof

Let φ be a successful reduction of a legal string u (φ exists by Theorem 1). Assume that $\mathcal{PC}_u$ is not connected. Since $\mathcal{PC}_\lambda$ is connected, we have $\varphi = \varphi_2 \rho \varphi_1$ for some reduction rule ρ , where $\mathcal{PC}_{\varphi_1(u)}$ is not connected, but $\mathcal{PC}_{\rho\varphi_1(u)}$ is. By the paragraph below Theorem 17, ρ cannot be a string double rule or a string positive rule, and therefore ρ is a string negative rule. By the paragraph below Lemma 18, $\mathcal{PC}_{\rho\varphi_1(u)}$ is obtained from $\mathcal{PC}_{\varphi_1(u)}$ by removing edge $p \in \text{dom}(\rho)$ and removing one of the two endpoints of p . Therefore, $\mathcal{PC}_{\rho\varphi_1(u)}$ has the same number of connected components as $\mathcal{PC}_{\varphi_1(u)}$ – a contradiction. ■

The next theorem characterizes successfulness in $\{Snr\}$ using spanning trees.

Theorem 20

Let u be a legal string. Then u is successful in $\{Snr\}$ iff $\mathcal{PC}_u$ is a tree.

Proof

If u is successful in $\{Snr\}$, then, by Theorem 12, $\mathcal{PC}_u$ has $|\zeta| - 1$ edges. Since $\mathcal{PC}_u$ has $|\zeta|$ vertices and is connected by Theorem 19, it follows that $\mathcal{PC}_u$ is a tree.

If $\mathcal{PC}_u$ is a tree, then $\mathcal{PC}_u$ has $|\zeta| - 1$ edges. Since the number of edges is $|\text{dom}(u)|$, we have $|\text{dom}(u)| = |\zeta| - 1$, and by Theorem 12 every $p \in \text{dom}(u)$ is used in a string negative rule, and thus u is successful in $\{Snr\}$. ■

By Lemma 18 (and the paragraph below it), the possible orders of string negative rules applicable to the legal string u in Theorem 20 is restricted by the form of the tree $\mathcal{PC}_u$. Indeed, if we take the linear component of $\mathcal{R}_u$ as the root of $\mathcal{PC}_u$, then a successful reduction corresponds to a sequence of reduction functions that iteratively removes leaves and their connecting edges. We will discuss this in more detail in Section 3.9.

It turns out that the pointers on which string negative rules are applied in a successful reduction of a legal string u form a spanning tree of $\mathcal{PC}_u$.

Theorem 21

Let u be a legal string, and let $D \subseteq \text{dom}(u)$. If there is a successful reduction φ of u with $\text{snrdom}(\varphi) = D$, then $\mathcal{PC}_u|_D$ is a tree.

Proof

By postponing the string negative rules, there is a successful reduction $\varphi' = \varphi'_2 \varphi'_1$ of u , where φ'_1 is a $\{\text{Spr}, \text{Sdr}\}$ -reduction and φ'_2 is a $\{\text{Snr}\}$ -reduction with $\text{dom}(\varphi'_2) = D$. By Theorem 20, $\mathcal{PC}_{\varphi'_1(u)}$ is a tree. By Theorem 17 $\mathcal{PC}_{\varphi'_1(u)} \approx \mathcal{PC}_u|_D$. ■

Example 13

We continue the example. We saw that $\varphi = \text{snr}_6 \text{snr}_4 \text{snr}_2 \text{spr}_7 \text{sdr}_{5,3}$ is a successful reduction of u . By Theorem 21, $\mathcal{PC}_u|_{\{2,4,6\}}$ is a tree. This is clear from Figure 3.7 where $\mathcal{PC}_u$ is depicted.

In the next few sections we prove the reverse implication of the previous theorem. This will require considerably more effort than the forward implication. The reason for this is that it is not obvious that when $\mathcal{PC}_u|_D$ is a tree, there is a reduction φ_1 of u such that $D = \text{dom}(\varphi_1(u))$. We will use the pointer removal operation to prove this.

First, we consider a special case of the previous theorem. Since a loop can never be part of a tree, we have the following corollary to Theorem 21.

Corollary 22

Let u be a legal string and let $p \in \text{dom}(u)$. If $p \in \text{snrdom}(\varphi)$ for some (successful) reduction φ of u , then $p \in \text{bridge}(u)$.

Example 14

We continue the example. Since $\varphi = \text{snr}_6 \text{snr}_4 \text{snr}_2 \text{spr}_7 \text{sdr}_{5,3}$ is a successful reduction of u , we have $2, 4, 6 \in \text{bridge}(u)$.

We show in Theorem 30 below that the reverse implication of Corollary 22 also holds. Hence, the pointers $p \in \text{bridge}(u)$ are exactly the pointers for which snr_p or $\text{snr}_{\bar{p}}$ can occur in a (successful) reduction of u .

3.7 Merging and Splitting Components

In this section we consider the effect of pointer removal operations on pointer-component graphs. It turns out that these operations correspond to the merging

and splitting of connected components of the underlying reduction graph. We now introduce the merge operation on pointer-component graphs. Intuitively, the p -merge rule ‘merges’ the two endpoints of edge p into one vertex, and therefore the resulting graph has exactly one vertex less than the original graph. Formally the merge operation is as follows.

Definition 23

For each edge p , the p -merge rule, denoted by merge_p , is a rule applicable to (i.e., defined on) $G = (V, E, \epsilon) \in \text{MGr}$ with $p \in E$ and $|\epsilon(p)| = 2$. It is defined by

$$\text{merge}_p(G) = (V', E', \epsilon'),$$

where $E' = E \setminus \{p\}$, $V' = (V \setminus \epsilon(p)) \cup \{v'\}$ with a new vertex $v' \notin V$, and $\epsilon'(e) = \{h(v_1), h(v_2)\}$ iff $\epsilon(e) = \{v_1, v_2\}$ where $h(v) = v'$ if $v \in \epsilon(p)$, otherwise it is the identity. ■

Again, we allow $v_1 = v_2$ in the previous definition. Note that p -merge rules commute under composition. Thus, if $(\text{merge}_q \text{ merge}_p)$ is applicable to G , then

$$(\text{merge}_q \text{ merge}_p)(G) = (\text{merge}_p \text{ merge}_q)(G).$$

Theorem 24

Let $G = (V, E, \epsilon) \in \text{MGr}$, and let $D = \{p_1, \dots, p_n\} \subseteq E$. Then $(\text{merge}_{p_n} \dots \text{merge}_{p_1})$ is applicable to G iff $G|_D$ is acyclic.

Proof

$(\text{merge}_{p_n} \dots \text{merge}_{p_1})$ is applicable on G iff for all p_i ($1 \leq i \leq n$), $\epsilon(p_i) \not\subseteq \epsilon(\{p_1, \dots, p_{i-1}\})$ and $|\epsilon(p_i)| = 2$. Furthermore, the latter holds iff $G|_D$ is acyclic. ■

Surprisingly, the pointer removal operation is crucial in the proofs of the main results. The next theorem compares $\mathcal{PC}_u$ with $\mathcal{PC}_{\text{rem}_{\{p\}}(u)}$ for a legal string u and $p \in \text{dom}(u)$. We distinguish three cases: either the number of vertices of $\mathcal{PC}_{\text{rem}_{\{p\}}(u)}$ is one less, is equal, or is one more than the number of vertices of $\mathcal{PC}_u$. The proof of this theorem shows that the first case corresponds to merging two connected components of $\mathcal{R}_u$ into one connected component, and the last case corresponds to splitting one connected component of $\mathcal{R}_u$ into two connected components.

Theorem 25

Let u be a legal string.

- If $p \in \text{bridge}(u)$, then $\mathcal{PC}_{\text{rem}_{\{p\}}(u)} \approx \text{merge}_p(\mathcal{PC}_u)$
(and therefore $o(\mathcal{PC}_{\text{rem}_{\{p\}}(u)}) = o(\mathcal{PC}_u) - 1$).
- If $p \in \text{dom}(u) \setminus \text{bridge}(u)$, then $o(\mathcal{PC}_u) \leq o(\mathcal{PC}_{\text{rem}_{\{p\}}(u)}) \leq o(\mathcal{PC}_u) + 1$.

Proof

Consider $p \in \text{bridge}(u)$ first. Then the two desire edges with vertices labelled by p belong to different connected components of $\mathcal{R}_u$. We distinguish two cases:

whether or not there are cyclic components consisting of only vertices labelled by p .

If there is cyclic component consisting of only vertices labelled by p , then by Lemma 7, pp or $\bar{p}\bar{p}$ are substrings of u , and $\mathcal{R}_u$ is

$$\begin{array}{c} p \equiv p \\ \text{---} \cup \text{---} \\ \dots \text{---} q_1 \equiv p \text{---} p \equiv q_2 \text{---} \dots \end{array}$$

where we omitted the parts of the graph that are the same compared to $\mathcal{R}_{rem_{\{p\}}(u)}$. Now, $\mathcal{R}_{rem_{\{p\}}(u)}$ is

$$\dots \text{---} q_1 \equiv q_2 \text{---} \dots$$

Therefore $\mathcal{PC}_{rem_{\{p\}}(u)}$ can be obtained (up to isomorphism) from $\mathcal{PC}_u$ by applying the merge_p operation.

Now assume that there are no cyclic components consisting of only vertices labelled by p . Then, $\mathcal{R}_u$ is

$$\begin{array}{c} \dots \text{---} q_1 \equiv p \text{---} p \equiv q_2 \text{---} \dots \\ \\ \dots \text{---} q_3 \equiv p \text{---} p \equiv q_4 \text{---} \dots \end{array}$$

where we again omitted the parts of the graph that are the same compared to $\mathcal{R}_{rem_{\{p\}}(u)}$. Now, depending on the positions of $q_1, \dots, q_4$ relative to p in u and on whether p is positive or negative in u , $\mathcal{R}_{rem_{\{p\}}(u)}$ is either

$$\dots \text{---} q_1 \equiv q_4 \text{---} \dots$$

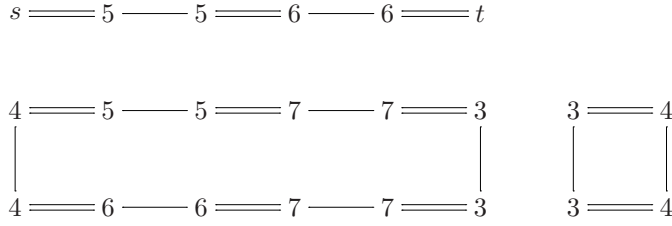
$$\dots \text{---} q_3 \equiv q_2 \text{---} \dots$$

or

$$\dots \text{---} q_1 \equiv q_3 \text{---} \dots$$

$$\dots \text{---} q_4 \equiv q_2 \text{---} \dots$$

Note that since a desire edge connects two ‘segments’ (each represented by a reality edge) corresponding to different occurrences of p or $\bar{p}$, it is not possible that q_1 and q_2 are connected by a reality edge (and also for q_3 and q_4) in $\mathcal{R}_{rem_{\{p\}}(u)}$. Therefore, we have only the above two cases. Since q_1 and q_2 remain part of the same connected component (the same holds for q_3 and q_4), in both cases the two connected components are merged, and thus $\mathcal{PC}_{rem_{\{p\}}(u)}$ can be obtained (up to isomorphism) from $\mathcal{PC}_u$ by applying the merge_p operation.

Figure 3.10: Reduction graph $\mathcal{R}_{rem_{\{2\}}(u)}$ from the Example.

Finally, consider $p \in \text{dom}(u) \setminus \text{bridge}(u)$. Then the two desire edges with vertices labelled by p belong to the same connected component of $\mathcal{R}_u$. By Lemma 7, there are no cyclic components consisting of four vertices which are all labelled by p . We can distinguish two cases: whether or not there is a reality edge e connecting two vertices labelled by p . If there is such a reality edge e then $\mathcal{R}_u$ is

$$\dots \text{---} q_1 == p \text{---} p == p \text{---} p == q_4 \text{---} \dots$$

This occurs precisely when $\bar{p}p$ or $p\bar{p}$ is a substring of u . Now, $\mathcal{R}_{rem_{\{p\}}(u)}$ is

$$\dots \text{---} q_1 == q_4 \text{---} \dots$$

Therefore, $\mathcal{R}_{rem_{\{p\}}(u)}$ has $N = o(\mathcal{PC}_u)$ cyclic components.

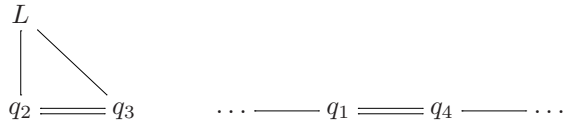
If there is no such a reality edge e , then $\mathcal{R}_u$ is

$$\dots \text{---} q_1 == p \text{---} p == q_2 \text{---} L \text{---} q_3 == p \text{---} p == q_4 \text{---} \dots$$

where L represents some (possibly empty) ‘linear subgraph’ of $\mathcal{R}_u$. Now, $\mathcal{R}_{rem_{\{p\}}(u)}$ is either

$$\dots \text{---} q_4 == q_2 \text{---} L \text{---} q_3 == q_1 \text{---} \dots$$

or



Therefore, $\mathcal{R}_{rem_{\{p\}}(u)}$ has either N cyclic components (corresponding with the first case) or $N + 1$ cyclic components (corresponding with the second case). ■

Example 15

We continue the example. By Theorem 25, we know from Figure 3.7 that $\mathcal{PC}_{rem_{\{2\}}(u)} \approx \text{merge}_2(\mathcal{PC}_u)$, merging components C_1 and C_2 . Indeed, this is transparent from Figures 3.7, 3.10 and 3.11, where $\mathcal{PC}_u$, $\mathcal{R}_{rem_{\{2\}}(u)}$, and $\mathcal{PC}_{rem_{\{2\}}(u)}$ are depicted, respectively.

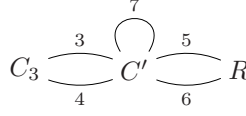


Figure 3.11: $\mathcal{PC}_{\text{rem}_{\{2\}}(u)}$ from the Example.

Again by Theorem 25, we know from Figure 3.10 that $\mathcal{R}_{\text{rem}_{\{2,7\}}(u)}$ has two or three cyclic components. Indeed, this is transparent from Figure 3.5, where $\mathcal{R}_{\text{rem}_{\{2,7\}}(u)}$ is depicted.

Note that by the definition of merge_p , merge_p is applicable to $\mathcal{PC}_u$ precisely when $p \in \text{bridge}(u)$. Therefore, by Theorems 24 and 25, we have the following corollary.

Corollary 26

Let u be a legal string, and let $D \subseteq \text{dom}(u)$. If $\mathcal{PC}_u|_D$ is acyclic, then

$$\mathcal{PC}_{\text{rem}_D(u)} \approx (\text{merge}_{p_n} \cdots \text{merge}_{p_1})(\mathcal{PC}_u),$$

where $D = \{p_1, \dots, p_n\}$.

3.8 Applicability of the String Negative Rule

In this section we characterize for a given set of pointers D , whether or not there is a (successful) strategy that applies string negative rules on exactly these pointers. First we will prove the following result which depends heavily on the results of the previous section. The forward implication of the result states that by removing pointers from u that form a spanning tree in $\mathcal{PC}_u$ we obtain a legal string u' for which the reduction graph does not have cyclic components.

Lemma 27

Let u be a legal string, and let $D \subseteq \text{dom}(u)$. Then $\mathcal{PC}_u|_D$ is a tree iff $\mathcal{R}_{\text{rem}_D(u)}$ and $\mathcal{R}_u$ have 0 and $|D|$ cyclic components, respectively.

Proof

We first prove the forward implication. Let $\mathcal{PC}_u|_D$ be a tree. By Corollary 26, $\mathcal{PC}_{\text{rem}_D(u)}$ contains a single vertex. Thus $\mathcal{R}_{\text{rem}_D(u)}$ has no cyclic components. Since $\mathcal{PC}_u|_D$ is a tree, we have $|D| = |\zeta| - 1$.

We now prove the reverse implication. Let $\mathcal{R}_{\text{rem}_D(u)}$ not contain cyclic components and $|D| = |\zeta| - 1$. By Theorem 25 we see that the removal of each pointer p in D corresponds to a merge_p operation, otherwise $\mathcal{R}_{\text{rem}_D(u)}$ would contain cyclic components. Therefore, $(\text{merge}_{p_n} \cdots \text{merge}_{p_1})$ is applicable to $\mathcal{PC}_u$ with $D = \{p_1, \dots, p_n\}$. Therefore, by Theorem 24, $\mathcal{PC}_u|_D$ is acyclic. Again since $|D| = |\zeta| - 1$, it is a tree. ■

$$C_3 \xrightarrow{3} C_1 \xrightarrow{2} C_2 \xrightarrow{5} R$$

Figure 3.12: A subgraph of the pointer-component graph from the Example.

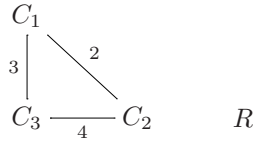


Figure 3.13: A subgraph of the pointer-component graph from the Example.

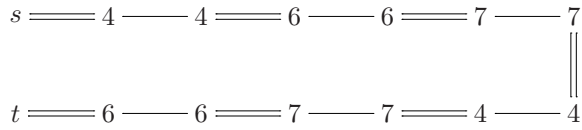


Figure 3.14: The reduction graph $\mathcal{R}_{rem_{D_1}(u)}$ from the Example.

$$s \xrightarrow{5} 5 \xrightarrow{5} 6 \xrightarrow{6} 6 \xrightarrow{6} t$$

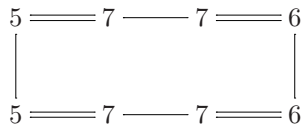


Figure 3.15: The reduction graph $\mathcal{R}_{rem_{D_2}(u)}$ from the Example.

Example 16

We continue the previous example. Let $D_1 = \{2, 3, 5\}$ and $D_2 = \{2, 3, 4\}$. Then $\mathcal{PC}_u|_{D_1}$ ($\mathcal{PC}_u|_{D_2}$, resp.) is given in Figure 3.12 (Figure 3.13, resp.). Notice that $|D_1| = |D_2| = |\zeta| - 1$. Since $\mathcal{PC}_u|_{D_1}$ is a tree and $\mathcal{PC}_u|_{D_2}$ is not a tree, by Lemma 27, it follows that $\mathcal{R}_{rem_{D_1}(u)}$ does not have cyclic components and that $\mathcal{R}_{rem_{D_2}(u)}$ does have at least one cyclic component. This is illustrated in Figures 3.14 and 3.15, where $\mathcal{R}_{rem_{D_1}(u)}$ and $\mathcal{R}_{rem_{D_2}(u)}$ are depicted respectively.

The next theorem is one of the main results of this chapter. It follows directly from Lemma 27 and Lemma 10, and improves Theorem 21 by characterizing exactly which string negative rules can be applied together in a successful reduction of a given legal string.

Theorem 28

Let u be a legal string, and let $D \subseteq \text{dom}(u)$. There is a successful reduction φ of u with $\text{snrdom}(\varphi) = D$ iff $\mathcal{PC}_u|_D$ is a tree.

Since there are many well known and efficient methods for determining spanning trees in a graph, it is easy to determine, for a given set of pointers D , whether or not there is a successful reduction applying string negative rules on exactly the pointers of D (for a given legal string u).

Example 17

We continue the example. By Theorem 28 and Figure 3.12, there is a successful reduction φ of u with $\text{snrdom}(\varphi) = \{2, 3, 5\}$. Indeed, we can take for example $\varphi = \mathbf{snr}_5 \mathbf{snr}_2 \mathbf{snr}_3 \mathbf{spr}_7 \mathbf{sdr}_{4,6}$.

By Theorem 28 (or Theorem 21) and Figure 3.13, there is no successful reduction φ of u with $\text{snrdom}(\varphi) = \{2, 3, 4\}$. For example, $(\mathbf{spr}_5 \mathbf{spr}_7)(u) = 62\bar{3}4\bar{2}346$ and thus there is no string pointer rule for pointer 6 applicable to this legal string.

In the next corollary we consider the more general case $|D| \leq |\zeta| - 1$, instead of $|D| = |\zeta| - 1$ in Theorem 28, i.e., we consider acyclic graphs rather than trees.

Corollary 29

Let u be a legal string, and let $D \subseteq \text{dom}(u)$. There is a (successful) reduction φ of u with $D \subseteq \text{snrdom}(\varphi)$ iff $\mathcal{PC}_u|_D$ is acyclic.

Proof

We first prove the forward implication. By Theorem 28, $\mathcal{PC}_u|_D$ is a subgraph of a tree, and therefore acyclic.

We now prove the reverse implication. By Theorem 19, $\mathcal{PC}_u$ is connected, and since $\mathcal{PC}_u|_D$ does not contain cycles, we can add edges $q \in \text{dom}(u) \setminus D$ from $\mathcal{PC}_u$ such that the resulting graph is a tree. Then by Theorem 28, it follows that there is a (successful) reduction φ of u with $D \subseteq \text{snrdom}(\varphi)$. ■

The previous corollary with $|D| = 1$ shows that the reverse implication of Corollary 22 also holds, since $\mathcal{PC}_u|_{\{p\}}$ acyclic implies that the edge p connects two different vertices in $\mathcal{PC}_u$.

Theorem 30

Let u be a legal string and let $p \in \text{dom}(u)$. Then $p \in \text{snrdom}(\varphi)$ for some (successful) reduction φ of u iff $p \in \text{bridge}(u)$.

This theorem can also be proven directly.

Proof

To prove the reverse implication, let no reduction of u contain either $\mathbf{snr}_p$ or $\mathbf{snr}_{\bar{p}}$. We prove that $p \notin \text{bridge}(u)$. By iteratively applying $\mathbf{snr}$, $\mathbf{spr}$ and $\mathbf{sdr}$ on pointers that are not equal to p or $\bar{p}$, we can reduce u to a legal string v such that for all $q \in \text{dom}(v) \setminus \{p\}$:

- qq and $\bar{q}\bar{q}$ are not substrings of v .
- q is negative in v .
- q does not overlap with any pointer in $\text{dom}(v) \setminus \{p\}$.

If $\text{rem}_{\{p\}}(v) = \lambda$, then v is equal to either $p\bar{p}$, $\bar{p}p$, pp or $\bar{p}\bar{p}$. If $\text{rem}_{\{p\}}(v) \neq \lambda$, then, by the last two conditions, there is a $q \in \Pi$ such that qq is a substring of $\text{rem}_{\{p\}}(v)$. Then, by the first condition, either qpq , $q\bar{p}q$, $qppq$, $q\bar{p}p$, qpq or $q\bar{p}q$ is a substring of v .

Thus, either qpq , $q\bar{p}q$, $p\bar{p}$, $\bar{p}p$, pp or $\bar{p}\bar{p}$ is a substring of v . Since no reduction of u contains $\mathbf{snr}_p$ or $\mathbf{snr}_{\bar{p}}$, the last two cases are not possible. The first two cases correspond to the following part of $\mathcal{R}_v$.

$$\dots \text{===== } p \text{ --- } p \text{ ===== } q \text{ --- } q \text{ ===== } p \text{ --- } p \text{ ===== } \dots$$

The cases where $p\bar{p}$ or $\bar{p}p$ is a substring of v correspond to the following part of $\mathcal{R}_v$

$$\dots \text{===== } p \text{ --- } p \text{ ===== } p \text{ --- } p \text{ ===== } \dots$$

Consequently, in either case, the two desired edges of $\mathcal{R}_v$ with vertices labelled by p belong to the same connected component. Thus $p \notin \text{bridge}(v)$. By Corollary 16, $p \notin \text{bridge}(u)$. ■

3.9 The Order of Loop Recombination

According to Theorem 28 a set D of pointers can occur as the domain of $\mathbf{Snr}$ rules in a successful reduction of a legal string u exactly when the graph $\mathcal{PC}_u|_D$ is a tree. This result can be strengthened to incorporate the order in which the $\mathbf{Snr}$ rules are applied. We show that in a successful reduction φ we can only apply $\mathbf{Snr}$ rules in orderings determined by the tree $\mathcal{PC}_u|_D$ with the linear component as root, where D is the domain of $\mathbf{Snr}$ rules in φ . These orderings are similar topological orderings in a directed acyclic graph, however, here we order the edges instead of the vertices.

Definition 31

Let $T = (V, E, \epsilon)$ be a tree, and let $R \in V$. We define the relation $\prec_{T,R}$ over E as follows. For $e_1, e_2 \in E$, we have $e_1 \prec e_2$ iff $\epsilon(e_1) = \{C_x, C_y\}$, $\epsilon(e_2) = \{C_y, C_z\}$, and C_y (C_z , resp.) is the father of C_x (C_y , resp.) in T considering R as the root of T . Also, an *edge-topological ordering* of T (with root R) is a topological ordering of $\prec_{T,R}$. ■

Example 18

We continue the example. Consider again tree $\mathcal{PC}_u|_{D_1}$ shown in Figure 3.12. Taking R as the root of $\mathcal{PC}_u|_{D_1}$, it follows that $(3, 2, 5)$ is an edge-topological ordering of $\mathcal{PC}_u|_{D_1}$.

The next theorem characterizes exactly the possible orderings in which string negative rules can be applied in a successful reduction of a given legal string.

Theorem 32

Let u be a legal string, let L be a linear ordering of a subset L' of $\text{dom}(u)$. There is a successful reduction φ of u with *Snr*-order L iff $\mathcal{PC}_u|_{L'}$ is a tree, where L is an edge-topological ordering of $\mathcal{PC}_u|_{L'}$ with the linear component R of $\mathcal{R}_u$ as root.

Proof

Let $L = (p_1, p_2, \dots, p_n)$. We first prove the forward implication. Recall that we can postpone the application of string negative rules, thus $\mathbf{snr}_{\tilde{p}_n} \mathbf{snr}_{\tilde{p}_{n-1}} \dots \mathbf{snr}_{\tilde{p}_1} \varphi'$ is also a successful reduction of u , where φ' is a $\{Spr, Sdr\}$ -reduction and $\tilde{p}_i \in \{p_i, \bar{p}_i\}$ for $i \in \{1, \dots, n\}$. By Theorem 28, $\mathcal{PC}_u|_{L'}$ is a tree.

We prove that L is an edge-topological ordering of $\mathcal{PC}_u|_{L'}$ with root R . By Theorem 17, $\mathcal{PC}_{\varphi'(u)} \approx \mathcal{PC}_u|_{L'}$. If $n > 0$, then $\mathbf{snr}_{\tilde{p}_1}$ is applicable to $\varphi'(u)$. By Lemma 18, edge p_1 is connected to a leaf of $\mathcal{PC}_{\varphi'(u)}$. By Theorem 14 and the paragraph below Lemma 18, $\mathcal{PC}_{(\mathbf{snr}_{\tilde{p}_1} \varphi')(u)}$ is isomorphic to the graph obtained from $\mathcal{PC}_{\varphi'(u)}$ by removing p_1 and its leaf. Now (assuming $n > 1$), since $\mathbf{snr}_{\tilde{p}_2}$ is applicable to $(\mathbf{snr}_{\tilde{p}_1} \varphi')(u)$, p_2 is connected to a leaf in $\mathcal{PC}_{(\mathbf{snr}_{\tilde{p}_1} \varphi')(u)}$. By iterating this argument, it follows that L is an edge-topological ordering of $\mathcal{PC}_{\varphi'(u)} \approx \mathcal{PC}_u|_{L'}$ with root R .

We now prove the reverse implication. Since $\mathcal{PC}_u|_{L'}$ is a tree, by Theorem 28 there is a successful reduction $\varphi = \varphi_2 \varphi_1$ of u , where φ_1 is a $\{Spr, Sdr\}$ -reduction and φ_2 is a $\{Snr\}$ -reduction with $\text{dom}(\varphi_2) = L'$. Let L be an edge-topological ordering of $\mathcal{PC}_u|_{L'}$ with the linear component R of $\mathcal{R}_u$ as root. Again, by Theorem 17, $\mathcal{PC}_u|_{L'} \approx \mathcal{PC}_{\varphi_1(u)}$.

If $n > 0$, then p_1 is connected to a leaf of $\mathcal{PC}_{\varphi_1(u)}$. By Lemma 18, $\mathbf{snr}_{\tilde{p}_1}$ is applicable to $\varphi_1(u)$ for some $\tilde{p}_1 \in \{p_1, \bar{p}_1\}$. Again by Theorem 14 and the paragraph below Lemma 18, $\mathcal{PC}_{(\mathbf{snr}_{\tilde{p}_1} \varphi_1)(u)}$ is isomorphic to the graph obtained from $\mathcal{PC}_{\varphi_1(u)}$ by removing p_1 and its leaf. By iterating this argument, it follows that $\mathbf{snr}_{\tilde{p}_n} \mathbf{snr}_{\tilde{p}_{n-1}} \dots \mathbf{snr}_{\tilde{p}_1}$ is a successful reduction of u for some $\tilde{p}_i \in \{p_i, \bar{p}_i\}$ and $1 \leq i \leq n$ with $n \geq 0$. ■

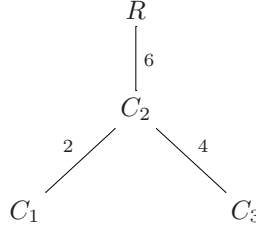


Figure 3.16: A subgraph of the pointer-component graph from the Example.

Example 19

We continue the example. Since $(3, 2, 5)$ is an edge-topological ordering of tree $\mathcal{PC}_u|_{D_1}$ with root R , by Theorem 32, there is a successful reduction φ of u with Snr -order $(3, 2, 5)$. Indeed, we can take for example $\varphi = \mathbf{snr}_5 \mathbf{snr}_2 \mathbf{snr}_3 \mathbf{spr}_7 \mathbf{sdr}_{4,6}$.

We say that two reduction rules ρ_1 and ρ_2 can be applied in *parallel* to u if both $\rho_2 \rho_1$ and $\rho_1 \rho_2$ are applicable to u (see [18]).

Corollary 33

Let u be a legal string, and $p, q \subseteq \text{dom}(u)$ with $p \neq q$. Then $\mathbf{snr}_{\tilde{p}}$ and $\mathbf{snr}_{\tilde{q}}$ can be applied in parallel to u for some $\tilde{p} \in \{p, \bar{p}\}$, $\tilde{q} \in \{q, \bar{q}\}$ iff there is a spanning tree T in $\mathcal{PC}_u$ such that p and q both connect to leaves (considering the linear component of $\mathcal{R}_u$ as the root).

Let R be the linear component of $\mathcal{R}_u$. Clearly, for spanning tree T in $\mathcal{PC}_u$ with root R that contains edges p and q , we have the following: p and q in T are independent for $\prec_{T,R}$ iff there is no simple path in T from R to another vertex of T containing both edges p and q . The next corollary considers the case whether or not $\mathbf{snr}_{\tilde{p}}$ and $\mathbf{snr}_{\tilde{q}}$ can *eventually* be applied in parallel.

Corollary 34

Let u be a legal string, and $p, q \subseteq \text{dom}(u)$ with $p \neq q$. Then $\mathbf{snr}_{\tilde{p}}$ and $\mathbf{snr}_{\tilde{q}}$ can be applied in parallel to $\varphi(u)$ for some $\tilde{p} \in \{p, \bar{p}\}$, $\tilde{q} \in \{q, \bar{q}\}$, and some reduction φ of u iff there is a spanning tree T in $\mathcal{PC}_u$ containing both edges p and q , where p and q are independent for $\prec_{T,R}$.

Example 20

We continue the example. Let $D_3 = \{2, 4, 6\}$. Then in the tree $\mathcal{PC}_u|_{D_3}$, depicted in Figure 3.16, there is no simple path from R to another vertex of $\mathcal{PC}_u|_{D_3}$ containing both edges 2 and 4. By Corollary 34, $\mathbf{snr}_{\tilde{2}}$ and $\mathbf{snr}_{\tilde{4}}$ can be applied in parallel to $\varphi(u)$ for some $\tilde{2} \in \{2, \bar{2}\}$, $\tilde{4} \in \{4, \bar{4}\}$, and some reduction φ of u . Indeed, if we take $\varphi = \mathbf{spr}_7 \mathbf{sdr}_{3,5}$, then $\mathbf{snr}_2$ and $\mathbf{snr}_4$ can be applied in parallel to $\varphi(u) = 622446$.

3.10 Conclusion

This chapter shows that one can efficiently determine the possible sequences of loop recombination operations that can be applied in the transformation of a given gene from its micronuclear to its macronuclear form. Formally, one can determine the orderings of string negative rules that can be present in successful reductions of u . This is a characterization in terms of a graph defined on the (components of the) reduction graph. Future research could focus on similar characterizations for the string positive rules and the string double rules. However, this would require other concepts, since the pointer-component graph does not retain information regarding positiveness or overlap of pointers, notions crucial for the applicability of the other two operations.

Chapter 4

The Fibers and Range of Reduction Graphs

Abstract

The biological process of gene assembly transforms a nucleus (the MIC) into a functionally and physically different nucleus (the MAC). For each gene in the MIC (the input), recombination operations transform the gene to its MAC form (the output). Here we characterize which inputs obtain the same output, and moreover characterize the possible forms of the outputs. We do this in the abstract and more general setting of so-called legal strings.

4.1 Introduction

Ciliates form a large group of one-cellular organisms that are able to transform one nucleus, called the micronucleus (MIC), into an astonishing different one, called the macronucleus (MAC). This intricate DNA transformation process is called *gene assembly*. Each gene occurs both in the MIC and MAC, but in very different forms. During gene assembly each gene is transformed from its MIC form to its MAC form.

Formally, the gene in MIC form (the input) can be described by a so-called legal string [12], while the gene in MAC form, including additionally generated structures, (the output) can be described by a so-called reduction graph [6, 5]. The reduction graph is based on the notion of breakpoint graph in the theory of sorting by reversal [17, 1, 23].

Given the function $\mathcal{R}$ that assigns to each legal string u its reduction graph $\mathcal{R}_u$, we (1) characterize the range of $\mathcal{R}$ (up to graph isomorphism) in terms of easy-to-check conditions on graphs (cf. Theorem 24), and (2) characterize the fiber $\mathcal{R}^{-1}(\mathcal{R}_u)$ (modulo graph isomorphism) for each reduction graph $\mathcal{R}_u$ (cf. Theorem 34). In fact we show that $\mathcal{R}^{-1}(\mathcal{R}_u)$ is the ‘orbit’ of u under two types

of string rewriting rules.

Result (1) characterizes which graphs are (isomorphic to) reduction graphs. Obviously, these graphs should have the ‘look and feel’ of reduction graphs. For instance, each vertex label should occur exactly four times, and the second type of edges connect vertices of the same label. Once these elementary and easy-to-check properties are satisfied, reduction graphs are characterized as having a connected pointer-component graph — a graph which represents the distribution of the vertex labels over the connected components, originally defined in [4]. This last condition can also be efficiently verified. The characterization implies restrictions on the form of the MAC structures that can possibly occur.

Result (2) determines, given two legal strings, whether or not they have the same reduction graph. This may allow one to determine which MIC genes obtain the same MAC structure. It turns out that two legal strings obtain the same reduction graph (up to isomorphism) exactly when they can be transformed into each other by two types of string rewriting rules. We will see that, surprisingly, these rules are in a sense dual to string rewriting rules in a model of gene assembly called string pointer reduction system (SPRS) [12].

The latter characterization has additional uses for the specific model SPRS as well. In this model, gene assembly is assumed to be performed by three types of recombination (splicing) operations that are modeled as types of string rewriting rules. The string negative rules form one of these types. It has been shown that the reduction graph allows for a complete characterization of applicability of the string negative rules during the transformation process [6, 4]. Moreover, it has been shown that the reduction graph does not retain much information about the applicability of the other two types of rules [4]. Therefore, the legal strings that obtain the same reduction graph are exactly the legal strings that have similar characteristics concerning the string negative rule.

To establish both main results, we augment the (abstract) reduction graph with a set of *merge-legal edges*. We will show that some “valid” sets of merge-legal edges for a reduction graph allows one to “go back” to a legal string corresponding to this (abstract) reduction graph. In this way the existence of such valid set determines which graphs are (isomorphic to) reduction graphs. The first main result shows that the existence of such valid set is computationally easy to verify. Moreover, the set of all sets of merge-legal edges can be transformed into each other by *flip operations*. These flip operations can be defined in terms of the above mentioned dual string pointer rules on legal strings. This will establish the other main result.

This chapter is organized as follows. Section 4.2 fixes notation of basic mathematical notions. In Section 4.3 we recall notions related to legal strings, in Section 4.4 we recall the reduction graph and the pointer-component graph, and in Section 4.5 we generalize the notion of reduction graph and give an extension through merge-legal edges. In Section 4.6 we provide a preliminary characterization that determines which graphs are (isomorphic to) reduction graphs. In the next three sections, we strengthen the result to allow for efficient algorithms: in

Section 4.7 we define the flip operation on sets of merge-legal edges, in Section 4.8 we show that the effect of flip operation corresponds to merging or splitting of connected components, and in Section 4.9 we prove the first main result, cf. Theorem 24. In Sections 4.10 and 4.11 we prove the second main result, cf. Theorem 34. We conclude this chapter with a discussion. A conference edition of this chapter, containing selected results without proofs, was presented at DLT '07 [2].

4.2 Mathematical Notation and Terminology

In this section we recall some basic notions concerning functions, strings, and graphs. We do this mainly to fix the basic notation and terminology.

The symmetric difference of sets X and Y , $(X \setminus Y) \cup (Y \setminus X)$, is denoted by $X \oplus Y$. As $\oplus$ is associative, one may define the symmetric difference of a finite family of sets $(X_i)_{i \in A}$ – it is denoted by $\bigoplus_{i \in A} X_i$. The *composition* of functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ is the function $gf : X \rightarrow Z$ such that $(gf)(x) = g(f(x))$ for every $x \in X$. The restriction of f to a subset A of X is denoted by $f|_A$. The range $f(X)$ of f will be denoted by $\text{rng}(f)$. The *fiber* (or *preimage*) of $y \in Y$ under f , denoted by $f^{-1}(y)$, is $\{x \in X \mid f(x) = y\}$. The fibers form a partition of X . If $Y = X$, then f is called *self-inverse* if f^2 is the identity function. We will use λ to denote the empty string.

We now turn to graphs. A (*undirected*) *graph* is a tuple $G = (V, E)$, where V is a finite set and $E \subseteq \{\{x, y\} \mid x, y \in V\}$. The elements of V are the *vertices* of G and the elements of E are the *edges* of G . In this chapter we allow $x = y$, and therefore edges can be of the form $\{x, x\} = \{x\}$ — an edge of this form should be seen as an edge connecting x to x , i.e., a ‘loop’ for x . The *restriction* of G to $E' \subseteq E$, denoted by $G|_{E'}$, is the subgraph (V, E') . The *order* $|V|$ of G is denoted by $o(G)$.

A *multigraph* is a (undirected) graph $G = (V, E, \epsilon)$, where parallel edges are possible. Therefore, E is a finite set of edges and $\epsilon : E \rightarrow \{\{x, y\} \mid x, y \in V\}$ is the *endpoint mapping*. Note that for multigraphs, E is not specified in terms of V — the relationship between V and E is specified by ϵ .

A *coloured base* B is a 4-tuple (V, f, s, t) such that V is a finite set, $s, t \in V$, and $f : V \setminus \{s, t\} \rightarrow \Gamma$ for some Γ . The elements of V , $\{\{x, y\} \mid x, y \in V, x \neq y\}$, and Γ are called *vertices*, *edges*, and *vertex labels* for B , respectively.

An *n -edge coloured graph*, $n \geq 1$, is a tuple $G = (V, E_1, E_2, \dots, E_n, f, s, t)$ where $B = (V, f, s, t)$ is a coloured base and, for $i \in \{1, \dots, n\}$, E_i is a set of edges for B . We also denote G by $B(E_1, E_2, \dots, E_n)$. We define $\text{dom}(G) = \text{rng}(f)$.

The previously defined notions and notation for graphs carry over to multigraphs and n -edge coloured graphs. Isomorphisms between graphs are defined in the usual way: graphs are considered *isomorphic* when they are equal modulo the identity of the vertices. However, the labels of the identified vertices in n -edge coloured graphs must be equal. Therefore n -edge coloured graphs $G = (V, E_1, \dots, E_n, f, s, t)$ and $G' = (V', E'_1, \dots, E'_n, f', s', t')$ are *isomorphic*, denoted by $G \approx G'$, if there is a bijection $q : V \rightarrow V'$ such that $q(s) = s'$, $q(t) = t'$,

$f(v) = f'(q(v))$ for all $v \in V$, and $\{x, y\} \in E_i$ iff $\{q(x), q(y)\} \in E'_i$, for all $x, y \in V$, and $i \in \{1, \dots, n\}$. Also, multigraphs $G = (V, E, \epsilon)$ and $G' = (V', E, \epsilon')$ are *isomorphic*, denoted by $G \approx G'$, if there is a bijection $\alpha : V \rightarrow V'$ such that $\alpha\epsilon = \epsilon'$, or more precisely, for $e \in E$, $\epsilon(e) = \{v_1, v_2\}$ implies $\epsilon'(e) = \{\alpha(v_1), \alpha(v_2)\}$. We assume the reader is familiar with the notions of *cycle* and *connected component* in a graph. A graph is called *connected* if it has exactly one connected component, and it is called *acyclic* when it does not contain cycles.

4.3 Legal strings

Gene assembly transforms each gene from its MIC form to its MAC form. Formally, the MIC form of a gene (the input) is represented by a legal string u , while the MAC form of that gene, including the additionally generated structures, (the output) is represented by the reduction graph of u . We define the notion of legal string and some accompanying notions in this section, and the notion of reduction graph in the next section. We refer to [12] for a detailed motivation of the notions of this section.

We fix $\kappa \geq 2$, and define the alphabet $\Delta = \{2, 3, \dots, \kappa\}$. For $D \subseteq \Delta$, we define $\bar{D} = \{\bar{a} \mid a \in D\}$ and $\Pi = \Delta \cup \bar{\Delta}$. The elements of Π will be called *pointers*. We use the ‘bar operator’ to move from Δ to $\bar{\Delta}$ and back from $\bar{\Delta}$ to Δ . Hence, for $p \in \Pi$, $\bar{\bar{p}} = p$. For a string $u = x_1 x_2 \dots x_n$ with $x_i \in \Pi$, the *inverse* of u is the string $\bar{u} = \bar{x}_n \bar{x}_{n-1} \dots \bar{x}_1$. For $p \in \Pi$, we define $\mathbf{p} = \begin{cases} p & \text{if } p \in \Delta \\ \bar{p} & \text{if } p \in \bar{\Delta} \end{cases}$, i.e., $\mathbf{p}$ is the ‘unbarred’

variant of p . The *domain* of a string $v \in \Pi^*$ is $\text{dom}(v) = \{\mathbf{p} \mid p \text{ occurs in } v\}$. A *legal string* is a string $u \in \Pi^*$ such that for each $p \in \Pi$ that occurs in u , u contains exactly two occurrences from $\{p, \bar{p}\}$. For a pointer p and a legal string u , if both p and $\bar{p}$ occur in u then we say that both p and $\bar{p}$ are *positive* in u ; if on the other hand only p or only $\bar{p}$ occurs in u , then both p and $\bar{p}$ are *negative* in u .

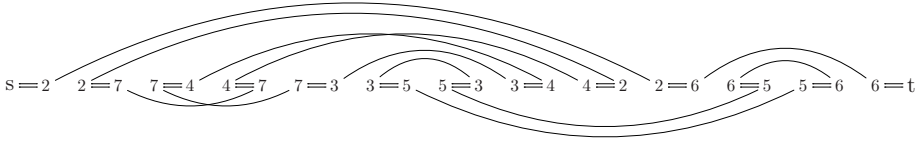
Let $u = x_1 x_2 \dots x_n$ be a legal string with $x_i \in \Pi$ for $1 \leq i \leq n$. For a pointer $p \in \Pi$ such that $\{x_i, x_j\} \subseteq \{p, \bar{p}\}$ and $1 \leq i < j \leq n$, the *p-interval* of u is the substring $x_i x_{i+1} \dots x_j$. Two distinct pointers $p, q \in \Pi$ *overlap* in u if both $\mathbf{q} \in \text{dom}(I_p)$ and $\mathbf{p} \in \text{dom}(I_q)$, where I_p (I_q , resp.) is the *p-interval* (*q-interval*, resp.) of u .

We say that legal strings u and v are *equivalent*, denoted by $u \approx v$, if there is homomorphism $\varphi : \Pi^* \rightarrow \Pi^*$ with $\varphi(p) \in \{p, \bar{p}\}$ and $\varphi(\bar{p}) = \overline{\varphi(p)}$ for all $p \in \Pi$ such that $\varphi(u) = v$.

Example 1

Legal strings $2\bar{2}33$ and $\bar{2}233$ are equivalent, while $\bar{2}\bar{2}33$ and $2\bar{2}\bar{3}3$ are not.

Note that $\approx$ is an equivalence relation. Equivalent legal strings are characterized by their ‘unbarred version’ and their set of positive pointers.

Figure 4.1: The reduction graph $\mathcal{R}_u$ of u in Example 2.

4.4 Reduction Graph

We now recall the definition of reduction graph. This definition is equal to the one in [4], and is in slightly less general form compared to the one in [6]. We refer to [6], where it was introduced, for a more detailed motivation and for more examples and results. The notion of reduction graph uses the intuition from the notion of breakpoint graph (or reality-and-desire diagram) known from another branch of DNA processing theory called sorting by reversal, see e.g. [23, 21]. From a biological point of view, the reduction graph represents the MAC form of a gene (including the additionally generated structures) given its MIC form. As the MIC form of a gene is represented by a legal string, reduction graphs are defined on legal strings.

Definition 1

Let $u = p_1 p_2 \cdots p_n$ with $p_1, \dots, p_n \in \Pi$ be a legal string. The *reduction graph* of u , denoted by $\mathcal{R}_u$, is a 2-edge coloured graph (V, E_1, E_2, f, s, t) , where

$$V = \{I_1, I_2, \dots, I_n\} \cup \{I'_1, I'_2, \dots, I'_n\} \cup \{s, t\},$$

$$E_1 = \{e_0, e_1, \dots, e_n\} \text{ with } e_i = \{I'_i, I_{i+1}\} \text{ for } 0 < i < n, e_0 = \{s, I_1\}, e_n = \{I'_n, t\},$$

$$E_2 = \{ \{I'_i, I_j\}, \{I_i, I'_j\} \mid i, j \in \{1, 2, \dots, n\} \text{ with } i \neq j \text{ and } p_i = p_j \} \cup \\ \{ \{I_i, I_j\}, \{I'_i, I'_j\} \mid i, j \in \{1, 2, \dots, n\} \text{ and } p_i = \bar{p}_j \}, \text{ and}$$

$$f(I_i) = f(I'_i) = \mathbf{p}_i \text{ for } 1 \leq i \leq n.$$

■

The edges of E_1 are called the *reality edges*, and the edges of E_2 are called the *desire edges*. Notice that for each $p \in \text{dom}(u)$, the reduction graph of u has exactly two desire edges containing vertices labelled by p . It follows from the construction of the reduction graph that, given legal strings u and v , $u \approx v$ iff $\mathcal{R}_u = \mathcal{R}_v$.

In depictions of reduction graphs, we will represent the vertices (except for s and t) by their labels, because the exact identity of the vertices is not essential for the problems considered in this chapter. We will also depict reality edges as ‘double edges’ to distinguish them from the desire edges.

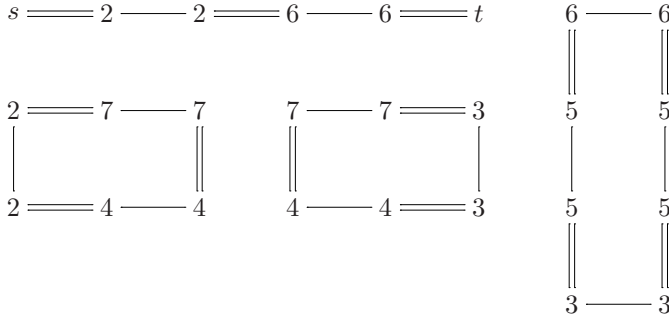


Figure 4.2: The reduction graph of Figure 4.1 obtained by rearranging the vertices.

Example 2

The reduction graph of $u = 2\bar{7}47353\bar{4}2656$ is depicted in Figure 4.1. Note how positive pointers are connected by crossing desire edges, while those for negative pointers are parallel. By rearranging the vertices we can depict the graph as shown in Figure 4.2.

Reality edges follow the linear order of the legal string, whereas desire edges connect positions in the string that will be joined when performing reduction rules, see [6].

We now recall the definition of pointer-component graph of a legal string, introduced in [4]. The graph represents how the labels of a reduction graph are distributed among its connected components. Surprisingly, this graph has different uses in this chapter compared to its original uses in [4]. There it was used in a specific model of gene assembly (which we do not assume here) to characterize a type of splicing operation called loop recombination.

Definition 2

Let u be a legal string. The *pointer-component graph* of u (or of $\mathcal{R}_u$), denoted by $\mathcal{PC}_u$, is a multigraph (ζ, E, ϵ) , where ζ is the set of connected components of $\mathcal{R}_u$, $E = \text{dom}(u)$ and ϵ is, for $e \in E$, defined by $\epsilon(e) = \{C \in \zeta \mid C \text{ contains vertices labelled by } e\}$. ■

Since for each $e \in \text{dom}(u)$, there are exactly two desire edges connecting vertices labelled by e , $1 \leq |\epsilon(e)| \leq 2$, and therefore ϵ is well defined (recall that the case $|\epsilon(e)| = 1$ corresponds to a loop).

Example 3

The pointer-component graph of the reduction graph from Figure 4.2 is shown in Figure 4.3.

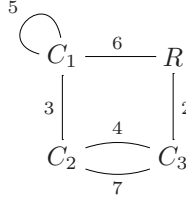


Figure 4.3: The pointer-component graph of the reduction graph from Figure 4.2.

4.5 Abstract Reduction Graphs and Extensions

In this section we generalize the notion of reduction graph as a starting point to consider which graphs are (isomorphic to) reduction graphs. Moreover, we extend the reduction graphs by a set of edges, called *merge edges*, such that, along with the reality edges, the linear structure of the legal string is preserved in the graph.

We will now define a set of edges for a given coloured base which has features in common with desire edges of a reduction graph.

Definition 3

Let $B = (V, f, s, t)$ be a coloured base. We say that a set of edges E for B is *desirable* if

1. for all $\{v_1, v_2\} \in E$, $f(v_1) = f(v_2)$,
2. for each $v \in V \setminus \{s, t\}$ there is exactly one $e \in E$ such that $v \in e$. ■

We now generalize the concept of reduction graph.

Definition 4

A 2-edge coloured graph $B(E_1, E_2)$ with $B = (V, f, s, t)$ is called an *abstract reduction graph* if

1. $\text{rng}(f) \subseteq \Delta$, and for each $p \in \text{rng}(f)$, $|f^{-1}(p)| = 4$,
2. for each $v \in V$ there is exactly one $e \in E_1$ such that $v \in e$,
3. E_2 is desirable for B . ■

The set of all abstract reduction graphs is denoted by $\mathcal{G}$.

Clearly, if $G \approx \mathcal{R}_u$ for some u , then $G \in \mathcal{G}$. Therefore, for abstract reduction graphs $G = B(E_1, E_2)$, the edges in E_1 are called *reality edges* and the edges in E_2 are called *desire edges*. For graphical depictions of abstract reduction graphs we will use the same conventions as we have for reduction graphs. Thus, edges in E_1 will be depicted as “double edges”, vertices are represented by their label, etc.

Example 4

The 2-edge coloured graph in Figure 4.4 is an abstract reduction graph.

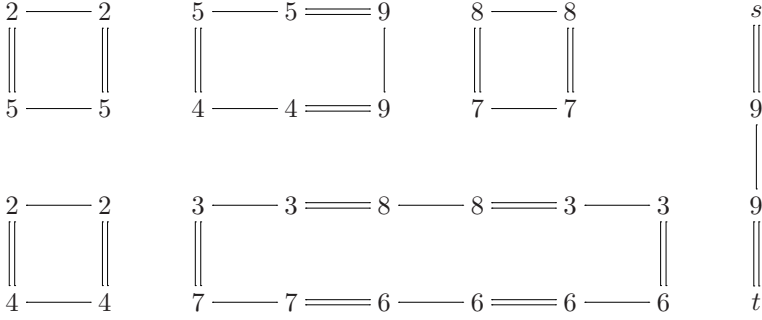
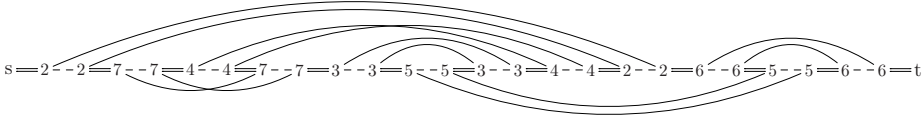


Figure 4.4: An abstract reduction graph.

Figure 4.5: The extended reduction graph $\mathcal{E}_u$ of u given in Example 2.

Note that conditions (1) and (3) in the previous definition imply that for each $p \in \text{rng}(f)$, there is a partition $\{e_1, e_2\}$ of $f^{-1}(p)$, denoted by $C_{G,p}$ or C_p when G is clear from the context, such that $e_1, e_2 \in E_2$.

We now introduce an extension to reduction graphs such that the ‘generic’ linear order of the vertices $s, I_1, I'_1, \dots, I_n, I'_n, t$ is retained, even when we consider the graphs up to isomorphism.

Definition 5

Let u be a legal string. The *extended reduction graph* of u , denoted by $\mathcal{E}_u$, is a 3-edge coloured graph $B(E_1, E_2, E_3)$, where $\mathcal{R}_u = B(E_1, E_2)$ and $E_3 = \{\{I_i, I'_i\} \mid 1 \leq i \leq n\}$ with $n = |u|$. ■

The edges in E_3 are called the *merge edges* of u , denoted by M_u . In this way, the reality edges and the merge edges form a unique path which passes through the vertices in the generic linear order. This is illustrated in the next example. In figures merge edges will be depicted by “dashed edges”.

Example 5

The extended reduction graph $\mathcal{E}_u$ of u given in Example 2 is shown in Figure 4.5, cf. Figure 4.1.

Remark

The notion of merge edges for (extended) reduction graphs is more closely related to the notion of reality edges for breakpoint graphs in the theory of sorting by reversal [17] compared to the notion of reality edges for (extended) reduction graphs. Thus in a way it would be more natural to call the merge edges reality

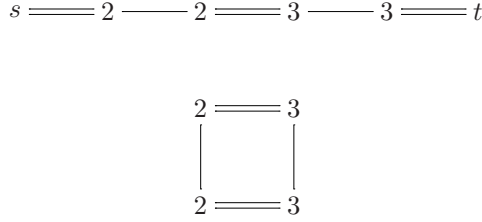


Figure 4.6: An abstract reduction graph.

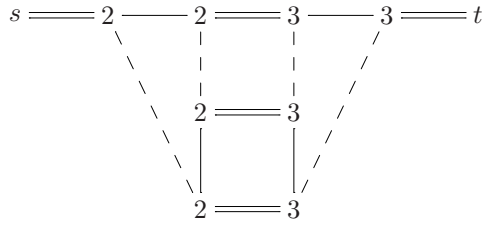


Figure 4.7: The abstract reduction graph of Figure 4.6 with a set of merge-legal edges.

edges for (extended) reduction graphs, and the other way around. However, to avoid confusion with earlier work, we do not change this terminology.

We now generalize this extension of reduction graphs to abstract reduction graphs.

Definition 6

Let $G = B(E_1, E_2) \in \mathcal{G}$, and let E be a set of edges for B . We say that E is *merge-legal* for G if E is desirable for B , and $E_2 \cap E = \emptyset$. We denote the set $\{E \mid E \text{ merge-legal for } G\}$ by ML_G . The set of all $E \in \text{ML}_G$ where $B(E_1, E)$ is connected is denoted by CON_G . ■

For legal string u , we also denote $\text{ML}_{\mathcal{R}_u}$ and $\text{CON}_{\mathcal{R}_u}$ by ML_u and CON_u , respectively. Notice that $M_u \in \text{CON}_u \subseteq \text{ML}_u$. Therefore, merge-legal edges will also be depicted by “dashed edges”.

Example 6

Let us consider the abstract reduction graph $G = B(E_1, E_2)$ of Figure 4.6. This graph is again depicted in Figure 4.7 including a merge-legal set E for G . In this way Figure 4.7 depicts the 3-edge coloured graph $B(E_1, E_2, E)$. Notice that $E \notin \text{CON}_G$. In Figure 4.8, the abstract reduction graph is depicted with a merge-legal set in CON_G .

We now define a natural abstraction of the notion of extended reduction graph.

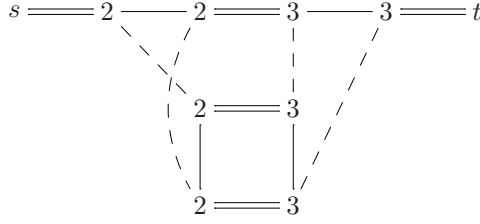


Figure 4.8: The abstract reduction graph of Figure 4.6 with another set of merge-legal edges.

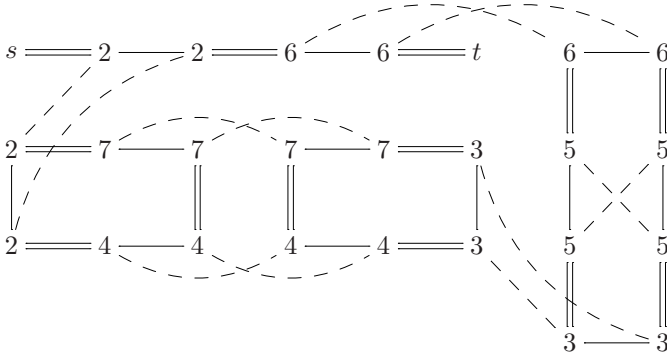


Figure 4.9: A extended abstract reduction graph obtained by augmenting the reduction graph of Figure 4.2 with merge edges.

Definition 7

Let $G = B(E_1, E_2) \in \mathcal{G}$ and $E \in \text{CON}_G$. Then $G' = B(E_1, E_2, E)$ is called a *extended abstract reduction graph*. ■

For each legal string u , $\mathcal{E}_u$ is an extended abstract reduction graph, since $M_u \in \text{CON}_u$. Therefore, the edges in E (in the previous definition) are called the *merge edges (of G')*. Since $E \in \text{CON}_G$, $B(E_1, E)$ has the following form:

$$s \equiv \mathbf{p}_1 - - - \mathbf{p}_1 \equiv \mathbf{p}_2 - - - \mathbf{p}_2 \equiv \cdots \equiv \mathbf{p}_n - - - \mathbf{p}_n \equiv t$$

Thus the property that reality and merge edges in an extended reduction graph induce a unique path from s to t that alternately passes through reality edges and merge edges is retained for extended abstract reduction graphs G in general.

Example 7

If we consider the reduction graph $\mathcal{R}_u = B(E_1, E_2)$ of Example 2 shown in Figure 4.2, then, of course, $B(E_1, E_2, M_u) = \mathcal{E}_u$ shown in Figure 4.5 is an extended abstract reduction graph. In Figure 4.9 another extended reduction graph is shown – it is $\mathcal{R}_u$ augmented with a set of merge edges E in CON_u . It is easy to see that indeed $E \in \text{CON}_u$: simply notice that the path from s to t induced by the reality and merge edges will go through every vertex of the graph.

4.6 Back to Legal Strings

In this section we show that for extended abstract reduction graphs G we can ‘go back’ in the sense that there are legal strings u such that G is isomorphic to $\mathcal{E}_u$. Moreover we show how to obtain the set L_G of all legal strings that corresponds to G . We will show that the legal strings in L_G are equivalent, and thus that extended reduction graphs retain all essential information of the legal strings.

As extended abstract reduction graphs have a natural linear order of the vertices given by their reality edges and merge edges, we can infer whether or not desire edges ‘cross’ or not – thereby providing a way to define negative and positive pointers for extended abstract reduction graphs. This is formalized as follows.

Definition 8

Let $G = B(E_1, E_2, E_3)$ be an extended abstract reduction graph, let $G' = B(E_1, E_2)$, and let $\pi = (s, v_1, v'_1, \dots, v_n, v'_n, t)$ be the path from s to t in $B(E_1, E_3)$. We say that $p \in \text{dom}(G)$ is *negative in G* iff $C_{G',p} = \{\{v_i, v'_j\}, \{v'_i, v_j\}\}$ for some $i, j \in \{1, \dots, n\}$ with $i \neq j$. Also, we say that $p \in \text{dom}(G)$ is *positive in G* if p is not negative in G . ■

Clearly, $p \in \text{dom}(G)$ is positive in G iff $C_{G',p} = \{\{v_i, v_j\}, \{v'_i, v'_j\}\}$ for some $i, j \in \{1, \dots, n\}$ with $i \neq j$. It is easy to see that p is negative in legal string u iff p is negative in $\mathcal{E}_u$.

Next, we assign to each extended abstract reduction graph G a set of legal strings L_G . We subsequently show that these strings are precisely the legal strings u such that $\mathcal{E}_u \approx G$.

Definition 9

Let $G = B(E_1, E_2, E_3)$ be an extended abstract reduction graph, and let $H = B(E_1, E_3)$ be as follows:

$$s \text{ --- } \mathbf{p}_1 \text{ --- } \mathbf{p}_1 \text{ --- } \mathbf{p}_2 \text{ --- } \mathbf{p}_2 \text{ --- } \dots \text{ --- } \mathbf{p}_n \text{ --- } \mathbf{p}_n \text{ --- } t$$

The *legalization* of G , denoted by L_G , is the set of legal strings $u = p_1 p_2 \dots p_n$ with $p_i \in \{\mathbf{p}_i, \bar{\mathbf{p}}_i\}$ and p_i is negative in u iff p_i is negative in G . ■

Example 8

Let us consider the extended abstract reduction graph G of Figure 4.9. By rearranging the vertices we obtain Figure 4.10. From this figure it is clear that $v = 274265374356 \in L_G$.

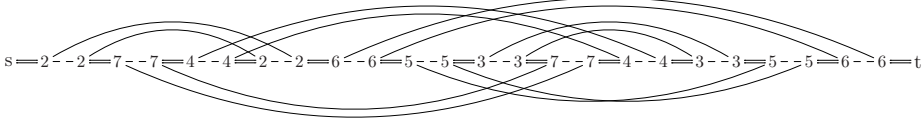


Figure 4.10: The extended abstract reduction graph G given in Example 8.

It is easy to see that, for a legal string u , we have $u \in L_{\mathcal{E}_u}$.

Note that L_G , for extended abstract reduction graph G , is a non-empty equivalence class w.r.t. to the $\approx$ relation (for legal strings). Since the definition of L_G does not depend on the exact identity of the vertices of G , we have, for extended abstract reduction graphs G and G' , $G \approx G'$ implies $L_G = L_{G'}$.

Theorem 10

1. Let G and G' be extended abstract reduction graphs. Then $G \approx G'$ iff $L_G = L_{G'}$.
2. Let u and v be legal strings. Then $u \approx v$ iff $\mathcal{E}_u \approx \mathcal{E}_v$.

Proof

We first consider statement 1. We have already established the forward implication. We now prove the reverse implication. Let $G = B(E_1, E_2, E_3)$, $G' = B'(E'_1, E'_2, E'_3)$, and $L_G = L_{G'}$. By the definition of legalization, $B(E_1, E_3) \approx B'(E'_1, E'_3)$ and p is negative in G iff p is negative in G' for $p \in \text{dom}(G) = \text{dom}(G')$. Therefore, $G \approx G'$.

We now consider statement 2. We have $u \approx v$ iff $u, v \in L_{\mathcal{E}_u} = L_{\mathcal{E}_v}$ (since legalizations are equivalence classes of legal strings w.r.t. $\approx$) iff $\mathcal{E}_u \approx \mathcal{E}_v$ (by the first statement). ■

Let G be an extended abstract reduction graph, and take $u \in L_G$ (such a u exists since L_G is nonempty). Since $u \in L_{\mathcal{E}_u}$ and legalizations are equivalence classes, we have $L_{\mathcal{E}_u} = L_G$ and therefore $G \approx \mathcal{E}_u$. Thus every extended abstract reduction graph G is isomorphic to an extended reduction graph. In fact, it is isomorphic to precisely those extended reduction graphs $\mathcal{E}_u$ with $u \in L_G$. Therefore, this u is unique up to equivalence.

Corollary 11

Let u and v be legal strings. If $\mathcal{R}_u \approx \mathcal{R}_v$, then there is a $E \in \text{CON}_u$ such that $\mathcal{E}_v \approx B(E_1, E_2, E)$ with $\mathcal{R}_u = B(E_1, E_2)$.

Proof

Since $\mathcal{R}_u \approx \mathcal{R}_v$, there is a set of edges E for $\mathcal{R}_u$ such that $\mathcal{E}_v \approx B(E_1, E_2, E)$. Since $M_v \in \text{CON}_v$, we have $E \in \text{CON}_u$. ■

We end this section with a graph theoretical characterization of reduction graphs.



Figure 4.11: Flip operation for p . All vertices are labelled by p

Theorem 12

Let G be a 2-edge coloured graph. Then G is isomorphic to a reduction graph iff $G \in \mathcal{G}$ and $\text{CON}_G \neq \emptyset$.

Proof

Let $G \approx \mathcal{R}_u$ for some legal string u . Then clearly, $G \in \mathcal{G}$. Also, $M_u \in \text{CON}_u$ and hence $\text{CON}_u \neq \emptyset$. Therefore, $\text{CON}_G \neq \emptyset$.

Let $E \in \text{CON}_G$. Then $G' = B(E_1, E_2, E)$ is an extended abstract reduction graph with $G = B(E_1, E_2)$. By the paragraph below Theorem 10, $G' \approx \mathcal{E}_u$ for some legal string u (take $u \in L_{G'}$). Hence, $G \approx \mathcal{R}_u$. ■

4.7 Flip Edges

In this section and the next two we provide characterizations of the statement $\text{CON}_G \neq \emptyset$. This allows, using Theorem 12, for a characterization that corresponds to an efficient algorithm that determines whether or not a given $G \in \mathcal{G}$ is isomorphic to a reduction graph. Moreover, it allows for an efficient algorithm that determines a legal string u for which $G \approx \mathcal{R}_u$.

Let $G \in \mathcal{G}$. Then a merge-legal set for G is easily obtained as follows. For each $p \in \text{dom}(G)$ with $C_p = \{\{v_1, v_2\}, \{v_3, v_4\}\}$, a merge-legal set for G must have either the edges $\{v_1, v_3\}$ and $\{v_2, v_4\}$ or the edges $\{v_1, v_4\}$ and $\{v_2, v_3\}$, see both sides in Figure 4.11. By assigning such edges for each $p \in \text{dom}(G)$ we obtain a merge-legal set for G . Thus, $\text{ML}_G \neq \emptyset$ for each $G \in \mathcal{G}$. Note that in particular, if $\text{dom}(G) = \emptyset$, then $\text{ML}_G = \{\emptyset\}$. However, CON_G can be empty as the next example illustrates.

Example 9

It is easy to see that the abstract reduction graph G of Figure 4.12 does not have a merge-legal set in CON_G .

We now formally define a type of operation that in Figure 4.11 transforms the situation on the left-hand side to the situation on the right-hand side, and the other way around. Informally speaking it “flips” edges of merge-legal sets.

Definition 13

Let $G = B(E_1, E_2) \in \mathcal{G}$, let f be the vertex labeling function of G , and let $p \in \text{dom}(G)$. The *flip operation for p* (w.r.t. G) is the function $\text{flip}_{G,p} : \text{ML}_G \rightarrow \text{ML}_G$

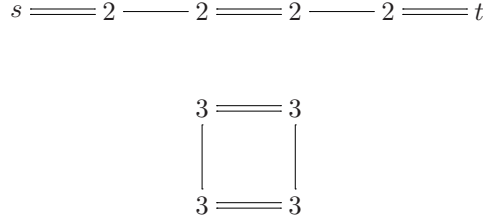


Figure 4.12: An abstract reduction graph G for which $\text{CON}_G = \emptyset$.

defined, for $E \in \text{ML}_G$, by:

$$\text{flip}_{G,p}(E) = \{\{v_1, v_2\} \in E \mid f(v_1) \neq p \neq f(v_2)\} \cup \{e_1, e_2\},$$

where e_1 and e_2 are the two edges with vertices labelled by p such that $e_1, e_2 \notin E_2 \cup E$. ■

When G is clear from the context, we also denote $\text{flip}_{G,p}$ by flip_p .

Since by Figure 4.11, there are exactly two edges e_1 and e_2 with vertices labelled by p that are not parallel to both the edges in $E_2 \cup E$, flip_p is well defined. It is now easy to see that indeed $\text{flip}_p(E) \in \text{ML}_G$ for $E \in \text{ML}_G$.

Example 10

Let G be the abstract reduction graph of Figure 4.6. If we apply $\text{flip}_{G,2}$ to the set of merge-legal edges depicted in Figure 4.7, then we obtain the set of merge-legal edges depicted in Figure 4.8.

The next theorem follows directly from the previous definition and from the fact that Figure 4.11 contains the only possible ways in which edges in merge-legal sets for G can be connected.

Theorem 14

Let $G \in \mathcal{G}$, and denote by $\mathcal{F}$ be the group generated by the flip operations w.r.t. G under function composition. Then each element of $\mathcal{F}$ is self-inverse, thus $\mathcal{F}$ is Abelian, and $\mathcal{F}$ acts transitively on ML_G .

Let $D = \{p_1, \dots, p_l\} \subseteq \text{dom}(G)$. Then we define $\text{flip}_D = \text{flip}_{p_l} \cdots \text{flip}_{p_1}$. Since $\mathcal{F}$ is Abelian, flip_D is well defined. Moreover, since each element in $\mathcal{F}$ is self-inverse, $\mathcal{F} = \{\text{flip}_D \mid D \subseteq \text{dom}(G)\}$. Also, if $D_1, D_2 \subseteq \text{dom}(G)$ and $D_1 \neq D_2$, then $\text{flip}_{D_1}(E) \neq \text{flip}_{D_2}(E)$. Thus the following holds.

Theorem 15

Let $G \in \mathcal{G}$. Then there is a bijection $Q : 2^{\text{dom}(G)} \rightarrow \mathcal{F}$ given by $Q(D) = \text{flip}_D$. Moreover, for each $E \in \text{ML}_G$, $\text{ML}_G = \{\text{flip}_D(E) \mid D \subseteq \text{dom}(G)\}$.

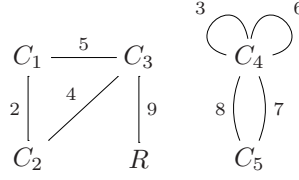


Figure 4.13: The pointer-component graph of the abstract reduction graph from Figure 4.4.

4.8 Merging and Splitting Connected Components

Let $G = B(E_1, E_2)$ be an abstract reduction graph and let $E \in \text{ML}_G$. In this section we consider the effect of the flip operation on the pointer-component graph defined on the abstract reduction graph $H = B(E_1, E)$. If we are able to obtain, using flip operations, a pointer-component graph consisting of one vertex, then $\text{CON}_G \neq \emptyset$, and consequently by Theorem 12, G is isomorphic to a reduction graph.

However, first we need to define the notion of pointer-component graph for abstract reduction graphs in general. Fortunately, this generalization is trivial.

Definition 16

Let $G \in \mathcal{G}$. The *pointer-component graph* of G , denoted by $\mathcal{PC}_G$, is a multigraph (ζ, E, ϵ) , where ζ is the set of connected components of G , $E = \text{dom}(G)$, and ϵ is, for $e \in E$, defined by $\epsilon(e) = \{C \in \zeta \mid C \text{ contains vertices labelled by } e\}$. ■

Example 11

The pointer-component graph of the graph from Figure 4.4 is shown in Figure 4.13.

Note that when $G = B(E_1, E_2) \in \mathcal{G}$ and $E \in \text{ML}_G$, then E is desirable for B . Hence, $H = B(E_1, E)$ is also an abstract reduction graph. Therefore, e.g., $\mathcal{PC}_H$ is defined.

It is useful to distinguish the pointers that form loops in the pointer-component graph. Therefore, we define, for $G \in \mathcal{G}$, $\text{bridge}(G) = \{e \in E \mid |\epsilon(e)| = 2\}$ where $\mathcal{PC}_G = (V, E, \epsilon)$. In [4], $\text{bridge}(G)$ is denoted as $\text{snrdom}(G)$. However, this notation does not make sense for its uses in this chapter.

Example 12

From Figure 4.13 it follows that $\text{bridge}(G) = \text{dom}(G) \setminus \{3, 6\}$ for the abstract reduction graph G depicted in Figure 4.4.

Merge rules have been used for multigraphs, and pointer-component graphs in particular in [4]. The definition presented here is slightly different from the one in [4] – here the pointer p on which the merge rule is applied remains present after the rule is applied.

Definition 17

For each edge p , the p -merge rule, denoted by merge_p , is a rule applicable to (defined on) multigraphs $G = (V, E, \epsilon)$ with $p \in \text{bridge}(G)$. It is defined by

$$\text{merge}_p(G) = (V', E, \epsilon'),$$

where $V' = (V \setminus \epsilon(p)) \cup \{v'\}$ with $v' \notin V$, and $\epsilon'(e) = \{h(v_1), h(v_2)\}$ iff $\epsilon(e) = \{v_1, v_2\}$ where $h(v) = v'$ if $v \in \epsilon(p)$, otherwise it is the identity. ■

It is easy to see that merge rules commute. We are now ready to state the following result which is similar to Theorem 27 in [4].

Theorem 18

Let $G = B(E_1, E_2) \in \mathcal{G}$, let $E \in \text{ML}_G$, let $H = B(E_1, E)$, and let, for $p \in \text{dom}(G)$, $H_p = B(E_1, \text{flip}_p(E))$.

- If $p \in \text{bridge}(H)$, then $\mathcal{PC}_{H_p} \approx \text{merge}_p(\mathcal{PC}_H)$
(and therefore $o(\mathcal{PC}_{H_p}) = o(\mathcal{PC}_H) - 1$).
- If $p \in \text{dom}(H) \setminus \text{bridge}(H)$, then $o(\mathcal{PC}_H) \leq o(\mathcal{PC}_{H_p}) \leq o(\mathcal{PC}_H) + 1$.

Proof

First let $p \in \text{bridge}(H)$. Let $C_{H,p} = \{\{v_1, v_2\}, \{v_3, v_4\}\}$. Then, H has the following form, where each of the two edges in $C_{H,p}$ are from different connected components in H and where, unlike our convention, we have depicted the vertices by their identity instead of their label:

$$\dots \equiv v_1 - - - v_2 \equiv \dots$$

$$\dots \equiv v_3 - - - v_4 \equiv \dots$$

Now, either $\{\{v_1, v_4\}, \{v_2, v_3\}\} \subseteq E_2$ or $\{\{v_1, v_3\}, \{v_2, v_4\}\} \subseteq E_2$. Thus H_p is of either

$$\begin{array}{ccc} \dots \equiv v_1 & & v_2 \equiv \dots \\ & | & | \\ & | & | \\ \dots \equiv v_3 & & v_4 \equiv \dots \end{array}$$

or

$$\begin{array}{ccc} \dots \equiv v_1 & & v_2 \equiv \dots \\ & \diagdown \quad \diagup & \\ & \times & \\ & \diagup \quad \diagdown & \\ \dots \equiv v_3 & & v_4 \equiv \dots \end{array}$$

form, respectively. Thus in both cases, the two connected components are merged, and thus $\mathcal{PC}_{H_p}$ can be obtained (up to isomorphism) from $\mathcal{PC}_H$ by applying the merge_p operation.

Now let $p \in \text{dom}(H) \setminus \text{bridge}(H)$. Then the edges in $C_{H,p}$ belong to the same connected component. Thus H has the following form

$$\cdots \text{---} v_1 \text{---} v_2 \text{---} \cdots \text{---} v_3 \text{---} v_4 \text{---} \cdots$$

where $C_{H,p} = \{\{v_1, v_2\}, \{v_3, v_4\}\}$. Again, we have either $\{\{v_1, v_4\}, \{v_2, v_3\}\} \subseteq E_2$ or $\{\{v_1, v_3\}, \{v_2, v_4\}\} \subseteq E_2$. Thus H_p is of either

$$\begin{array}{ccccccc} \cdots & \text{---} & v_1 & & v_2 & \text{---} & \cdots & \text{---} & v_3 & & v_4 & \text{---} & \cdots \\ & & & \searrow & & \nearrow & & \searrow & & \nearrow & & & \\ & & & & & & & & & & & & \end{array}$$

or

$$\begin{array}{ccccccc} \cdots & \text{---} & v_1 & & v_2 & \text{---} & \cdots & \text{---} & v_3 & & v_4 & \text{---} & \cdots \\ & & & \searrow & & \nearrow & & \searrow & & \nearrow & & & \\ & & & & & & & & & & & & \end{array}$$

form, respectively. Thus, H_p has either the same number of connected components of H or exactly one more, respectively. Thus, $o(\mathcal{PC}_H) \leq o(\mathcal{PC}_{H_p}) \leq o(\mathcal{PC}_H) + 1$. ■

Example 13

Let $G = B(E_1, E_2) \in \mathcal{G}$ be as in Figure 4.6. If we take $E \in \text{ML}_G$ as in Figure 4.7, then $2 \in \text{bridge}(H)$ with $H = B(E_1, E)$. Therefore, by Theorem 18 and the fact that G has exactly two connected components, $H_2 = B(E_1, \text{flip}_2(E))$ is a connected graph. Indeed, this is clear from Figure 4.8 (by ignoring the edges from E_2).

Informally, the next lemma shows that by applying flip operations, we can shrink a connected pointer-component graph to a single vertex. In this way, the underlying abstract reduction graph is a connected graph.

Remark

The next lemma appears to be similar to Lemma 29 in [4]. Although the flip operation (defined on graphs) and the rem operation (defined on strings) are quite distinct, they do have a similar effect on the pointer-component graph.

Lemma 19

Let $G = B(E_1, E_2) \in \mathcal{G}$, let $E \in \text{ML}_G$, let $H = B(E_1, E)$, and let $D \subseteq \text{dom}(G) = \text{dom}(H)$. Then $\mathcal{PC}_H|_D$ is a tree iff $B(E_1, \text{flip}_D(E))$ and H have 1 and $|D| + 1$ connected components, respectively.

Proof

Let $D = \{p_1, \dots, p_n\}$. We first prove the forward implication. If $\mathcal{PC}_H|_D$ is a tree, then it has $|D|$ edges, and thus $|D| + 1$ vertices. Therefore, $\mathcal{PC}_H$ has $|D| + 1$

vertices, and consequently, H has $|D| + 1$ connected components. Since $\mathcal{PC}_H|_D$ is acyclic, by Theorem 18,

$$\mathcal{PC}_{B(E_1, \text{flip}_D(E))} = \mathcal{PC}_{B(E_1, (\text{flip}_{p_n} \cdots \text{flip}_{p_1})(E))} \approx (\text{merge}_{p_n} \cdots \text{merge}_{p_1})(\mathcal{PC}_H).$$

Now, applying $|D|$ merge operations on a graph with $|D| + 1$ vertices, results in a graph containing exactly one vertex. Thus $B(E_1, \text{flip}_D(E))$ has one connected component.

We now prove the reverse implication. Moving from $H = B(E_1, E)$ to graph $B(E_1, \text{flip}_D(E))$ reduces the number of connected components in $|D|$ steps from $|D| + 1$ to 1. By Theorem 18, each flip operation of flip_D corresponds to a merge operation. Therefore $(\text{merge}_{p_n} \cdots \text{merge}_{p_1})$ is applicable to $\mathcal{PC}_H$. Consequently, $\mathcal{PC}_H|_D$ is acyclic. Since this graph has $|D| + 1$ vertices, $\mathcal{PC}_H|_D$ is a tree. ■

4.9 Connectedness of Pointer-Component Graph

In this section we use the results of the previous two sections to prove our first main result, cf. Theorem 24, which strengthens Theorem 12 by replacing the requirement $\text{CON}_G \neq \emptyset$ by a simple test on $\mathcal{PC}_G$. We now characterize the connectedness of $\mathcal{PC}_G$.

Definition 20

Let $B = (V, f, s, t)$ be a coloured base. We say that a set of edges E for B is *well-coloured (for B)* if for each partition $\rho = (V_1, V_2)$ of V with $f(V_1) \cap f(V_2) = \emptyset$, there is an edge $\{v_1, v_2\} \in E$ with $v_1 \in V_1$ and $v_2 \in V_2$. ■

We call $G = B(E_1, E_2) \in \mathcal{G}$ *well-coloured* if E_1 is well-coloured for B .

Lemma 21

Let $G \in \mathcal{G}$. Then $\mathcal{PC}_G$ is a connected graph iff G is well-coloured.

Proof

Let $G = B(E_1, E_2)$ with $B = (V, f, s, t)$. We first prove the forward implication. Let G be not well-coloured. Then there is a partition $\rho = (V_1, V_2)$ of V with $f(V_1) \cap f(V_2) = \emptyset$ such that for each $e \in E_1$, either $e \subseteq V_1$ or $e \subseteq V_2$. Since for each $\{v_1, v_2\} \in E_2$ we have $f(v_1) = f(v_2)$, we have either $\{v_1, v_2\} \subseteq V_1$ or $\{v_1, v_2\} \subseteq V_2$. Therefore V_1 and V_2 induce two non-empty sets of connected components which have no vertex label in common. Therefore, $\mathcal{PC}_G$ is not a connected graph.

We now prove the reverse implication. Assume that $\mathcal{PC}_G = (\zeta, E, \epsilon)$ is not a connected graph. Then, by the definition of pointer-component graph, there is a partition (C_1, C_2) of ζ such that C_1 and C_2 have no vertex label in common. Let V_i be the set of vertices of the connected components in C_i ($i \in \{1, 2\}$). Then for partition $\rho = (V_1, V_2)$ of V we have $f(V_1) \cap f(V_2) = \emptyset$ and for each $e \in E_1 \cup E_2$, either $e \subseteq V_1$ or $e \subseteq V_2$. Therefore G is not well-coloured. ■

Clearly, if $G = B(E_1, E_2) \in \mathcal{G}$ is well-coloured and E is desirable for B (e.g., one could take $E \in \text{ML}_G$), then $H = B(E_1, E) \in \mathcal{G}$ and H is well-coloured. Therefore, by Lemma 21, $\mathcal{PC}_G$ is a connected graph iff $\mathcal{PC}_H$ is a connected graph.

By Theorem 12 the next result is essential to efficiently determine which abstract reduction graphs are isomorphic to reduction graphs.

Theorem 22

Let $G \in \mathcal{G}$. Then $\mathcal{PC}_G$ is a connected graph iff $\text{CON}_G \neq \emptyset$.

Proof

Let $G = B(E_1, E_2)$. We first prove the forward implication. Let $\mathcal{PC}_G$ be a connected graph and let $E \in \text{ML}_G$. Then $\mathcal{PC}_H$ with $H = B(E_1, E)$ is a connected graph. Thus there exists a $D \subseteq \text{dom}(G)$ such that $\mathcal{PC}_H|_D$ is a tree. By Lemma 19, $B(E_1, \text{flip}_D(E))$ is a connected graph, and consequently $\text{flip}_D(E) \in \text{CON}_G$.

We now prove the reverse implication. Let $E \in \text{CON}_G$. Thus, $H = B(E_1, E)$ is a connected graph, and hence $\mathcal{PC}_H$ is a connected graph. Therefore, $\mathcal{PC}_G$ is also a connected graph. ■

We can summarize the last two results as follows.

Corollary 23

Let $G \in \mathcal{G}$. Then the following conditions are equivalent:

1. G is well-coloured,
2. $\mathcal{PC}_G$ is a connected graph, and
3. $\text{CON}_G \neq \emptyset$.

Example 14

By Figure 4.3 and Corollary 23, for (abstract) reduction graph G_1 in Figure 4.2 we have $\text{CON}_{G_1} \neq \emptyset$. On the other hand, by Figure 4.13 and Corollary 23, for abstract reduction graph G_2 in Figure 4.4 we have $\text{CON}_{G_2} = \emptyset$.

By Corollary 23 and Theorem 12 we obtain the first main result of this chapter. It shows that one needs to check only a few computationally easy conditions to determine whether or not a 2-edge coloured graph is (isomorphic to) a reduction graph. Surprisingly, the ‘high-level’ notion of pointer-component graph is crucial in this characterization.

Theorem 24

Let G be a 2-edge coloured graph. Then G isomorphic to a reduction graph iff $G \in \mathcal{G}$ and $\mathcal{PC}_G$ is a connected graph.

Note that in the previous theorem we can equally well replace “ $\mathcal{PC}_G$ is a connected graph” by one of the other equivalent conditions in Corollary 23.

In Theorem 21 in [4] it is shown that the pointer-component graph of each reduction graph is a connected graph. We did not use that result here – in fact it is now a direct consequence of Theorem 24.

Not only is it computationally efficient to determine whether or not a 2-edge coloured graph G is isomorphic to a reduction graph, but, when this is the case, then it is also computationally easy to determine a legal string u for which $G \approx \mathcal{R}_u$. Indeed, we can determine such a u from $G = B(E_1, E_2)$ as follows:

1. Determine an $E \in \text{ML}_G$. As we have mentioned before, such an E is easily obtained.
2. Compute $\mathcal{PC}_H$ with $H = B(E_1, E)$, and determine a set of edges D such that $\mathcal{PC}_H|_D$ is a tree.
3. Compute $G' = B(E_1, E_2, \text{flip}_D(E))$, and determine a $u \in L_{G'}$.

As a consequence, pointer-component graphs of legal strings can, surprisingly, take all imaginable forms.

Corollary 25

Every connected multigraph $G = (V, E, \epsilon)$ with $E \subseteq \Delta$ is isomorphic to a pointer-component graph of a legal string.

4.10 Flip and the Underlying Legal String

We now move to the second part of this chapter, where we characterize the fibers $\mathcal{R}^{-1}(\mathcal{R}_u)$ modulo graph isomorphism. Thus, we describe the set of strings that have the same reduction graph (up to isomorphism) as u . First we consider the effect of flip operations on the set of merge edges.

Lemma 26

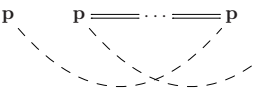
Let u be a legal string and let $p \in \text{dom}(u)$. If p is negative in u , then $\text{flip}_p(M_u) \in \text{CON}_u$. If p is positive in u , then $\text{flip}_p(M_u) \notin \text{CON}_u$. In other words, $\text{flip}_p(M_u) \in \text{CON}_u$ iff p is negative in u .

Proof

Let $\mathcal{R}_u = B(E_1, E_2)$. By the definition of flip_p , $\text{flip}_p(M_u) \in \text{ML}_u$. It suffices to prove that $G = B(E_1, \text{flip}_p(M_u))$ is a connected graph when p is negative in u and not a connected graph when p is positive in u . Graph $B(E_1, M_u)$ has the following form:

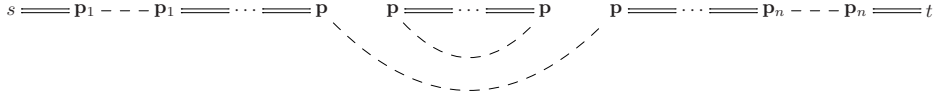
$$s \text{ --- } \mathbf{p}_1 \text{ --- } \mathbf{p}_1 \text{ --- } \cdots \text{ --- } \mathbf{p} \text{ --- } \mathbf{p} \text{ --- } \cdots \text{ --- } \mathbf{p} \text{ --- } \mathbf{p} \text{ --- } \cdots \text{ --- } \mathbf{p}_n \text{ --- } \mathbf{p}_n \text{ --- } t$$

Now if p is negative in u , then G has the following form:

$$s \text{ --- } \mathbf{p}_1 \text{ --- } \mathbf{p}_1 \text{ --- } \cdots \text{ --- } \mathbf{p} \text{ --- } \mathbf{p} \text{ --- } \cdots \text{ --- } \mathbf{p} \text{ --- } \mathbf{p} \text{ --- } \cdots \text{ --- } \mathbf{p}_n \text{ --- } \mathbf{p}_n \text{ --- } t$$


Thus in this case G is connected.

If p is positive in u , then G has the following form:



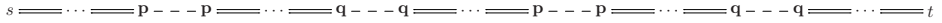
Thus in this case G is not connected. ■

Lemma 27

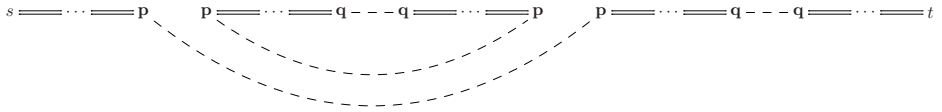
Let u be a legal string and let $p, q \in \text{dom}(u)$. If p and q are overlapping in u and not both negative in u , then $\text{flip}_{\{p,q\}}(M_u) \in \text{CON}_u$.

Proof

Let $\mathcal{R}_u = B(E_1, E_2)$. Then $B(E_1, M_u)$ has the following form (we can assume without loss of generality that p appears before q in the path from s to t):



Assume that p is positive in u – the other case (q is positive in u) is proved similarly. By the proof of Lemma 26 it follows that $B(E_1, \text{flip}_p(M_u))$ has the following form:



Therefore, $q \in \text{bridge}(B(E_1, \text{flip}_p(M_u)))$. By Theorem 18, the pointer-component graph of $B(E_1, \text{flip}_{\{p,q\}}(M_u))$ has only one vertex. Hence, $B(E_1, \text{flip}_{\{p,q\}}(M_u))$ is connected and thus $\text{flip}_{\{p,q\}}(M_u) \in \text{CON}_u$. ■

Lemma 28

Let u be a legal string, and let $D \subseteq \text{dom}(u)$ be nonempty. If $\text{flip}_D(M_u) \in \text{CON}_u$, then either there is a $p \in D$ negative in u or there are $p, q \in D$ positive and overlapping in u .

Proof

Let $\mathcal{E}_u = B(E_1, E_2, M_u)$ and let $\text{flip}_D(M_u) \in \text{CON}_u$. Then $B(E_1, \text{flip}_D(M_u))$ is a connected graph. Assume to the contrary that all elements in D are positive and pairwise non-overlapping in u . Then there is a $p \in D$ such that the domain of the p -interval does not contain an element in $D \setminus \{p\}$. By the proof of Lemma 26, $B(E_1, \text{flip}_p(M_u))$ consists of two connected components, one of which does not have vertices labelled by elements in $D \setminus \{p\}$. Therefore $B(E_1, \text{flip}_D(M_u))$ also contains this connected component, and thus $B(E_1, \text{flip}_D(M_u))$ has more than one connected component – a contradiction. ■

By the previous lemmata, we have the following result.

Theorem 29

Let u be a legal string, and let $D \subseteq \text{dom}(u)$ be nonempty. If $\text{flip}_D(M_u) \in \text{CON}_u$, then either there is a $p \in D$ negative in u with $\text{flip}_p(M_u) \in \text{CON}_u$ or there are $p, q \in D$ positive and overlapping in u with $\text{flip}_{\{p,q\}}(M_u) \in \text{CON}_u$.

4.11 Dual String Pointer Rules

We now define the dual string pointer rules. These rules will be used to characterize the effect of flip operations on the underlying legal string. For all $p, q \in \Pi$ with $\mathbf{p} \neq \mathbf{q}$, we define

- the *dual string positive rule* for p by $\mathbf{dspr}_p(u_1pu_2pu_3) = u_1p\bar{u}_2pu_3$,
- the *dual string double rule* for p, q by $\mathbf{dsdr}_{p,q}(u_1pu_2qu_3\bar{p}u_4\bar{q}u_5) = u_1pu_4qu_3\bar{p}u_2\bar{q}u_5$,

where $u_1, u_2, \dots, u_5$ are arbitrary (possibly empty) strings over Π . Notice that the dual string pointer rules are self-inverse.

The names of these rules are due to their strong similarities with the two of the three types of string rewriting rules of a specific model of gene assembly, called string pointer reduction system (SPRS) [12]. In this model, gene assembly is performed by three types of recombination (splicing) operations that are subsequently modeled as string rewriting rules. For convenience we now recall these string rewriting rules.

For all $p, q \in \Pi$ with $\mathbf{p} \neq \mathbf{q}$, we define

- the *string negative rule* for p by $\mathbf{snr}_p(u_1ppu_2) = u_1u_2$,
- the *string positive rule* for p by $\mathbf{spr}_p(u_1pu_2\bar{p}u_3) = u_1\bar{u}_2u_3$,
- the *string double rule* for p, q by $\mathbf{sdr}_{p,q}(u_1pu_2qu_3pu_4qu_5) = u_1u_4u_3u_2u_5$,

where $u_1, u_2, \dots, u_5$ are arbitrary (possibly empty) strings over Π .

Notice the strong similarities between $\mathbf{dspr}$ and $\mathbf{spr}$, and between $\mathbf{dsdr}$ and $\mathbf{sdr}$. Both $\mathbf{dspr}_p$ and $\mathbf{spr}_p$ invert the substring between the two occurrences of p or $\bar{p}$. However, $\mathbf{dspr}_p$ is applicable when p is negative, while $\mathbf{spr}_p$ is applicable when p is positive. Also, $\mathbf{spr}_p$ removes the occurrences of p and $\bar{p}$, while $\mathbf{dspr}$ does not. A similar comparison can be made between $\mathbf{dsdr}$ and $\mathbf{sdr}$.

The *domain* of a dual string pointer rule ρ , denoted by $\text{dom}(\rho)$, is defined by $\text{dom}(\mathbf{dspr}_p) = \{\mathbf{p}\}$ and $\text{dom}(\mathbf{dsdr}_{p,q}) = \{\mathbf{p}, \mathbf{q}\}$ for $p, q \in \Pi$. For a composition $\varphi = \rho_n \cdots \rho_2 \rho_1$ of such rules $\rho_1, \rho_2, \dots, \rho_n$, the *domain*, denoted by $\text{dom}(\varphi)$, is $\text{dom}(\rho_1) \cup \text{dom}(\rho_2) \cup \cdots \cup \text{dom}(\rho_n)$. Also, we define $\text{odom}(\varphi) = \bigoplus_{1 \leq i \leq n} \text{dom}(\rho_i)$. Thus, $\text{odom}(\varphi) \subseteq \text{dom}(\varphi)$ consists of the pointers that are used an odd number of times. We call φ *reduced* if every $p \in \text{dom}(\varphi)$ is used exactly once, i.e., $\text{dom}(\rho_i) \cap \text{dom}(\rho_j) = \emptyset$ for all $1 \leq i < j \leq n$. Note that if φ is reduced, then $\text{dom}(\varphi) = \text{odom}(\varphi)$.

Definition 30

Let u and v be legal strings. We say that u and v are *dual*, denoted by $\approx_d$ if there is a (possibly empty) sequence φ of dual string pointer rules applicable to u such that $\varphi(u) \approx v$. ■

Notice that $\approx_d$ is an equivalence relation. Clearly, $\approx_d$ is reflexive. It is symmetrical since dual string pointer rules are self-inverse, and it is transitive by function composition: if $\varphi_1(u) \approx v$ and $\varphi_2(v) \approx w$, then $(\varphi_2 \varphi_1)(u) \approx w$.

Since $\mathbf{dspr}_p$ is applicable when p is negative in u and $\mathbf{dsdr}_{p,q}$ is applicable when p and q are positive and overlapping, the following result is a direct corollary to Lemma 28.

Corollary 31

Let u be a legal string, and let $D \subseteq \text{dom}(u)$ be nonempty. If $\text{flip}_D(M_u) \in \text{CON}_u$, then there is a dual string pointer rule ρ with $\text{dom}(\rho) \subseteq D$ applicable to u .

Let $G = B(E_1, E_2, E_3)$ be an extended abstract reduction graph, and let $D \subseteq \text{dom}(G)$. Then we define $\text{flip}_D(G) = B(E_1, E_2, \text{flip}_{G',D}(E_3))$, where $G' = B(E_1, E_2)$.

Lemma 32

Let u be a legal string, and let φ be a sequence of dual string rules applicable to u . Then $\mathcal{E}_{\varphi(u)} \approx \text{flip}_D(\mathcal{E}_u)$ with $D = \text{odom}(\varphi)$. Consequently, $\mathcal{R}_{\varphi(u)} \approx \mathcal{R}_u$.

Proof

It suffices to prove the result for the case $\varphi = \mathbf{dspr}_p$ with $p \in \Pi$ and for the case $\varphi = \mathbf{dsdr}_{p,q}$ with $p, q \in \Pi$. We first prove the case where $\varphi = \mathbf{dspr}_p$ for some $p \in \Pi$ is applicable to u . Then by the second figure in the proof of Lemma 26 we see that the inversion of the substring between the two occurrences of p in u accomplished by φ faithfully simulates the corresponding effect of flip_p on $\mathcal{E}_u$. We only need to verify that p is negative in $\text{flip}_p(\mathcal{E}_u)$. To do this, we depict $\mathcal{E}_u$ such that the vertices are represented by their identity instead of their label:

$$s \text{ --- } \dots \text{ --- } v_1 \text{ --- } v_2 \text{ --- } \dots \text{ --- } v_3 \text{ --- } v_4 \text{ --- } \dots \text{ --- } t$$

where the vertices v_i , $i \in \{1, 2, 3, 4\}$, are labelled by $\mathbf{p}$. Then $\text{flip}_p(\mathcal{E}_u)$ is

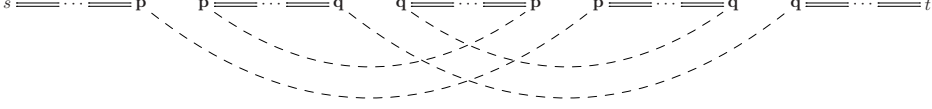
$$s \text{ --- } \dots \text{ --- } v_1 \text{ --- } v_3 \text{ --- } \dots \text{ --- } v_2 \text{ --- } v_4 \text{ --- } \dots \text{ --- } t$$

Therefore p is indeed negative in $\text{flip}_p(\mathcal{E}_u)$, and consequently $\mathcal{E}_{\varphi(u)} \approx \text{flip}_p(\mathcal{E}_u)$.

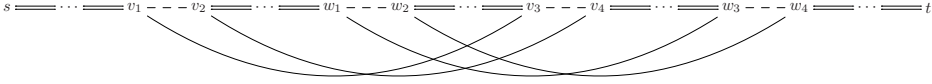
We now prove the case where $\varphi = \mathbf{dsdr}_{p,q}$ with $p, q \in \Pi$. Let $\mathcal{E}_u = B(E_1, E_2, E_3)$, then $\mathcal{E}_u$ has the following form

$$s \text{ --- } \dots \text{ --- } \mathbf{p} \text{ --- } \mathbf{p} \text{ --- } \dots \text{ --- } \mathbf{q} \text{ --- } \mathbf{q} \text{ --- } \dots \text{ --- } \mathbf{p} \text{ --- } \mathbf{p} \text{ --- } \dots \text{ --- } \mathbf{q} \text{ --- } \mathbf{q} \text{ --- } \dots \text{ --- } t$$

where we omitted the edges in E_2 . Since p and q are positive in u , $\text{flip}_{\{p,q\}}(\mathcal{E}_u)$ has the following form:



where we again omitted the edges in E_2 . Thus, we see that interchanging the substring in u between p and q and the substring in u between $\bar{p}$ and $\bar{q}$ accomplished by φ faithfully simulates the corresponding effect of $\text{flip}_{p,q}$ on $\mathcal{E}_u$. We only need to verify that both p and q are positive in $\text{flip}_{p,q}(\mathcal{E}_u)$. To do this, we depict $\mathcal{E}_u$ such that the vertices are represented by their identity instead of their label:



where the vertices v_i and w_i , $i \in \{1, 2, 3, 4\}$, are labelled by p and q , respectively. Then $\text{flip}_{p,q}(\mathcal{E}_u)$ is



Therefore both p and q are indeed positive in $\text{flip}_{p,q}(\mathcal{E}_u)$, and consequently $\mathcal{E}_{\varphi(u)} \approx \text{flip}_{p,q}(\mathcal{E}_u)$. ■

Thus, if φ_1 and φ_2 are sequences of dual string pointer rules applicable to a legal string u with $\text{odom}(\varphi_1) = \text{odom}(\varphi_2)$, then $\mathcal{E}_{\varphi_1(u)} \approx \mathcal{E}_{\varphi_2(u)}$ and thus $\varphi_1(u) \approx \varphi_2(u)$.

Lemma 33

Let u be a legal string, and let $D \subseteq \text{dom}(u)$. There is a reduced sequence φ of dual string pointer rules applicable to u such that $\text{dom}(\varphi) = D$ iff $\text{flip}_D(M_u) \in \text{CON}_u$.

Proof

The forward implication follows directly from Lemma 32. We now prove the reverse implication. If $D = \emptyset$, we have nothing to prove. Let $D \neq \emptyset$. By Corollary 31, there is a dual string pointer rule ρ_1 with $D_1 = \text{dom}(\rho_1) \subseteq D$ applicable to u . By Lemma 32, $\mathcal{E}_{\rho_1(u)} \approx \text{flip}_{D_1}(\mathcal{E}_u)$ and $D_1 = \text{odom}(\rho_1) = \text{dom}(\rho_1)$. Thus, $\text{flip}_{D \setminus D_1}(M_{\rho_1(u)}) \in \text{CON}_{\rho_1(u)}$. Now by iteration, there is a reduced sequence φ of dual string pointer rules applicable to u such that $\text{odom}(\varphi) = \text{dom}(\varphi) = D$. ■

It follows from Lemmata 32 and 33 that reduced sequences of dual string pointer rules are a normal form of sequences of dual string pointer rules. Indeed, by Lemma 32, if φ is a sequence of dual string pointer rules applicable to a legal string u with $D = \text{odom}(\varphi)$, then $\text{flip}_D(M_u) \in \text{CON}_u$. By Lemma 33, there is a reduced sequence φ' of dual string pointer rules applicable to u such that $\text{dom}(\varphi') = \text{odom}(\varphi') = D$. By the paragraph below Lemma 32, we have $\varphi(u) \approx \varphi'(u)$.

We are now ready to prove the second (and final) main result of this chapter. It shows that the fiber $\mathcal{R}^{-1}(\mathcal{R}_u)$ for each legal string u is the ‘orbit’ of u under the dual string pointer rules. Hence, the partition of the set of all legal strings induced by the fibers under $\mathcal{R}$, and the one induced by $\approx_d$ coincide. Equivalently, the legal strings obtained from u by applying dual string pointer rules are exactly those legal strings that have the same reduction graph as u (up to isomorphism).

Theorem 34

Let u and v be legal strings. Then $\mathcal{R}_u \approx \mathcal{R}_v$ iff $u \approx_d v$.

Proof

The reverse implication follows directly from Lemma 32. We now prove the forward implication. Let $\mathcal{R}_u \approx \mathcal{R}_v$. By Corollary 11, there is a $E \in \text{CON}_u$ such that $\mathcal{E}_v \approx B(E_1, E_2, E)$ with $\mathcal{R}_u = B(E_1, E_2)$. By Theorem 15, $E = \text{flip}_D(M_u)$ for some $D \subseteq \text{dom}(u)$. Since $\text{flip}_D(M_u) \in \text{CON}_u$, by Lemma 33, there is a reduced sequence φ of dual string pointer rules applicable to u such that $\text{dom}(\varphi) = D$. Now by Lemma 32, $\mathcal{E}_{\varphi(u)} \approx \text{flip}_D(\mathcal{E}_u) \approx \mathcal{E}_v$, and therefore, by Theorem 10, $\varphi(u) \approx v$. ■

4.12 Discussion

This chapter characterizes, letting $\mathcal{R}$ be the function that assigns to each legal string u its reduction graph $\mathcal{R}_u$, the range of $\mathcal{R}$ (Theorem 24) and each fiber $\mathcal{R}^{-1}(\mathcal{R}_u)$ (Theorem 34) modulo graph isomorphism.

The first characterization corresponds to a computationally efficient algorithm that determines whether or not a graph G is isomorphic to a reduction graph. Moreover, if this is the case, then the algorithm given below Theorem 24 allows for an efficient determination of a legal string u such that $G \approx \mathcal{R}_u$. The first characterization relies on the notion of merge-legal edges and its flip operation introduced in this chapter. In particular, the connected components in the subgraph induced by the reality edges and the merge-legal edges and the flip operation turn out to be relevant in this context.

The second characterization determines, given u , the whole set $\mathcal{R}^{-1}(\mathcal{R}_u)$. It turns out that $\mathcal{R}^{-1}(\mathcal{R}_u)$ is the orbit of u under the dual string pointer rules. Moreover, each two legal strings u and v in such a fiber can be transformed into each other by a sequence φ of dual string pointer rules without using any pointer more than once. Therefore, the number of dual string pointer rules in φ can be bounded by the size of the domain of u (and v). Surprisingly, the dual string

pointer rules are very similar to those used in a specific model of gene assembly called SPRS.

The second characterization has additional uses for SPRS. The reduction graph of a legal string u in a certain sense retains all information regarding applicability of string negative rules (defined in SPRS) in transformations of u to its end result, while discarding almost all other information regarding the applicability of the other rules, see [4]. Therefore, the fibers in a sense provide the equivalence classes of legal strings having the same properties regarding the application of string negative rules.

From a biological point of view, the first characterization provide requirements on the structure of MAC genes, while the second characterization determines which types of MIC genes obtain the same MAC structure.

Chapter 5

How Overlap Determines Reduction Graphs for Gene Assembly

Abstract

Ciliates are unicellular organisms having two types of functionally different nuclei: micronucleus and macronucleus. Gene assembly transforms a micronucleus into a macronucleus, thereby transforming each gene from its micronuclear form to its macronuclear form. Within a formal intramolecular model of gene assembly based on strings, the notion of reduction graph represents the macronuclear form of a gene, including byproducts, given only a description of the micronuclear form of that gene. For a more abstract model of gene assembly based on graphs, one cannot, in general, define the notion of reduction graphs. We show that if we restrict ourselves to the so-called realistic overlap graphs (which correspond to genes occurring in nature), then the notion of reduction graph can be defined in a manner equivalent to the string model. This allows one to carry over from the string model to the graph model several results that rely on the notion of reduction graph.

5.1 Introduction

Gene assembly is a process that takes place in unicellular organisms called ciliates, which have two types of functionally different nuclei: micronucleus (MIC) and macronucleus (MAC). Gene assembly transforms the genome of the MIC into the genome of the MAC. The two genomes are dramatically different in both the global form of their chromosomes and in the local form of single genes. During gene assembly each gene in its MIC form gets transformed into the same gene in its MAC form. See [12] for a detailed account of the biology of gene assembly.

In this chapter we consider only intramolecular models of gene assembly – thus here we do not consider the intermolecular models initiated by Landweber and Kari [20], and further developed by Daley et al. [10, 9]. Among the formal models of intramolecular gene assembly the string pointer reduction system (SPRS) and the graph pointer reduction system (GPRS), see [12], are of interest for this chapter. The SPRS consist of three types of string rewriting rules operating on so-called legal strings, while the GPRS consist of three types of graph rewriting rules operating on so-called overlap graphs. The GPRS is an abstraction of the SPRS: some information present in the SPRS is lost in the GPRS.

Realistic strings are strings that represent genes in their micronuclear form. Legal strings are an abstraction of realistic strings. The reduction graph, which is defined for legal strings, is a notion that describes the gene corresponding to the legal string in its macronuclear form (along with its waste products: the substrings “spliced out” in the process) – it is unique for a given legal string. It has been shown that the reduction graph retains the information needed to characterize which string negative rules (one of the three types of string rewriting rules) can be used during the transformation of a MIC form of a gene to its MAC form [6, 4]. Therefore it would be useful to have a notion of the reduction graph also for the GPRS. However, this is not so straightforward. We will demonstrate that, since the GPRS loses some information concerning the application of string negative rules, in general there is no unique reduction graph for a given overlap graph, cf. Example 6. However, as we will show, when we restrict ourselves to “realistic” overlap graphs then one gets a unique reduction graph. These overlap graphs are called realistic since they correspond to (micronuclear) genes. In this chapter, we explicitly define the notion of reduction graph for realistic overlap graphs (within the GPRS) and show that it is equivalent to the notion of reduction graph for legal strings (within the SPRS). Finally, we give a number of direct corollaries of this equivalence, including an answer to an open problem formulated in Chapter 13 in [12].

In Section 5.2 we recall some basic notions and notation concerning sets, strings and graphs. In Section 5.3 we recall notions used in models for gene assembly, such as legal strings, realistic strings and overlap graphs. In Section 5.4 we recall the notion of reduction graph within the framework of SPRS and we prove some elementary properties of this graph for legal strings. In particular we establish a calculus for the sets of overlapping pointers between vertices of the reduction graph. In Section 5.5 we prove properties of the reduction graph for a more restricted type of legal strings, the realistic strings. It is shown that reduction graphs of realistic strings have a subgraph of a specific structure, the root subgraph. Moreover, we show (using the calculus from Section 5.4) that the existence of the other edges in the reduction graph depends directly on the overlap graph. In Section 5.6 we provide a convenient function for reduction graphs that allows one to simplify reduction graphs without losing any information. In Section 5.7 we define the reduction graph for realistic overlap graphs, and prove the main theorem of this chapter: the equivalence of reduction graphs defined for realistic strings with

the reduction graphs defined for realistic overlap graphs. In Section 5.8 we discuss some immediate consequences of this theorem. A conference version of this chapter, which does not contain any proofs, was presented at FCT '07 [7].

5.2 Notation and Terminology

In this section we recall some basic notions concerning functions, strings, and graphs. We do this mainly to set up the basic notation and terminology for this chapter.

The cardinality of a set X is denoted by $|X|$. The symmetric difference of sets X and Y , $(X \setminus Y) \cup (Y \setminus X)$, is denoted by $X \oplus Y$. Since symmetric difference is associative, we extend it to (finite) families of sets $(X_i)_{i \in A}$, and denote this by $\bigoplus_{i \in A} X_i$. The *composition* of functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ is the function $gf : X \rightarrow Z$ such that $(gf)(x) = g(f(x))$ for every $x \in X$. The restriction of f to a subset A of X is denoted by $f|A$.

We use λ to denote the empty string. For strings u and v , we say that v is a *substring* of u if $u = w_1vw_2$, for some strings w_1, w_2 ; we also say that v *occurs in* u . Also, v is a *cyclic substring* of u if either v is a substring of u or $u = v_2wv_1$ and $v = v_1v_2$ for some strings v_1, v_2, w . We say that v is a *conjugate* of u if $u = w_1w_2$ and $v = w_2w_1$ for some strings w_1 and w_2 . For a string $u = x_1x_2 \cdots x_n$ over Σ with $x_i \in \Sigma$ for all $i \in \{1, \dots, n\}$, we say that $v = x_nx_{n-1} \cdots x_1$ is the *reversal* of u . A *homomorphism* is a function $\varphi : \Sigma^* \rightarrow \Delta^*$ such that $\varphi(uv) = \varphi(u)\varphi(v)$ for all $u, v \in \Sigma^*$.

We move now to graphs. A *labelled graph* is a 4-tuple $G = (V, E, f, \Gamma)$, where V is a finite set, $E \subseteq \{\{x, y\} \mid x, y \in V, x \neq y\}$, and $f : V \rightarrow \Gamma$. The elements of V are called *vertices* and the elements of E are called *edges*. Function f is the *labelling function* and the elements of Γ are the *labels*. Note that our graphs are not directed and do not have loops.

We say that G is *discrete* if $E = \emptyset$. Labelled graph $G' = (V', E', f|V', \Gamma)$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E_{V'} = E \cap \{\{x, y\} \mid x, y \in V', x \neq y\}$. If $E' = E_{V'}$, we say that G' is the *subgraph of G induced by V'* . A string $\pi = e_1e_2 \cdots e_n \in E^*$ with $n \geq 1$ is a *path in G* if there is a $v_1v_2 \cdots v_{n+1} \in V^*$ such that $e_i = \{v_i, v_{i+1}\}$ for all $1 \leq i \leq n$. Labelled graph G is *connected* if there is a path between every two vertices of G . A subgraph H of G induced by V_H is a *component* of G if both H is connected and for every edge $e \in E$ we have either $e \subseteq V_H$ or $e \subseteq V \setminus V_H$.

Labelled graphs $G = (V, E, f, \Gamma)$ and $G' = (V', E', f', \Gamma)$ are *isomorphic*, denoted by $G \approx G'$, if there is a bijection $\alpha : V \rightarrow V'$ such that $f(v) = f'(\alpha(v))$ for all $v \in V$, and $\{x, y\} \in E$ iff $\{\alpha(x), \alpha(y)\} \in E'$ for all $x, y \in V$. Any such bijection α is then called an *isomorphism from G to G'* . It is important to realize that we require that the labels of vertices identified by an isomorphism are equal.

In this chapter we will consider graphs with two sets of edges. Therefore, we need the notion of 2-edge coloured graphs. A *2-edge coloured graph* is a 5-tuple $G = (V, E_1, E_2, f, \Gamma)$, where both (V, E_1, f, Γ) and (V, E_2, f, Γ) are labelled graphs.

The basic notions and notation for labelled graphs carry over to 2-edge coloured graphs. However, to extend the notion of isomorphism care must be taken that the two sorts of edges are preserved. Thus, if $G = (V, E_1, E_2, f, \Gamma)$ and $G' = (V', E'_1, E'_2, f', \Gamma')$ are 2-edge coloured graphs, and α is an isomorphism from G to G' , then $(x, y) \in E_i$ iff $(\alpha(x), \alpha(y)) \in E'_i$ for $x, y \in V$ and $i \in \{1, 2\}$.

5.3 Gene Assembly in Ciliates

Two models that are used to formalize the process of gene assembly in ciliates are the string pointer reduction system (SPRS) and the graph pointer reduction system (GPRS). The SPRS consist of three types of string rewriting rules operating on *legal strings* while the GPRS consist of three types of graph rewriting rules operating on *overlap graphs*. For the purpose of this chapter it is not necessary to recall the string and graph rewriting rules; a complete description of SPRS and GPRS, as well as a proof of their “weak” equivalence, can be found in [12]. We do recall the notions of legal string and overlap graph, and we also recall the notion of realistic string.

We fix $\kappa \geq 2$, and define the alphabet $\Delta = \{2, 3, \dots, \kappa\}$. For $D \subseteq \Delta$, we define $\bar{D} = \{\bar{a} \mid a \in D\}$ and $\Pi_D = D \cup \bar{D}$; also $\Pi = \Pi_\Delta$. The elements of Π are called *pointers*. We use the “bar operator” to move from Δ to $\bar{\Delta}$ and back from $\bar{\Delta}$ to Δ . Hence, for $p \in \Pi$, $\bar{\bar{p}} = p$. For $p \in \Pi$, we define $\mathbf{p}$ to be p if $p \in \Delta$, and $\bar{p}$ if $p \in \bar{\Delta}$, i.e., $\mathbf{p}$ is the “unbarred” variant of p . For a string $u = x_1 x_2 \cdots x_n$ with $x_i \in \Pi$ ($1 \leq i \leq n$), the *complement* of u is $\bar{x}_1 \bar{x}_2 \cdots \bar{x}_n$. The *inverse* of u , denoted by $\bar{u}$, is the complement of the reversal of u , thus $\bar{u} = \bar{x}_n \bar{x}_{n-1} \cdots \bar{x}_1$. The *domain* of u , denoted by $\text{dom}(u)$, is $\{\mathbf{p} \mid p \text{ occurs in } u\}$. We say that u is a *legal string* if for each $p \in \text{dom}(u)$, u contains exactly two occurrences from $\{p, \bar{p}\}$. For a pointer p and a legal string u , if both p and $\bar{p}$ occur in u then we say that both p and $\bar{p}$ are *positive* in u ; if on the other hand only p or only $\bar{p}$ occurs in u , then both p and $\bar{p}$ are *negative* in u . So, every pointer occurring in a legal string is either positive or negative in it. Therefore, we can define a partition of $\text{dom}(u) = \text{pos}(u) \cup \text{neg}(u)$, where $\text{pos}(u) = \{p \in \text{dom}(u) \mid p \text{ is positive in } u\}$ and $\text{neg}(u) = \{p \in \text{dom}(u) \mid p \text{ is negative in } u\}$.

Let $u = x_1 x_2 \cdots x_n$ be a legal string with $x_i \in \Pi$ for $1 \leq i \leq n$. For a pointer $p \in \Pi$, the *p-interval* of u is the substring $x_i x_{i+1} \cdots x_j$ with $\{x_i, x_j\} \subseteq \{p, \bar{p}\}$ and $1 \leq i < j \leq n$. Substrings $x_{i_1} \cdots x_{j_1}$ and $x_{i_2} \cdots x_{j_2}$ *overlap* in u if $i_1 < i_2 < j_1 < j_2$ or $i_2 < i_1 < j_2 < j_1$. Two distinct pointers $p, q \in \Pi$ *overlap* in u if the p -interval of u overlaps with the q -interval of u . Thus, two distinct pointers $p, q \in \Pi$ overlap in u iff there is exactly one occurrence from $\{p, \bar{p}\}$ in the q -interval, or equivalently, there is exactly one occurrence from $\{q, \bar{q}\}$ in the p -interval of u . Also, for $p \in \text{dom}(u)$, we denote the set of all $q \in \text{dom}(u)$ such that p and q overlap in u by $O_u(p)$, and for $0 \leq i \leq j \leq n$, we denote by $O_u(i, j)$ the set of all $p \in \text{dom}(u)$ such that there is exactly one occurrence from $\{p, \bar{p}\}$ in $x_{i+1} x_{i+2} \cdots x_j$. Also, we define $O_u(j, i) = O_u(i, j)$. Intuitively, $O_u(i, j)$ is the set of $p \in \text{dom}(u)$ for which the substring between “positions” i and j in u contains exactly one representative

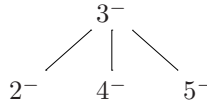


Figure 5.1: The overlap graph of legal string $u = 24535423$.

from $\{p, \bar{p}\}$, where position i for $0 < i < n$ means the “space” between x_i and x_{i+1} in u . For $i = 0$ it is the “space” to the left of x_1 , and for $i = n$ it is the “space” to the right of x_n . A few elementary properties of $O_u(i, j)$ follow. We have $O_u(i, n) = O_u(0, i)$ for i with $0 \leq i \leq n$. Moreover, for $i, j, k \in \{0, \dots, n\}$, $O_u(i, j) \oplus O_u(j, k) = O_u(i, k)$; this is obvious when $i < j < k$, but it is valid in general. Also, for $0 \leq i \leq j \leq n$, $O_u(i, j) = \emptyset$ iff $x_{i+1} \cdots x_j$ is a legal string.

Definition 1

Let u be a legal string. The *overlap graph* of u , denoted by γ_u , is the labelled graph $(\text{dom}(u), E, \sigma, \{+, -\})$, where for $p, q \in \text{dom}(u)$ with $p \neq q$, $\{p, q\} \in E$ iff p and q overlap in u , and σ is defined by: $\sigma(p) = +$ if $p \in \text{pos}(u)$, and $\sigma(p) = -$ if $p \in \text{neg}(u)$. ■

Example 1

Let $u = 24535423$ be a legal string. The overlap graph of u is

$$\gamma = (\{2, 3, 4, 5\}, \{\{2, 3\}, \{4, 3\}, \{5, 3\}\}, \sigma, \{+, -\}),$$

where $\sigma(v) = -$ for all vertices v of γ . The overlap graph is depicted in Figure 5.1.

Let γ be the overlap graph of a legal string u . We define $\text{dom}(\gamma)$ as the set of vertices of γ , $\text{pos}(\gamma) = \{p \in \text{dom}(\gamma) \mid \sigma(p) = +\}$, $\text{neg}(\gamma) = \{p \in \text{dom}(\gamma) \mid \sigma(p) = -\}$, and for $q \in \text{dom}(u)$, $O_\gamma(q) = \{p \in \text{dom}(\gamma) \mid \{p, q\} \in E\}$. We have $\text{dom}(\gamma) = \text{dom}(u)$, $\text{pos}(\gamma) = \text{pos}(u)$, $\text{neg}(\gamma) = \text{neg}(u)$, and $O_\gamma(q) = O_u(q)$ for all $q \in \text{dom}(\gamma) = \text{dom}(u)$.

We define the alphabet $\Theta_\kappa = \{M_i, \bar{M}_i \mid 1 \leq i \leq \kappa\}$, and say that $\delta \in \Theta_\kappa^*$ is a *micronuclear arrangement* if for each i with $1 \leq i \leq \kappa$, δ contains exactly one occurrence from $\{M_i, \bar{M}_i\}$. With each string over Θ_κ , we associate a unique string over Π through the homomorphism $\pi_\kappa : \Theta_\kappa^* \rightarrow \Pi^*$ defined by: $\pi_\kappa(M_1) = 2$, $\pi_\kappa(M_\kappa) = \kappa$, $\pi_\kappa(M_i) = i(i+1)$ for $1 < i < \kappa$, and $\pi_\kappa(\bar{M}_j) = \overline{\pi_\kappa(M_j)}$ for $1 \leq j \leq \kappa$. A string u is a *realistic string* if there is a micronuclear arrangement δ such that $u = \pi_\kappa(\delta)$. We then say that δ is a *micronuclear arrangement* for u .

Note that every realistic string is a legal string. However, not every legal string is a realistic string. For example, a realistic string cannot have “gaps” (missing pointers): thus 2244 is not realistic while it is legal. It is also easy to produce examples of legal strings which do not have gaps but still are not realistic — 3322 is such an example. Realistic strings are most useful for the gene assembly models, since only these legal strings can correspond to genes in ciliates.

An overlap graph γ is called *realistic* if it is the overlap graph of a realistic string. Not every overlap graph of a legal string is realistic. For example, it can be shown that the overlap graph γ of $u = 24535423$ depicted in Figure 5.1 is not realistic. In fact, one can show that it is not even *realizable* — there is no isomorphism α such that $\alpha(\gamma)$ is realistic.

5.4 The Reduction Graph

We now recall the notion of a (full) reduction graph, which was first introduced in [6].

Remark

Below we present the notion of reduction graph in a slightly modified form: we omit the special vertices s and t , called the source vertex and target vertex respectively, which did appear in the definition presented in [6]. As shown in Section 5.5, in this way a realistic overlap graph corresponds to exactly one reduction graph. Fortunately, several results concerning reduction graphs do not rely on the special vertices, and therefore carry over in a straightforward way to reduction graphs as defined here.

Definition 2

Let $u = p_1 p_2 \cdots p_n$ with $p_1, \dots, p_n \in \Pi$ be a legal string. The *reduction graph of u* , denoted by $\mathcal{R}_u$, is the 2-edge coloured graph

$$(V, E_1, E_2, f, \text{dom}(u)),$$

where

$$V = \{I_1, I_2, \dots, I_n\} \cup \{I'_1, I'_2, \dots, I'_n\},$$

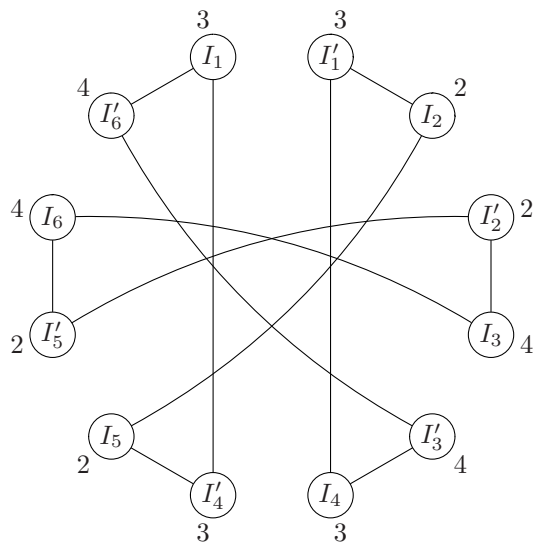
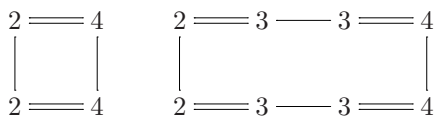
$$E_1 = \{e_1, e_2, \dots, e_n\} \text{ with } e_i = \{I'_i, I_{i+1}\} \text{ for } 1 \leq i \leq n-1, e_n = \{I'_n, I_1\},$$

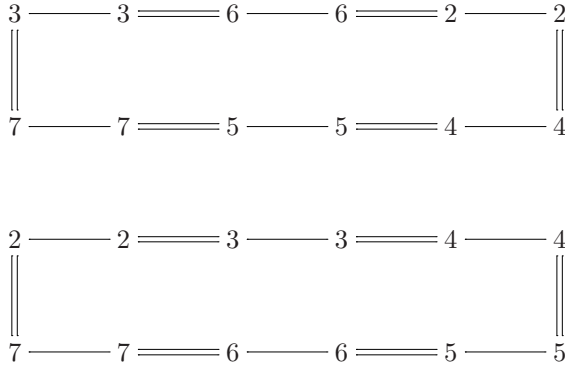
$$E_2 = \{ \{I'_i, I_j\}, \{I_i, I'_j\} \mid i, j \in \{1, 2, \dots, n\} \text{ with } i \neq j \text{ and } p_i = p_j \} \cup \\ \{ \{I_i, I_j\}, \{I'_i, I'_j\} \mid i, j \in \{1, 2, \dots, n\} \text{ and } p_i = \bar{p}_j \}, \text{ and}$$

$$f(I_i) = f(I'_i) = \mathbf{p}_i \text{ for } 1 \leq i \leq n.$$

■

The edges of E_1 are called the *reality edges*, and the edges of E_2 are called the *desire edges*. Intuitively, the “space” between p_i and p_{i+1} corresponds to the reality edge $e_i = \{I'_i, I_{i+1}\}$. Hence, we say that i is the *position of e_i* , denoted by $\text{posn}(e_i)$, for all $i \in \{1, 2, \dots, n\}$. Note that positions are only defined for reality edges. Since for every vertex v there is a unique reality edge e such that $v \in e$, we also define the *position of v* , denoted by $\text{posn}(v)$, as the position of e . Thus, $\text{posn}(I'_i) = \text{posn}(I_{i+1}) = i$ (while $\text{posn}(I_1) = n$).

Figure 5.2: The reduction graph of u from Example 2.Figure 5.3: The reduction graph of u from Example 2 in the simplified representation.

Figure 5.4: The reduction graph of u from Example 3.**Example 2**

Let $u = 3243\bar{2}4$ be a legal string. Since $\bar{4}3\bar{2}$ can not be a substring of a realistic string, u is not realistic. The reduction graph $\mathcal{R}_u$ of u is depicted in Figure 5.2. The labels of the vertices are also shown in this figure. Note that the desire edges corresponding to positive pointers (here 2 and 4) cross (in the figure), while those for negative pointers are parallel.

We consider reduction graphs up to isomorphism. Therefore, the exact identity of the vertices in a reduction graph is not essential for the problems considered in this chapter, and in pictorial representations of reduction graphs we denote the vertices by their labels. We also depict reality edges by double line segments to distinguish them from the desire edges. Figure 5.3 shows the reduction graph of Figure 5.2 in this simplified representation.

Example 3

Let $u = \pi_7(M_7M_1M_6M_3M_5\overline{M_2}M_4) = 72673456\bar{3}245$. Thus, unlike in the previous example, u is a realistic string. The reduction graph is given in Figure 5.4. Note that according to our convention, the vertices are represented by their labels.

The reduction graph is defined for legal strings. In this chapter, we show how to directly construct the reduction graph of a realistic string from its overlap graph. In this way we can define the reduction graph for realistic overlap graphs in a direct way.

Next we consider sets of overlapping pointers corresponding to pairs of vertices in reduction graphs, and we begin to develop a calculus for these sets that will later enable us to characterize the existence of certain edges in the reduction graph, cf. Theorem 15.

Lemma 3

Let u be a legal string. Let $e = \{v_1, v_2\}$ be a desire edge of $\mathcal{R}_u$ and let p be the

label of both v_1 and v_2 . Then

$$O_u(\text{posn}(v_1), \text{posn}(v_2)) = \begin{cases} O_u(p) & \text{if } p \text{ is negative in } u, \\ O_u(p) \oplus \{p\} & \text{if } p \text{ is positive in } u. \end{cases}$$

Proof

Let $u = p_1 p_2 \dots p_n$ with $p_1, p_2, \dots, p_n \in \Pi$ and let i and j be such that $i < j$ and $p = p_i = p_j$. Without loss of generality, we can assume $\text{posn}(v_1) < \text{posn}(v_2)$. Then, $v_1 \in \{I_i, I'_i\}$ and $v_2 \in \{I_j, I'_j\}$, hence $\text{posn}(v_1) \in \{i-1, i\}$ and $\text{posn}(v_2) \in \{j-1, j\}$.

First, assume that p is negative in u . Then, by the definition of reduction graph, the following two cases are possible:

1. $e = \{I_i, I'_j\}$, thus $O_u(\text{posn}(I_i), \text{posn}(I'_j)) = O_u(i-1, j) = O_u(p)$,
2. $e = \{I'_i, I_j\}$, thus $O_u(\text{posn}(I_i), \text{posn}(I'_j)) = O_u(i, j-1) = O_u(p)$,

Thus in both cases we have $O_u(\text{posn}(v_1), \text{posn}(v_2)) = O_u(p)$.

Now, assume that p is positive in u . Then, by the definition of reduction graph, the following two cases are possible:

1. $e = \{I_i, I_j\}$, thus $O_u(\text{posn}(I_i), \text{posn}(I_j)) = O_u(i-1, j-1) = O_u(p) \oplus \{p\}$,
2. $e = \{I'_i, I'_j\}$, thus $O_u(\text{posn}(I'_i), \text{posn}(I'_j)) = O_u(i, j) = O_u(p) \oplus \{p\}$,

Thus in both cases we have $O_u(i_1, i_2) = O_u(p) \oplus \{p\}$. ■

Let u be a legal string. For $P \subseteq \text{dom}(u)$, we define $\Pi_u(P) = (\text{pos}(u) \cap P) \oplus (\bigoplus_{t \in P} O_u(t))$. Similarly, we define $\Pi_\gamma(P)$ for an overlap graph γ (by replacing u by γ in the definition).

The following result follows by iteratively applying Lemma 3 and using the definition of $\Pi_u(P)$.

Corollary 4

Let u be a legal string. Let

$$p_0 \equiv p_1 \text{ --- } p_1 \equiv p_2 \text{ --- } p_2 \equiv \dots \equiv p_n \text{ --- } p_n \equiv p_{n+1}$$

be a subgraph of $\mathcal{R}_u$, and let e_1 (e_2 , resp.) be the leftmost (rightmost, resp.) edge. Then $O_u(\text{posn}(e_1), \text{posn}(e_2)) = \Pi_u(P)$ with $P = \{p_1, \dots, p_n\}$.

Note that, in the above, e_1 and e_2 are reality edges and therefore $\text{posn}(e_1)$ and $\text{posn}(e_2)$ are defined.

By the definition of reduction graph the following lemma holds.

Lemma 5

Let u be a legal string. If I_i and I'_i are vertices of $\mathcal{R}_u$, then $O_u(\text{posn}(I_i), \text{posn}(I'_i)) = \{p\}$, where p is the label of I_i and I'_i .

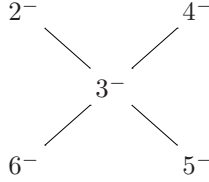


Figure 5.5: The overlap graph of both legal strings u and v from Example 5.

Example 4

We again consider the legal string $u = 32\bar{4}3\bar{2}4$ and its reduction graph $\mathcal{R}_u$ from Example 2. Desire edge $e = \{I'_2, I'_5\}$ with vertices labelled by 2 is connected to reality edges $\{I'_2, I'_3\}$ and $\{I'_5, I'_6\}$ with positions 2 and 5 respectively. By Lemma 3, we have $O_u(2, 5) = O_u(2) \oplus \{2\} = \{2, 3, 4\}$. This can of course also be verified by directly considering the corresponding substring $\bar{4}3\bar{2}$ between positions 2 and 5 of u . Also, since I_2 and I'_2 with positions 1 and 2 respectively are labelled by 2, by Lemma 5 we have $O_u(1, 2) = \{2\}$.

5.5 The Reduction Graph of Realistic Strings

The next theorem asserts that the overlap graph γ for a realistic string u retains all information of $\mathcal{R}_u$ (up to isomorphism). In this chapter, we give a method to determine $\mathcal{R}_u$ (up to isomorphism), from γ . Of course, the naive method is to first determine a legal string u corresponding to γ and then to determine the reduction graph of u . However, we present a method that allows one to construct $\mathcal{R}_u$ in a direct way from γ .

Theorem 6

Let u and v be realistic strings. If $\gamma_u = \gamma_v$, then $\mathcal{R}_u \approx \mathcal{R}_v$.

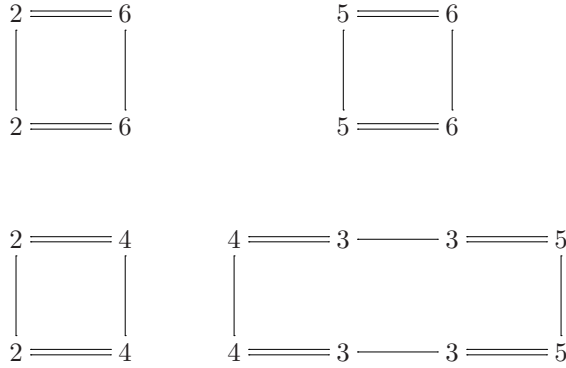
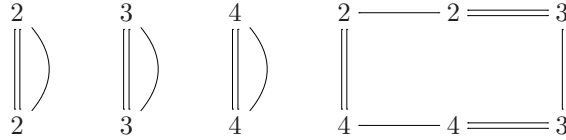
Proof

By Theorem 1 in [19] (or Theorem 10.2 in [12]), we have $\gamma_u = \gamma_v$ iff v can be obtained from u by a composition of reversal, complement and conjugation operations. By the definition of reduction graph it is clear that the reduction graph is invariant under these operations (up to isomorphism). Thus, $\mathcal{R}_u \approx \mathcal{R}_v$. ■

This theorem does *not* hold for legal strings in general — the next two examples illustrate that legal strings having the same overlap graph can have different reduction graphs up to isomorphism.

Example 5

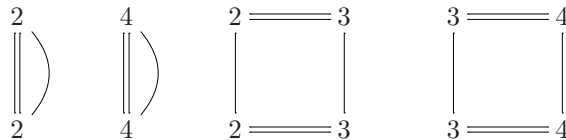
Let $u = 2653562434$ and $v = h(u)$, where h is the homomorphism that interchanges 5 and 6. Thus, $v = 2563652434$. Note that both u and v are not realistic, because substrings 535 of u and 636 of v can obviously not be substrings of realistic strings. The overlap graph of u is depicted in Figure 5.5. From Figure 5.5

Figure 5.6: The reduction graph of u from Example 5.Figure 5.7: The reduction graph of u from Example 6.

and the fact that v is obtained from u by renumbering 5 and 6, it follows that the overlap graphs of u and v are equal. The reduction graph $\mathcal{R}_u$ of u is depicted in Figure 5.6. The reduction graph $\mathcal{R}_v$ of v is obtained from $\mathcal{R}_u$ by renumbering the labels of the vertices according to h . Clearly, $\mathcal{R}_u \not\approx \mathcal{R}_v$.

Example 6

Let $u = \pi_\kappa(M_1M_2M_3M_4) = 223344$ be a realistic string and let $v = 234432$ be a legal string. Note that v is not realistic. Legal strings u and v have the same overlap graph γ ($\gamma = (\{2, 3, 4\}, \emptyset, \sigma, \{+, -\})$, where $\sigma(v) = -$ for $v \in \{2, 3, 4\}$). The reduction graph $\mathcal{R}_u$ of u is depicted in Figure 5.7, and the reduction graph $\mathcal{R}_v$ of v is depicted in Figure 5.8. Note that $\mathcal{R}_u$ has a component consisting of six vertices, while $\mathcal{R}_v$ does not have such a component. Therefore, $\mathcal{R}_u \not\approx \mathcal{R}_v$.

Figure 5.8: The reduction graph of v from Example 6.

For realistic strings the reduction graph has a special form. This is seen as follows. For $1 < i < \kappa$ the symbol M_i (or $\bar{M}_i$) in the micronuclear arrangement defines two pointers p_i and p_{i+1} (or $\bar{p}_{i+1}$ and $\bar{p}_i$) in the corresponding realistic string u . At the same time the substring $p_i p_{i+1}$ (or $\bar{p}_{i+1} \bar{p}_i$, respectively) of u corresponding to M_i (or $\bar{M}_i$, respectively) defines four vertices $I_j, I'_j, I_{j+1}, I'_{j+1}$ in $\mathcal{R}_u$. It is easily verified (cf. Theorem 8 below) that the “middle” two vertices I'_j and I_{j+1} , labelled by p_i and p_{i+1} respectively, are connected by a reality edge and I'_j (I_{j+1} , respectively) is connected by a desire edge to a “middle vertex” resulting from M_{i-1} or $\bar{M}_{i-1}$ (M_{i+1} or $\bar{M}_{i+1}$, respectively). This leads to the following definition.

Definition 7

Let u be a legal string and let $\kappa = |\text{dom}(u)| + 1$. If $\mathcal{R}_u$ contains a subgraph L of the following form:

$$2 \text{ --- } 2 \text{ === } 3 \text{ --- } 3 \text{ === } \dots \text{ === } \kappa \text{ --- } \kappa$$

where the vertices in the figure are represented by their labels, then we say that u is *rooted* and L is called a *root subgraph* of $\mathcal{R}_u$. ■

Example 7

The realistic string u with $\text{dom}(u) = \{2, 3, \dots, 7\}$ from Example 3 is rooted because the reduction graph of u , depicted in Figure 5.4, contains the subgraph

$$2 \text{ --- } 2 \text{ === } 3 \text{ --- } 3 \text{ === } \dots \text{ === } 7 \text{ --- } 7$$

The next theorem shows that indeed every realistic string is rooted.

Theorem 8

Every realistic string is rooted.

Proof

Consider a micronuclear arrangement for a realistic string u . Let $\kappa = |\text{dom}(u)| + 1$. By the definition of π_κ , there is a reality edge e_i (corresponding to either $\pi_\kappa(M_i) = i(i+1)$ or $\pi_\kappa(\bar{M}_i) = (i+1)\bar{i}$) connecting a vertex labelled by i to a vertex labelled by $i+1$ for each $2 \leq i < \kappa$. It suffices to prove that there is a desire edge connecting e_i to e_{i+1} for each $2 \leq i < \kappa - 1$. This can easily be seen by checking the four cases where e_i corresponds to either $\pi_\kappa(M_i)$ or $\pi_\kappa(\bar{M}_i)$, and e_{i+1} corresponds to either $\pi_\kappa(M_{i+1})$ or $\pi_\kappa(\bar{M}_{i+1})$. ■

In the remainder of this chapter, we denote $|\text{dom}(u)| + 1$ just by κ for rooted strings, whenever the rooted string u is understood from the context of considerations. The reduction graph of a realistic string may have more than one root subgraph: it is easy to verify that realistic string $234 \dots \kappa 234 \dots \kappa$ for $\kappa \geq 2$ has two root subgraphs.

Example 2 shows that not every rooted string is realistic. The results in the remainder of this chapter that consider realistic strings also hold for rooted strings,

since we will not be using any properties of realistic string that are not true for rooted strings in general.

For a given root subgraph L , it is convenient to uniquely identify every reality edge containing a vertex of L . This is done through the following definition.

Definition 9

Let u be a rooted string and let L be a root subgraph of $\mathcal{R}_u$. We define $rspos_{L,k}$ for $2 \leq k < \kappa$ as the position of the edge of L that has vertices labelled by k and $k+1$. We define $rspos_{L,1}$ ($rspos_{L,\kappa}$, resp.) as the position of the edge of $\mathcal{R}_u$ not in L containing a vertex of L labelled by 2 (κ , resp.). When $\kappa = 2$, to ensure that $rspos_{L,1}$ and $rspos_{L,\kappa}$ are well defined, we additionally require that $rspos_{L,1} < rspos_{L,\kappa}$. ■

Thus, $rspos_{L,k}$ (for $1 \leq k \leq \kappa$) uniquely identifies every reality edge containing a vertex of L . If it is clear which root subgraph L is meant, we simply write $rspos_k$ instead of $rspos_{L,k}$ for $1 \leq k \leq \kappa$.

The next lemma is essential to prove the main result (Theorem 15) of this chapter.

Lemma 10

Let u be a rooted string. Let L be a root subgraph of $\mathcal{R}_u$. Let i and j be positions of reality edges in $\mathcal{R}_u$ that are not edges of L . Then $O_u(i, j) = \emptyset$ iff $i = j$.

Proof

The reverse implication is trivially satisfied. We now prove the forward implication. The reality edge e_k (for $2 \leq k < \kappa$) in L with vertices labelled by k and $k+1$ corresponds to a cyclic substring $\tilde{M}_k \in \{p_1 p_2, p_2 p_1 \mid p_1 \in \{k, \bar{k}\}, p_2 \in \{k+1, \bar{k}+1\}\}$ of u . Let k_1 and k_2 with $2 \leq k_1 < k_2 < \kappa$. If $k_1 + 1 = k_2$, then reality edges e_{k_1} and e_{k_2} are connected by a desire edge (by the definition of L). Therefore, pointer k_2 common in $\tilde{M}_{k_1}$ and $\tilde{M}_{k_2}$ originates from two different occurrences in u . If on the other hand $k_1 + 1 \neq k_2$, then $\tilde{M}_{k_1}$ and $\tilde{M}_{k_2}$ do not have a letter in common. Therefore, in both cases, $\tilde{M}_{k_1}$ and $\tilde{M}_{k_2}$ are disjoint cyclic substrings of u . Thus the $\tilde{M}_k$ for $2 \leq k < \kappa$ are pairwise disjoint cyclic substrings of u .

Without loss of generality assume $i \leq j$. Let $u = u_1 u_2 \cdots u_n$ with $u_i \in \Pi$. Since u is a legal string, every u_l for $1 \leq l \leq n$ is either part of a $\tilde{M}_k$ (with $2 \leq k < \kappa$) or in $\{2, \bar{2}, \kappa, \bar{\kappa}\}$. Consider $u' = u_{i+1} u_{i+2} \cdots u_j$. Since i and j are positions of reality edges in $\mathcal{R}_u$ that are not edges of L , we have $u' = \tilde{M}_{k_1} \tilde{M}_{k_2} \cdots \tilde{M}_{k_m}$ for some distinct $k_1, k_2, \dots, k_m \in \{1, 2, \dots, \kappa\}$, where $\tilde{M}_1 \in \{2, \bar{2}\}$ and $\tilde{M}_\kappa \in \{\kappa, \bar{\kappa}\}$.

It suffices to prove that $u' = \lambda$. Assume to the contrary that $u' \neq \lambda$. Then there is a $1 \leq l \leq \kappa$ such that $\tilde{M}_l$ is a substring of u' . Because $O_u(i, j) = \emptyset$, we know that u' is legal. If $l > 1$, then $\tilde{M}_{l-1}$ is also a substring of u' , otherwise u' would not be a legal string. Similarly, if $l < \kappa$, then $\tilde{M}_{l+1}$ is also a substring of u' . By iteration, we conclude that $u' = u$. Therefore, $i = 0$. This is a contradiction, since 0 cannot be a position of a reality edge. Thus, $u' = \lambda$. ■

Lemma 11

Let u be a rooted string. Let L be a root subgraph of $\mathcal{R}_u$. If I_i and I'_i are vertices of $\mathcal{R}_u$, then exactly one of I_i and I'_i belongs to L .

Proof

By the definition of reduction graph, I_i and I'_i have a common vertex label p but are not connected by a desire edge. Therefore, I_i and I'_i do not both belong to L . Now, if I_i and I'_i both do not belong to L , then the other vertices labelled by p , which are I_j and I'_j for some j , both belong to L – a contradiction by the previous argument. Therefore, either I_i or I'_i belongs to L , and the other one does not belong to L . ■

The following result captures the main idea that allows for the determination of the reduction graph from the overlap graph only. It relies heavily on the previous lemmas.

Very roughly, the intuition is that there is a reality edge with vertices labelled by p and q outside a fixed root subgraph L precisely when: we can make a “sidestep over” p in the underlying string u “into” L and then “walk over” L to q and finally make a sidestep over q in u in such a way that the accumulated overlap is the empty set.

Theorem 12

Let u be a rooted string, let L be a root subgraph of $\mathcal{R}_u$, and let $p, q \in \text{dom}(u)$ with $p < q$. There is a reality edge e in $\mathcal{R}_u$ with both vertices not in L , one labelled by p and the other by q iff $\Pi_u(P) = \{p, q\}$ where $P = \{p + 1, \dots, q - 1\} \cup P'$ for some $P' \subseteq \{p, q\}$.

Proof

We first prove the forward implication. Let $e = \{v_1, v_2\}$ with v_1 labelled by p , v_2 labelled by q , and $\text{posn}(e) = i$. Thus $e = \{I'_i, I_{i+1}\}$. We assume that $v_1 = I'_i$ and $v_2 = I_{i+1}$, the other case is proved similarly. Let $i_1 = \text{posn}(I_i)$ and $i_2 = \text{posn}(I'_{i+1})$. By Lemma 5, $O_u(i, i_1) = \{p\}$ and $O_u(i_2, i) = \{q\}$. By Lemma 11, I_i (labelled by p) and I'_{i+1} (labelled by q) belong to L . Thus $i_1 \in \{\text{rspos}_{p-1}, \text{rspos}_p\}$ and $i_2 \in \{\text{rspos}_{q-1}, \text{rspos}_q\}$. By applying Corollary 4 on L , we have $O_u(i_1, i_2) = \Pi_u(P)$ with $P = \{p + 1, \dots, q - 1\} \cup P'$ for some $P' \subseteq \{p, q\}$. By definition of $O_u(i, j)$ we have

$$\emptyset = O_u(i, i) = O_u(i, i_1) \oplus O_u(i_1, i_2) \oplus O_u(i_2, i)$$

Since $p \neq q$, we have $\{p\} \oplus \{q\} = \{p, q\}$, and the desired result follows.

We now prove the reverse implication. By applying Corollary 4 on L , we have $O_u(i_1, i_2) = \Pi_u(P)$ for some $i_1 \in \{\text{rspos}_{p-1}, \text{rspos}_p\}$ and $i_2 \in \{\text{rspos}_{q-1}, \text{rspos}_q\}$ (depending on P'). By Lemma 5, there is a vertex v_1 (v_2 , resp.) labelled by p (q , resp.) with position i (j , resp.) such that $O_u(i, i_1) = \{p\}$ and $O_u(i_2, j) = \{q\}$. By Lemma 11 these vertices are not in L . We have now

$$\emptyset = O_u(i, i_1) \oplus O_u(i_1, i_2) \oplus O_u(i_2, j) = O_u(i, j)$$

By Lemma 10, $O_u(i, j) = \emptyset$ implies that $i = j$. Thus, there is a reality edge $\{v_1, v_2\}$ in $\mathcal{R}_u$ (with position i), such that v_1 is labelled by p and v_2 is labelled by q and both are not vertices of L . ■

Let γ_u be the overlap graph of some legal string u . Clearly we have $\text{pos}(u) = \text{pos}(\gamma_u)$ and for all $p \in \text{dom}(u) = \text{dom}(\gamma_u)$, $O_u(p) = O_{\gamma_u}(p)$. Thus by Theorem 12 we can determine, given the overlap graph of a rooted string u , if there is a reality edge in $\mathcal{R}_u$ with both vertices outside L that connects a vertex labelled by p to a vertex labelled by q . We will extend this result to completely determine the reduction graph given the overlap graph of a rooted string (or a realistic string in particular).

5.6 Compressing the Reduction Graph

In this section we define the cps function. The cps function simplifies reduction graphs by replacing the subgraph $p \text{ --- } p$ by a single vertex labelled by p . In this way, one can simplify reduction graphs without “losing information”. We define cps for a general family of graphs $\mathcal{G}$ which includes all reduction graphs. The formal definitions of $\mathcal{G}$ and cps are given below.

Let $\mathcal{G}$ be the set of 2-edge coloured graphs $G = (V, E_1, E_2, f, \Gamma)$ such that $f(v_1) = f(v_2)$ for all $\{v_1, v_2\} \in E_2$. Note that for a reduction graph $\mathcal{R}_u$, we have $\mathcal{R}_u \in \mathcal{G}$ because both vertices of a desire edge have the same label. For all $G \in \mathcal{G}$, $\text{cps}(G)$ is obtained from G by considering the second set of edges as vertices in the labelled graph. Thus, for the case when G is a reduction graph, the function cps “compresses” the desire edges to vertices.

Definition 13

The function cps from $\mathcal{G}$ to the set of labelled graphs is defined as follows. If $G = (V, E_1, E_2, f, \Gamma) \in \mathcal{G}$, then

$$\text{cps}(G) = (E_2, E'_1, f', \Gamma)$$

is a labelled graph, where

$$E'_1 = \{\{e_1, e_2\} \subseteq E_2 \mid \exists v_1, v_2 \in V : v_1 \in e_1, v_2 \in e_2, e_1 \neq e_2 \text{ and } \{v_1, v_2\} \in E_1\},$$

and for $e \in E_2$: $f'(e) = f(v)$ with $v \in e$. ■

Note that f' is well defined, because for all $\{v_1, v_2\} \in E_2$, it holds that $f(v_1) = f(v_2)$.

Example 8

We are again considering the realistic string u defined in Example 3. The reduction graph of $\mathcal{R}_u$ is depicted in Figure 5.4. The labelled graph $\text{cps}(\mathcal{R}_u)$ is depicted in Figure 5.9. Since this graph has just one set of edges, the reality edges are depicted by single line segments rather than double line segments as we did for reduction graphs.

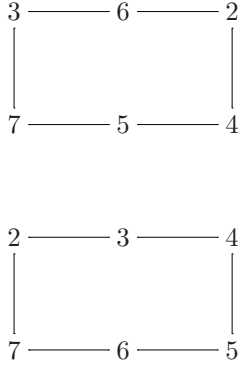


Figure 5.9: The labelled graph $\text{cps}(\mathcal{R}_u)$, where $\mathcal{R}_u$ is defined in Example 8.

It is not hard to see that for reduction graphs $\mathcal{R}_u$ and $\mathcal{R}_v$, we have $\mathcal{R}_u \approx \mathcal{R}_v$ iff $\text{cps}(\mathcal{R}_u) \approx \text{cps}(\mathcal{R}_v)$. In this sense, the cps function allows one to simplify reduction graphs without losing information.

5.7 From Overlap Graph to Reduction Graph

In this section we define (compressed) reduction graphs for realistic overlap graphs, inspired by the characterization from Theorem 12, and then demonstrate their equivalence to reduction graphs for realistic strings.

Definition 14

Let $\gamma = (Dom_\gamma, E_\gamma, \sigma, \{+, -\})$ be a realistic overlap graph and let $\kappa = |Dom_\gamma| + 1$. The *reduction graph* of γ , denoted by $\mathcal{R}_\gamma$, is a labelled graph

$$\mathcal{R}_\gamma = (V, E, f, Dom_\gamma),$$

where

$$V = \{J_p, J'_p \mid 2 \leq p \leq \kappa\},$$

$$f(J_p) = f(J'_p) = p, \text{ for } 2 \leq p \leq \kappa, \text{ and}$$

$e \in E$ iff one of the following conditions hold:

1. $e = \{J'_p, J'_{p+1}\}$ and $2 \leq p < \kappa$.
2. $e = \{J_p, J_q\}$, $2 \leq p < q \leq \kappa$, and $\Pi_\gamma(P) = \{p, q\}$, where $P = \{p+1, \dots, q-1\} \cup P'$ for some $P' \subseteq \{p, q\}$.
3. $e = \{J'_2, J_p\}$, $2 \leq p \leq \kappa$, and $\Pi_\gamma(P) = \{p\}$, where $P = \{2, \dots, p-1\} \cup P'$ for some $P' \subseteq \{p\}$.

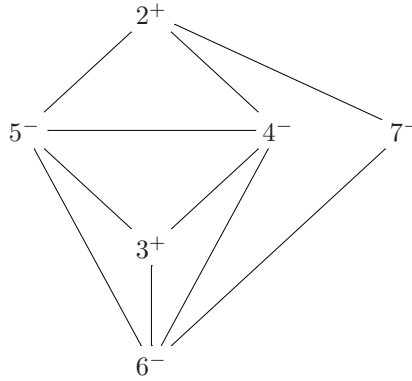


Figure 5.10: The overlap graph γ of a realistic string (used in Example 9).

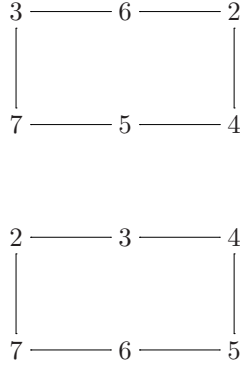
4. $e = \{J'_\kappa, J_p\}$, $2 \leq p \leq \kappa$, and $\Pi_\gamma(P) = \{p\}$, where $P = \{p+1, \dots, \kappa\} \cup P'$ for some $P' \subseteq \{p\}$.
5. $e = \{J'_2, J'_\kappa\}$, $\kappa > 3$, and $\Pi_\gamma(P) = \emptyset$, where $P = \{2, \dots, \kappa\}$. ■

An algorithm that constructs $\mathcal{R}_\gamma$ is efficiently implemented by observing that $\Pi_\gamma(P)$ only needs to be calculated for all intervals $P = [i, j] = \{i, \dots, j\}$ with $i \leq j$ and $i, j \in \{2, \dots, \kappa\}$. These values can be stored in an upper-triangular matrix $A = (a_{i,j})$ having $a_{i,j} = \Pi_\gamma([i, j])$. Note that A can be defined recursively as follows: we have $a_{i,j} = a_{i,j-1} \oplus a_{j,j}$ if $i < j$, and $a_{i,i} = O_\gamma(i)$ if i is negative in γ , and $a_{i,i} = O_\gamma(i) \oplus \{i\}$ if i is positive in γ . After calculating A , we can obtain the edges of $\mathcal{R}_\gamma$. If $2 < i$ and $j < \kappa$, then $a_{i,j} \in \{\{i, j\}, \{i-1, j\}, \{i, j+1\}, \{i-1, j+1\}\}$ iff there is an edge $e = \{J_p, J_q\}$ with $a_{i,j} = \{p, q\}$. The cases $2 = i$ or $j = \kappa$ are handled similarly.

Example 9

The overlap graph γ in Figure 5.10 is realistic. Indeed, realistic string $u = \pi_7(M_7 M_1 M_6 M_3 M_5 \overline{M_2} M_4) = 726734563\bar{2}45$ introduced in Example 3 has this overlap graph. Clearly, the reduction graph $\mathcal{R}_\gamma$ of γ has the edges $\{J'_p, J'_{p+1}\}$ for $2 \leq p < 7$.

Now to obtain the remaining edges, we construct the upper-triangular matrix $A = (a_{i,j})$ having $a_{i,j} = \Pi_\gamma([i, j])$ with $i \leq j$ and $i, j \in \{2, \dots, \kappa\}$ (as discussed above, this can be done recursively). This matrix is given below, where the entries corresponding to edges of $\mathcal{R}_\gamma$ are underlined.

Figure 5.11: The reduction graph $\mathcal{R}_\gamma$ of the overlap graph γ from Example 9.

$\Pi_\gamma([i, j])$	2	3	4	5	6	7
2	$\{2, 4, 5, 7\}$	$\{2, 3, 6, 7\}$	$\{5, 7\}$	$\{2, 3, 4, 5, 6, 7\}$	<u>$\{2, 6\}$</u>	$\emptyset$
3		$\{3, 4, 5, 6\}$	<u>$\{2, 4\}$</u>	<u>$\{3, 6\}$</u>	$\{4, 5, 6, 7\}$	$\{2, 4, 5, 7\}$
4			$\{2, 3, 5, 6\}$	<u>$\{4, 5\}$</u>	$\{3, 7\}$	$\{2, 3, 6, 7\}$
5				$\{2, 3, 4, 6\}$	$\{2, 5, 6, 7\}$	<u>$\{5, 7\}$</u>
6					$\{3, 4, 5, 7\}$	$\{2, 3, 4, 5, 6, 7\}$
7						$\{2, 6\}$

From matrix A we see that, $a_{2,7} = \emptyset$ corresponds to edge $\{J'_2, J'_7\}$, while the other underlined values $\{2, 4\}$, $\{4, 5\}$, $\{5, 7\}$, $\{3, 7\}$, $\{3, 6\}$, and $\{2, 6\}$ correspond to edges $\{J_2, J_4\}$, $\{J_4, J_5\}$, $\dots$, $\{J_2, J_6\}$, respectively.

We have now completely determined $\mathcal{R}_\gamma$; it is shown in Figure 5.11 (again, as we have done for reduction graphs of legal strings, in the figures the vertices of reduction graphs of realistic overlap graphs are represented by their labels).

Figures 5.9 and 5.11 show that, for $u = 72673456\bar{3}245$, $\text{cps}(\mathcal{R}_u) \approx \mathcal{R}_\gamma$. The next theorem shows that this is a general property for realistic strings u .

Theorem 15

Let u be a realistic string. Then, $\text{cps}(\mathcal{R}_u) \approx \mathcal{R}_{\gamma_u}$.

Proof

Let $\kappa = |\text{dom}(u)| + 1$, let $\gamma = \gamma_u$, let $\mathcal{R}_\gamma = (V_\gamma, E_\gamma, f_\gamma, \text{dom}(\gamma))$, let $R_u = \text{cps}(\mathcal{R}_u) = (V_u, E_u, f_u, \text{dom}(u))$, and let L be a root subgraph of $\mathcal{R}_u$. Recall that the elements of V_u are the desire edges of $\mathcal{R}_u$.

Let $h : V_u \rightarrow V_\gamma$ defined by

$$h(v) = \begin{cases} J_{f_u(v)} & \text{if } v \text{ is not an edge of } L, \\ J'_{f_u(v)} & \text{if } v \text{ is an edge of } L. \end{cases}$$

We will show that h is an isomorphism from R_u to $\mathcal{R}_\gamma$. Since for every $l \in \text{dom}(u)$ there exists exactly one desire edge v of $\mathcal{R}_u$ that belongs to L with $f_u(v) = l$ and there exists exactly one desire edge v of $\mathcal{R}_u$ that does not belong to L with $f_u(v) = l$, it follows that h is one-to-one and onto. Also, it is clear from the definition of f_γ that $f_u(v) = f_\gamma(h(v))$. Thus, it suffices to prove that $\{v_1, v_2\} \in E_u \Leftrightarrow \{h(v_1), h(v_2)\} \in E_\gamma$.

We first prove the forward implication $\{v_1, v_2\} \in E_u \Rightarrow \{h(v_1), h(v_2)\} \in E_\gamma$. Let $\{v_1, v_2\} \in E_u$, let $p = f_u(v_1)$ and let $q = f_u(v_2)$. Clearly, $v_1 \neq v_2$. By the definition of cps, there is a reality edge $\tilde{e} = \{\tilde{v}_1, \tilde{v}_2\}$ of $\mathcal{R}_u$ with $\tilde{v}_1 \in v_1$ and $\tilde{v}_2 \in v_2$ (and thus $\tilde{v}_1$ and $\tilde{v}_2$ are labelled by p and q in $\mathcal{R}_u$, respectively). Let i be the position of $\tilde{e}$. We consider four cases (remember that v_1 and v_2 are both desire edges of $\mathcal{R}_u$):

1. Assume that $\tilde{e}$ belongs to L . Then clearly, v_1 and v_2 are edges of L . Without loss of generality, we can assume that $p \leq q$. From the structure of root subgraph and the fact that $\tilde{e}$ is a reality edge of $\mathcal{R}_u$ in L , it follows that $q = p + 1$. Now, $h(v_1) = J'_p$ and $h(v_2) = J'_q = J'_{p+1}$. By the first condition of Definition 14, it follows that $\{h(v_1), h(v_2)\} = \{J'_p, J'_{p+1}\} \in E_\gamma$. This proves the first case. In the remaining cases, $\tilde{e}$ does not belong to L .
2. Assume that v_1 and v_2 are both not edges of L (thus $\tilde{e}$ does not belong to L). Now by Theorem 12 and the second condition of Definition 14, it follows that $\{h(v_1), h(v_2)\} = \{J_p, J_q\} \in E_\gamma$. This proves the second case.
3. Assume that either v_1 or v_2 is an edge of L and that the other one is not an edge of L (thus $\tilde{e}$ does not belong to L). We follow the same line of reasoning as we did in Theorem 12. Without loss of generality, we can assume that v_1 is not an edge of L and that v_2 is an edge of L . Clearly,

$$\emptyset = O_u(i, i) = O_u(i, i_1) \oplus O_u(i_1, i)$$

for each position i_1 . By the structure of L we know that $q = 2$ or $q = \kappa$. Let $q = 2$ ($q = \kappa$, resp.). By Lemma 5 and Lemma 11, we can choose $i_1 \in \{\text{rpos}_{p-1}, \text{rpos}_p\}$ such that $O_u(i_1, i) = \{p\}$. By applying Corollary 4 to L , we get $O_u(i, i_1) = \Pi_u(P)$ with $P = \{2, \dots, p-1\} \cup P'$ ($P = \{p+1, \dots, \kappa\} \cup P'$, resp.) for some $P' \subseteq \{p\}$. By the third (fourth, resp.) condition of Definition 14, it follows that $\{h(v_1), h(v_2)\} = \{J'_2, J_q\} \in E_\gamma$ ($\{h(v_1), h(v_2)\} = \{J'_\kappa, J_q\} \in E_\gamma$, resp.). This proves the third case.

4. Assume that both v_1 and v_2 are edges of L , but $\tilde{e}$ does not belong to L . Again, we follow the same line of reasoning as we did in Theorem 12. Without loss of generality, we can assume that $p \leq q$. By the structure of L , we know that $p = 2$ and $q = \kappa > 3$. By applying Corollary 4 to L , we get $\emptyset = O_u(i, i) = \Pi_u(P)$ with $P = \{2, \dots, \kappa\}$. By the fifth condition of Definition 14, it follows that $\{h(v_1), h(v_2)\} = \{J'_2, J'_\kappa\} \in E_\gamma$. This proves the last case.

This proves the forward implication.

We now prove the reverse implication $\{v_1, v_2\} \in E_\gamma \Rightarrow \{h^{-1}(v_1), h^{-1}(v_2)\} \in E_u$, where h^{-1} , the inverse of h , is given by:

- $h^{-1}(J_p)$ is the unique $v \in V_u$ with $f_u(v) = p$ that is not an edge of L ,
- $h^{-1}(J'_p)$ is the unique $v \in V_u$ with $f_u(v) = p$ that is an edge of L ,

for $2 \leq p \leq \kappa$. Let $e \in E_\gamma$. We consider each of the five types of edges in the definition of reduction graph of an overlap graph.

1. Assume e is of the first type. Then $e = \{J'_p, J'_{p+1}\}$ for some p with $2 \leq p < \kappa$. Since $h^{-1}(J'_p)$ is the desire edge of L with both vertices labelled by p and $h^{-1}(J'_{p+1})$ is the desire edge of L with both vertices labelled by $p+1$, it follows, by the definition of root subgraph, that $h^{-1}(J'_p)$ and $h^{-1}(J'_{p+1})$ are connected by a reality edge in L . Thus, we have $\{h^{-1}(J'_p), h^{-1}(J'_{p+1})\} \in E_u$. This proves the reverse implication when e is of the first type (in Definition 14).
2. Assume e is of the second type. Then $e = \{J_p, J_q\}$ for some p and q with $2 \leq p < q \leq \kappa$ and $\Pi_u(P) = \Pi_\gamma(P) = \{p, q\}$ with $P = \{p+1, \dots, q-1\} \cup P'$ for some $P' \subseteq \{p, q\}$. By Theorem 12, there is a reality edge $\{w_1, w_2\}$ in $\mathcal{R}_u$, such that w_1 has label p and w_2 has label q and both are not vertices of L . By the definition of cps, we have a $\{w'_1, w'_2\} \in E_u$ such that $f_u(w'_1) = p$ ($f_u(w'_2) = q$, resp.) and w'_1 (w'_2 , resp.) is not an edge of L . Therefore $w'_1 = h^{-1}(J_p)$ and $w'_2 = h^{-1}(J_q)$. This proves the reverse implication when e is of the second type.
3. The last three cases are proved similarly.

This proves the reverse implication.

Altogether, we have shown that h is an isomorphism from R_u to $\mathcal{R}_\gamma$. ■

Example 10

Consider again realistic string $u = 72673456\bar{3}245$ from Example 9 (and Example 3). The reduction graph $\mathcal{R}_\gamma$ of the overlap graph of u is given in Figure 5.11. Recall that the reduction graph $\mathcal{R}_u$ of u is given in Figure 5.4. It is easy to see that after applying cps to $\mathcal{R}_u$ one obtains a graph that is indeed isomorphic to $\mathcal{R}_\gamma$.

Formally, we have not yet constructed (up to isomorphism) the reduction graph $\mathcal{R}_u$ of a realistic string u from its overlap graph. We have “only” constructed $\text{cps}(\mathcal{R}_u)$ (up to isomorphism). However, it is clear that $\mathcal{R}_u$ can easily be obtained from $\text{cps}(\mathcal{R}_u)$ (up to isomorphism) by considering the edges as reality edges and replacing every vertex by a desire edge of the same label.

5.8 Consequences

We can now apply our main theorem, Theorem 15, to carry over results that rely on the notion of reduction graph for legal strings. To illustrate this, we characterize successfulness for realistic overlap graphs in any given $S \subseteq \{Gnr, Gpr, Gdr\}$. To accomplish this, we also use results from [13] (or Chapter 13 in [12]). The notions of successful reduction, string negative rule and graph negative rule used in this section are defined in [12].

First we restate a theorem of [6].

Theorem 16

Let u be a legal string, and N be the number of components in $\mathcal{R}_u$. Then every successful reduction of u has exactly $N - 1$ string negative rules.

Due to the “weak equivalence” of the string pointer reduction system and the graph pointer reduction system, proved in Chapter 11 of [12], we can, using Theorem 15, restate Theorem 16 in terms of graph reduction rules.

Theorem 17

Let γ be a realistic overlap graph, and N be the number of components in $\mathcal{R}_\gamma$. Then every successful reduction of γ has exactly $N - 1$ graph negative rules.

As an immediate consequence we get the following corollary. It provides an answer to an open problem formulated in Chapter 13 in [12]: to provide a graph theoretic characterization of successfulness in $\{Gpr, Gdr\}$. However, note that our answer is only for the case where γ is a *realistic* overlap graph.

Corollary 18

Let γ be a realistic overlap graph. Then γ is successful in $\{Gpr, Gdr\}$ iff $\mathcal{R}_\gamma$ is connected.

Example 11

Every successful reduction of the overlap graph of Example 9 has exactly one graph negative rule. For example $\mathbf{gnr}_2 \mathbf{gpr}_4 \mathbf{gpr}_5 \mathbf{gpr}_7 \mathbf{gpr}_6 \mathbf{gpr}_3$ is a successful reduction of this overlap graph.

With the help of [13] (or Chapter 13 in [12]) and Corollary 18, we are ready to complete the characterization of successfulness for realistic overlap graphs in any given $S \subseteq \{Gnr, Gpr, Gdr\}$.

Theorem 19

Let γ be a realistic overlap graph. Then γ is successful in:

- $\{Gnr\}$ iff γ is a discrete graph with only negative vertices.
- $\{Gnr, Gpr\}$ iff each component of γ that consists of more than one vertex contains a positive vertex.
- $\{Gnr, Gdr\}$ iff all vertices of γ are negative.

- $\{Gnr, Gpr, Gdr\}$.
- $\{Gdr\}$ iff all vertices of γ are negative and $\mathcal{R}_\gamma$ is connected.
- $\{Gpr\}$ iff each component of γ contains a positive vertex and $\mathcal{R}_\gamma$ is connected.
- $\{Gpr, Gdr\}$ iff $\mathcal{R}_\gamma$ is connected.

Proof

The cases where $Gnr \in S$ (cf. the first four cases in the theorem) are known from [13], and case $\{Gpr, Gdr\}$ holds by Corollary 18. Case $\{Gdr\}$ ($\{Gpr\}$, resp.) is obtained by combining the results of cases $\{Gnr, Gdr\}$ ($\{Gnr, Gpr\}$, resp.) and $\{Gpr, Gdr\}$. Note that if γ has an isolated negative vertex, then $\mathcal{R}_\gamma$ is not connected. ■

5.9 Discussion

We have shown a way to directly construct the reduction graph of a realistic string (up to isomorphism) from its overlap graph γ . This allows one to (directly) determine the number n of graph negative rules that are necessary to reduce γ successfully. Surprisingly, although a lot of structural information is lost in the overlap graph (compared to a string representation), this information can be retrieved from the overlap graph via its reduction graph. The main result allows for a complete characterization of successfulness of γ in any given $S \subseteq \{Gnr, Gpr, Gdr\}$ by extending [13] for the cases where $Gnr \notin S$. It remains an open problem to find a (direct) method to determine this number n for overlap graphs γ in general (not just for realistic overlap graphs).

Bibliography

- [1] A. Bergeron, J. Mixtacki, and J. Stoye. On sorting by translocations. In S. Miyano et al., editors, *RECOMB*, volume 3500 of *Lecture Notes in Computer Science*, pages 615–629. Springer, 2005.
- [2] R. Brijder and H.J. Hoogeboom. Characterizing reduction graphs for gene assembly in ciliates. In T. Harju, J. Karhumäki, and A. Lepistö, editors, *Developments in Language Theory (DLT) 2007*, volume 4588 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2007.
- [3] R. Brijder, H.J. Hoogeboom, and M. Muskulus. Applicability of loop recombination in ciliates using the breakpoint graph. In M.R. Berthold et al., editors, *CompLife '06*, volume 4216 of *Lecture Notes in Computer Science*, pages 97–106. Springer, 2006.
- [4] R. Brijder, H.J. Hoogeboom, and M. Muskulus. Strategies of loop recombination in ciliates. *Discrete Applied Mathematics*, 156:1736–1753, 2008.
- [5] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. The breakpoint graph in ciliates. In M.R. Berthold et al., editors, *CompLife '05*, volume 3695 of *Lecture Notes in Computer Science*, pages 128–139. Springer, 2005.
- [6] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. Reducibility of gene patterns in ciliates using the breakpoint graph. *Theoretical Computer Science*, 356:26–45, 2006.
- [7] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. From micro to macro: How the overlap graph determines the reduction graph in ciliates. In E. Csuhaj-Varjú and Z. Ésik, editors, *Fundamentals of Computation Theory (FCT) 2007*, volume 4639 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 2007.
- [8] A.R.O. Cavalcanti, T.H. Clarke, and L.F. Landweber. MDS_IES_DB: a database of macronuclear and micronuclear genes in spirotrichous ciliates. *Nucleic Acids Res.*, 33:D396–D398, 2005.

-
- [9] M. Daley, O.H. Ibarra, and L. Kari. Closure and decidability properties of some language classes with respect to ciliate bio-operations. *Theoretical Computer Science*, 306(1-3):19–38, 2003.
 - [10] M. Daley and L. Kari. Some properties of ciliate bio-operations. In M. Ito and M. Toyama, editors, *Developments in Language Theory*, volume 2450 of *Lecture Notes in Computer Science*, pages 116–127. Springer, 2002.
 - [11] A. Ehrenfeucht, T. Harju, I. Petre, D.M. Prescott, and G. Rozenberg. Formal systems for gene assembly in ciliates. *Theoretical Computer Science*, 292:199–219, 2003.
 - [12] A. Ehrenfeucht, T. Harju, I. Petre, D.M. Prescott, and G. Rozenberg. *Computation in Living Cells – Gene Assembly in Ciliates*. Springer Verlag, 2004.
 - [13] A. Ehrenfeucht, T. Harju, I. Petre, and G. Rozenberg. Characterizing the micronuclear gene patterns in ciliates. *Theory of Computing Systems*, 35:501–519, 2002.
 - [14] A. Ehrenfeucht, I. Petre, D.M. Prescott, and G. Rozenberg. Circularity and other invariants of gene assembly in ciliates. In M. Ito et al., editors, *Words, Semigroups, and Transductions*, pages 81–97. World Scientific, Singapore, 2001.
 - [15] A. Ehrenfeucht, I. Petre, D.M. Prescott, and G. Rozenberg. String and graph reduction systems for gene assembly in ciliates. *Mathematical Structures in Computer Science*, 12:113–134, 2002.
 - [16] S. Hannenhalli and P. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC '95)*, pages 178–189, 1995.
 - [17] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *J. ACM*, 46(1):1–27, 1999.
 - [18] T. Harju, C. Li, I. Petre, and G. Rozenberg. Parallelism in gene assembly. In C. Ferretti et al., editors, *DNA 10*, volume 3384 of *Lecture Notes in Computer Science*, pages 138–148. Springer, 2004.
 - [19] T. Harju, I. Petre, and G. Rozenberg. Formal properties of gene assembly: Equivalence problem for overlap graphs. In N. Jonoska, G. Paun, and G. Rozenberg, editors, *Aspects of Molecular Computing*, volume 2950 of *Lecture Notes in Computer Science*, pages 202–212. Springer, 2004.
 - [20] L.F. Landweber and L. Kari. The evolution of cellular computing: Nature’s solution to a computational problem. *Biosystems*, 52:2–13, 1999.

-
- [21] P.A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. MIT Press, 2000.
 - [22] D.M. Prescott and M. DuBois. Internal eliminated segments (IESs) of oxytrichidae. *J. Euk. Microbiol.*, 43:432–441, 1996.
 - [23] J.C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

Part II

Membrane Computing

Chapter 6

Membrane Systems with Proteins Embedded in Membranes

Abstract

Membrane computing is a biologically inspired computational paradigm. Motivated by brane calculi we investigate membrane systems which differ from conventional membrane systems by the following features: (1) biomolecules (proteins) can move through the regions of the systems, and can attach onto (and de-attach from) membranes, and (2) membranes can evolve depending on the attached molecules. The evolution of membranes is performed by using rules that are motivated by the operation of pinocytosis (the pino rule) and the operation of cellular dripping (the drip rule) that take place in living cells.

We show that such membrane systems are computationally universal. We also show that if only the second feature is used then one can generate at least the family of Parikh images of the languages generated by programmed grammars without appearance checking (which contains non-semilinear sets of vectors).

If, moreover, the use of pino/drip rules is non-cooperative (i.e., not dependent on the proteins attached to membranes), then one generates a family of sets of vectors that is strictly included in the family of semilinear sets of vectors.

We also consider a number of decision problems concerning reachability of configurations and boundness.

6.1 Introduction

Membrane computing is a biologically inspired computational paradigm introduced by Gh. Păun in 1998, [20]. The model is based on a hierarchical structure of nested membranes, inspired by the structure of living cells. In each region

(enclosed by a membrane) some objects are present, modeling the presence of molecules inside the compartments of living cells. Moreover, each region has an associated set of multiset rewriting rules. These rules are motivated by chemical reactions that occur inside the regions of living cells. Membranes play a crucial role in living cells: the cell membrane separates, and hence protects the cell from its environment and the inner membranes delimit the structure of various organelles of the cell, e.g., the nuclear membrane separates the nucleus from the rest of the cell.

Membranes are not only “containers” but they also regulate the flow of molecules into and out of the cell. This is facilitated by proteins that are embedded in membranes and which provide channels for the transport of molecules through membranes.

In brane calculi, presented in [5], several operations (*pino*, *exo*, *phago*, *mate*, *drip*, *bud*) involving membranes with embedded proteins are considered and formalized in the framework of process calculi. The important difference with membrane computing is that the evolution of the system happens *on* the membranes and *not inside* the compartments (regions) delimited by them. The computational power of several brane calculi operations has been investigated in [4] where universality has been obtained for systems using *phago* and *exo*. In [6] these operations from brane calculi have been represented in the membrane computing framework and then studied by using tools from formal language theory.

In this chapter we investigate operations involving membranes with embedded proteins, but we also add the ability of proteins to attach/de-attach to/from the membranes, and also to move through the membranes. Hence, in our case, the evolution of the system takes place both on the membranes and inside the regions, which is natural from a biological point of view.

More specifically, we consider *protein-membrane rules* – rules that modify the structure of (the membranes of) the system where the modifications are based on the multisets of proteins embedded in the membranes (we say that such multisets *mark* the membranes). In particular, we consider the *pino* and *drip* rules inspired by the operation of *pinocytosis* and the operation of cellular *dripping*, respectively. Both pinocytosis and dripping split off a membrane from another membrane, however, in pinocytosis, this new (empty) membrane is found inside the original membrane, while in dripping, this new membrane is found outside the original membrane. We also use *protein movement rules*, that model the attachment, de-attachment and movement of the proteins. Also these rules are applied according to the proteins marking the involved membranes. The protein movement rules do not change the membrane structure of the system, but they can change the multisets of embedded proteins marking the membranes of the system.

The chapter is structured in the following way. In Section 6.2 we provide preliminaries concerning formal languages, recalling in particular the definition of programmed grammars often used in the proofs. In Section 6.3 we recall the formal definition of *pino* and *drip* rules, and introduce the protein movement rules, and in Section 6.4 we introduce membrane systems based on these rules – the model

is called *membrane system with marked membranes, protein-membrane rules, and protein movement rules*, abbreviated as P_{pp} system.

In Section 6.5, we investigate the computational power of P_{pp} systems which use only protein movement rules, and in Section 6.6 of P_{pp} systems using only pino (or drip) rules. In Section 6.7, we discuss P_{pp} systems using both types of rules. In Section 6.8 we prove several decidability results concerning reachability of configurations and boundness of P_{pp} systems with pino, drip rules, and protein movement rules. In the last section we discuss the results obtained in this chapter and formulate a number of research directions.

6.2 Preliminaries

We will briefly recall the main notions and results of formal language theory used in this chapter. For more details the reader can consult standard books, such as [16], [25], [11], and the handbook [24].

Given a set A , we denote by $|A|$ its cardinality and by $\mathbb{P}(A)$ the power set of A . The empty set is denoted by $\emptyset$.

As usual, an *alphabet* V is a finite set of symbols. By V^* we denote the set of all strings over V . The empty string is denoted by λ . The *length* of a string $w \in V^*$ is denoted by $|w|$, while the number of occurrences of $a \in V$ in w is denoted by $|w|_a$. For a language $L \subseteq V^*$, the set $\text{length}(L) = \{|w| \mid w \in L\}$ is called the *length set* of L . Given a string w , a string u is a *subword* of w if there exist two strings x, y , possibly empty, such that $w = xuy$. The string u is a *scattered subword* of w if and only if there exist strings $x_1, \dots, x_k$, and $y_0, \dots, y_k$, possibly empty, such that $u = x_1 \cdots x_k$, and $w = y_0 x_1 y_1 \cdots x_k y_k$. We use $\text{Sub}(w)$ to denote the set of all subwords of w , while $\text{Scub}(w)$ denotes the set of the scattered subwords of w .

Given an alphabet $V = \{a_1, a_2, \dots, a_n\}$, with every string $w \in V^*$ we can associate the *Parikh vector* $\Psi_V(w) = (|w|_{a_1}, |w|_{a_2}, \dots, |w|_{a_n})$, where the ordering $(a_1, \dots, a_n)$ of V is assumed. Given a language $L \subseteq V^*$, the *Parikh image* of L is defined as $\Psi_V(L) = \{\Psi_V(w) \mid w \in L\}$.

If FL is a family of languages, then $PsFL$ denotes the family of Parikh images of languages in FL (w.r.t. a given alphabet V), and NFL denotes the family of length sets of languages in FL . Note that each $L \in PsFL$ is a set of vectors with a fixed dimension. We denote by FIN , REG , CF , CS , and RE the family of finite, regular, context-free, context-sensitive, and recursively enumerable languages, respectively. Accordingly, the family of Parikh images of languages in RE is denoted by $PsRE$ (this is the family of all recursively enumerable sets of vectors of natural numbers). The family of all recursively enumerable sets of natural numbers is denoted by NRE . As usual, two language generating/accepting devices are called *equivalent* if they generate/accept the same language.

A *generalized sequential machine* (in short *gsm*) is a system $\Gamma = (K, V_1, V_2, s_0, F, \delta)$, where K is a finite set of states, $s_0 \in K$ is the initial state, $F \subseteq K$ the set of final states, and V_1, V_2 are the input and output alphabet, respectively. The transition function δ is defined by $\delta : K \times V_1 \longrightarrow \mathbb{P}(V_2^* \times K)$. For $s, s' \in K, a \in$

$V_1, y \in V_1^*, x, z \in V_2^*$ we write $(x, s, ay) \mapsto (xz, s', y)$ if $(z, s') \in \delta(s, a)$. Then, for $w \in V_1^*$, we define $\Gamma(w) = \{z \in V_2^* \mid (\lambda, s_0, w) \mapsto^* (z, s, \lambda), s \in F\}$. The mapping Γ is extended in natural way to languages over V_1 .

A context-free *programmed grammar with appearance checking* is a construct $G = (N, T, S, P)$, where N (T , resp.) is a finite set of nonterminals (terminals, resp.), $S \in N$ is the start symbol, and P is a finite set of productions of the form $(b : A \rightarrow x, E_b, F_b)$, where b is a label, $A \rightarrow x$ with $A \in N$ and $x \in (N \cup T)^*$ is a context-free production, and E_b, F_b are two sets of labels of productions of G (E_b is called the *success field* and F_b the *failure field* of the production). A production $(b : A \rightarrow x, E_b, F_b)$ is applied as follows: if A is present in the sentential form, then the production $A \rightarrow x$ is applied and the next production is chosen from those with the labels in E_b , otherwise, the sentential form remains unchanged and we choose the next production from the set of productions labeled by some element of F_b . A derivation step is denoted by $\Rightarrow$ while $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$. If no failure field is given for any of the productions, then we obtain a programmed grammar *without appearance checking*.

We denote the set of labels as $Lab(G) = \{b \mid \text{there exists } (b : A \rightarrow x, E_b, F_b) \in P\}$. Also, for $X \in N$, we denote $\{b \mid \text{there exists } (b : X \rightarrow x, E_b, F_b) \in P\}$ by $l_G(X)$, or $l(X)$ for short.

The language generated by a grammar G is denoted by $L(G)$. By PR we denote the family of languages generated by programmed grammars without appearance checking, and by PR_{ac} we denote the family of languages generated by programmed grammars with appearance checking. Proofs of the following results can be found in [11].

Lemma 1

$CF \subset PR \subset PR_{ac} = RE$.

In *pure* programmed grammars there is no distinction between terminals and nonterminals. Consequently, the language generated by a pure programmed grammar is defined as the set of all strings that can be generated from the axiom, hence the set of all sentential forms. The family of languages generated by pure programmed grammars without appearance checking is denoted by pPR . It is easy to prove (in a constructive way) that

Lemma 2

$pPR \subset PR$.

The following normal form for programmed grammars, referred to as the l_h -normal form, will be useful in this chapter.

Lemma 3

For any programmed grammar G (with appearance checking) there exists an equivalent programmed grammar G' (with appearance checking, respectively) such that there is a unique initial production (with label l_0) and a unique final production $Z \rightarrow \lambda$ (with label l_h) in G' .

Proof

Consider an arbitrary programmed grammar $G = (N, T, P, S)$. We recall that $l(S)$ is the set of labels corresponding to productions having S at the left-hand side. Given a set of labels L , we use L' to denote the set of the primed version of the labels in L .

Let $G' = (N', T, P', S')$, where $N' = N \cup \{S', Z\}$, $S', Z \notin N$, $Lab(G') = Lab'(G) \cup \{l_0, l_h\}$, and P' consists of the following productions:

$$\{(l_0 : S' \rightarrow ZS, l'(S), \emptyset), (l_h : Z \rightarrow \lambda, \emptyset, \emptyset)\} \cup \{(l'_i : A \rightarrow \alpha, E'_{l_i} \cup \{l_h\}, F'_{l_i} \cup \{l_h\}) \mid (l_i : A \rightarrow \alpha, E_{l_i}, F_{l_i}) \in P\}.$$

It is easily seen that $L(G) = L(G')$, and that the final production in any successful derivation in G' is the one labeled by l_h (deletion of the nonterminal Z). The same construction works for both programmed grammars with or without appearance checking (in this last case the failure fields are removed from the productions in P'). It is worth to notice that the unsuccessful derivations in G' are of the following types: $S' \Rightarrow^* Z(N \cup T)^*$ or $S' \Rightarrow^* Z(N \cup T)^*N(N \cup T)^* \Rightarrow (N \cup T)^*N(N \cup T)^*$, if G' is without appearance checking while only of the second type if G' is with appearance checking. ■

We assume the reader to be familiar with the basic notions of membrane computing, see, e.g., [21].

6.3 Operations for Marked Membranes

In [5] several membrane operations involving membranes and embedded proteins have been modeled in the framework of process calculi. In [6] these operations have been expressed in the framework of membrane systems.

We will briefly recall these operations, however in a slightly modified form: while in [5] and [6] a region (enclosed by a membrane) can contain other membranes but not objects, we allow a region to contain objects.

As usual in membrane computing, a membrane is represented by a pair of square brackets, $[]$. To each membrane $[]$ we associate a multiset u (over a certain alphabet V) and this is denoted by $[]_u$. We say that the membrane is *marked* with u (u is called a *marking*). The objects of V are called *proteins* or, simply, *objects*. The contents of a membrane can consist of proteins and/or other membranes.

The *protein-membrane rules* over V are of the following form (the subscript i stands for *internal*, e for *external*):

$$\begin{aligned} pino_i &: []_{uav} \rightarrow [[]_{ux}]_v, \\ pino_e &: []_{uav} \rightarrow [[]_v]_{ux}, \\ drip &: []_{uav} \rightarrow []_{ux} []_v. \end{aligned}$$

where $a \in V$, and $u, x, v \in V^*$ (thus the restriction of having the right-hand sides of the rules nonempty, as in [6], has been relaxed here). If $uv = \lambda$, then we have a *non-cooperative rule*; we add the prefix (*ncoo*) to denote it. Thus $(ncoo)pino_i : []_a \rightarrow [[]_x]$ is a non-cooperative $pino_i$ rule.

The described rules are applicable to any membrane whose marking *includes* the multiset indicated on the left-hand side of the rules; all the proteins not specified in the rules are not affected by the use of the rules, but they are *randomly distributed* between the two resulting membranes. When using any rule of any type, we say that the membrane from its left-hand side is *involved* in the rule; the membrane involved is “consumed” while the membranes from the right-hand side of the rule are “produced”. Similarly, the protein a specified on the left hand side of the rules is consumed, and it is replaced by the multiset of proteins x (that might be empty).

After the application of a $pino_i$ or $pino_e$ rule, the contents of the consumed membrane is moved into the region of the created external membrane (thus, membrane $[]_v$ for $pino_i$ and membrane $[]_{ux}$ for $pino_e$), and after the application of a *drip* rule, the contents of the consumed membrane is moved into the region of the produced membrane $[]_v$.

We also define rules that can attach/de-attach proteins to/from the membranes, and rules to move the proteins through the membranes of the system. The *protein movement rules* over V can have one of the following forms (the subscript i stands for *inside*, o for *outside*):

$$\begin{aligned} attach_i &: [a]_u \rightarrow []_{ua}, \quad attach_o : []_ua \rightarrow []_{ua}, \\ de-attach_i &: []_{ua} \rightarrow [a]_u, \quad de-attach_o : []_{ua} \rightarrow []_ua, \\ move_{out} &: [a]_u \rightarrow []_ua, \\ move_{in} &: []_u a \rightarrow [a]_u, \end{aligned}$$

with $a \in V$, $u \in V^*$.

The effect of the rules $attach_i$ and $attach_o$ is to attach the protein a to the corresponding membrane if the marking of the membrane *includes* u .

The rules $move_{out}$ ($move_{in}$) move the protein a outside (inside, resp.) if the marking of the corresponding membrane *includes* u . We use $prot$ to denote the set of protein movement rules.

6.4 Membrane Systems with Marked Membranes

In this section we define membrane systems (also called P systems) having membranes marked with multisets of proteins, and using the protein-membrane rules and the protein movement rules introduced in Section 6.3.

Formally, a *membrane system with marked membranes, protein-membrane rules, and protein movement rules*, in short P_{pp} system, is a construct

$$\Pi = (V, \mu, u_1, \dots, u_m, R, F),$$

- V is a finite, nonempty alphabet of proteins;
- μ is a membrane structure with $m \geq 1$ membranes;
- $u_1, \dots, u_m \in V^*$ are the markings of the m membranes of μ at the beginning of the computation (the *initial markings* of Π);
- R is a finite set of protein-membrane rules and protein movement rules over the alphabet V ;
- $F \subseteq V$ is the set of *protein-flags*, simply called *flags* (marking the *output membranes*).

We will also use V_Π , μ_Π , R_Π , and F_Π to denote V , μ , R , and F respectively.

A *configuration* of Π consists of a membrane structure, the markings of the membranes, and the multisets of proteins present inside the regions. In what follows, configurations are denoted by writing the markings as subscripts of the right-hand parentheses which identify the membranes, e.g., $[[]_{ab}[aaa]_b[]_{bb}]_a$ is an example of a configuration.

We suppose that in the *initial configuration* the regions are empty, thus the initial configuration is defined by μ and $u_1, \dots, u_m$.

As standard for membrane systems, we assume the existence of a global clock which marks the timing of steps (single *transitions*) for the whole system.

A single transition of Π from a configuration to a new one is performed by applying, to each membrane of the system, either (i) the protein movement rules in the nondeterministic maximally parallel manner, or (ii) one of the protein-membrane rules.

The choice between using protein movement rules or using a protein-membrane rule, for each membrane, is done in a nondeterministic way if both types of rules can be applied for a given membrane. A membrane remains unchanged (only) if no rules can be applied to it.

The application in the nondeterministic maximally parallel manner of the protein movement rules means that, for the chosen membrane, the proteins (the ones marking the membrane and those present in the enclosed region) are assigned with the rules in such a way that, after the assignment is done, no other protein movement rule is applicable to the proteins that have no rules assigned to them. If a protein can be used by several rules, then it is assigned to one of them in a nondeterministic way.

As usual, a sequence of transitions forms a *computation*. A computation which starts from the initial configuration is *successful* if it halts, that is, it reaches a *halting configuration*, i.e., a configuration where no rule can be applied, anywhere in the system. In the halting configuration we consider the *output membranes* – these are membranes whose markings contain at least one flag from F .

Then, the *result* of a successful computation is the set of vectors describing the multiplicities of proteins present in the markings of the output membranes. Owing to the nondeterminism in the choice of rules, one can get a set of (successful)

computations, and thus a set of results.

Collecting all the results, for all possible successful computations, we get *the set of vectors generated by Π* , and denoted by $Ps(\Pi)$.

Since a halting configuration in a P_{pp} system can have several membranes marked with F , we can have more than one output membrane. Therefore, the output of a successful computation is a finite family of vectors (each vector corresponding to an output membrane). This differs from assigning the output in “standard” membrane systems, where we have only one output vector. However since the set of vectors $Ps(\Pi)$ generated by a P_{pp} system is taken over the union of results of all successful computations, this difference “disappears” in the sense that we can compare the output $Ps(\Pi)$ with the output of “standard” membrane systems.

For $\alpha \in \{pino_i, pino_e, drip, (ncoo)pino_i, (ncoo)pino_e, (ncoo)drip\}$, and $m \geq 1$, we denote the class of P_{pp} systems using protein-membrane rules of type α , using protein movement rules, and having at most m membranes, by $PP_m(\alpha, prot)$ (α or $prot$ are removed if the corresponding rules are not used). Therefore, the family of sets of vectors generated by P_{pp} systems from $PP_m(\alpha, prot)$ is denoted by $PsPP_m(\alpha, prot)$ (again, α or $prot$ are removed if the corresponding rules not used). If m is substituted by $*$, then the number of membranes considered is arbitrary.

Since one cannot mark the empty multiset by a flag, we consider the equality of families of multisets modulo the empty multiset, i.e., if two families differ only by the empty multiset, then we consider them to be equal.

A configuration of a P_{pp} system Π that can be reached by a (possibly empty) sequence of transitions, starting from the initial configuration, is called *reachable*. A multiset w of proteins is a *reachable marking* for Π if there exists a reachable configuration of Π which contains a membrane marked by w .

6.5 Preliminary Results

We begin with some preliminary results that follow directly from the definitions and from the Turing-Church thesis.

Theorem 4

$$\begin{aligned} PsPP_*(\alpha, prot) &\subseteq PsRE, \quad PsPP_*(\alpha) \subseteq PsPP_*(\alpha, prot). \\ PsPP_*((ncoo)\alpha, prot) &\subseteq PsPP_*(\alpha, prot), \\ PsPP_*((ncoo)\alpha) &\subseteq PsPP_*(\alpha), \\ \alpha &\in \{pino_i, pino_e, drip\}. \end{aligned}$$

First we consider P_{pp} systems that use only the protein movement rules. The power of such systems is very restricted, even when there is no bound on the number of membranes to be used.

Theorem 5

$PsPP_*(prot) = PsFIN$.

Proof

The inclusion $PsPP_*(prot) \subseteq PsFIN$ comes from the fact that, using only protein movement rules, it is not possible to increase the total number of objects (and membranes) present in a P_{pp} system during the computation.

On the other hand, the Parikh image of every finite language can be generated by a P_{pp} system from $PP_*(prot)$: in fact, the Parikh image of a finite language can be represented in the initial markings, where each protein is a flag, and actually there is no need to use protein movement rules. Therefore, also the inclusion $PsFIN \subseteq PsPP_*(prot)$ holds and then the theorem follows. ■

6.6 Membrane Systems Using Protein-Membrane Rules

As stated by Theorem 5 the use of only protein movement rules results in a very limited generative power. In this section we turn to the dual situation: the use of protein-membrane rules only.

In this case the membrane structure can change during the computation, but the proteins cannot move through the regions of the system.

First we investigate P_{pp} systems using the non-cooperative versions of the pino and of the drip rules. In this case the power of the system is still very limited: the family of the so generated sets of vectors is strictly included in the family of Parikh images of context-free languages. Then we will study P_{pp} systems using only pino and drip rules; in this case the power of the system increases: one can generate now at least the family of Parikh images of the languages generated by programmed grammars without appearance checking.

First we give an example.

Example 1

Consider the regular language $L = \{a^{2n} \mid n \geq 1\}$. It is easy to show that the Parikh image of L , $\Psi_{\{a\}}(L) = \{2n \mid n \geq 1\}$, cannot be generated by a P_{pp} system Π from $PP_*(ncoo)pino_i$. Indeed, suppose that there is such a Π . Since L is infinite, there is a $x \in L$ with $|x|$ larger than the length of the right-hand side of any $pino_i$ rule of Π . Thus x has some proteins that were not involved in the application of the last $pino_i$ rule. Since during the application of a $pino_i$ rule one such protein could also have moved to the other membrane, we also have $x - 1 \in \Psi_{\{a\}}(L)$, a contradiction.

The previous example illustrates that the “random splitting” of the proteins that are not specified in the applied rule is a feature that can reduce the computational power of the system.

Lemma 6

Let $\alpha \in \{pino_i, pino_e, drip\}$. Then, $PsPP_1((ncoo)\alpha) \subseteq PsCF$ iff $PsPP_*((ncoo)\alpha) \subseteq PsCF$.

Proof

The “if” part of the statement obviously holds. We now prove the “only if” part. Let $\Pi = (V, \mu, u_1, \dots, u_m, R, M) \in PsPP_m((ncoo)\alpha)$ for some $m > 1$. Now, let $\Pi_i = (V, \mu', u_i, R, M) \in PsPP_1((ncoo)\alpha)$ for $i \in \{1, \dots, m\}$, with μ' a single membrane initially marked with u_i . We have $Ps(\Pi) = Ps(\Pi_1) \cup Ps(\Pi_2) \cup \dots \cup Ps(\Pi_m)$ if each Π_i has a halting configuration, and otherwise $Ps(\Pi) = \emptyset$. Since CF is closed under finite union, we have the desired result. ■

Theorem 7

$PsPP_*((ncoo)\alpha) \subset PsCF, \alpha \in \{pino_i, pino_e, drip\}$.

Proof

By Example 7 and Lemma 6 it suffices to show that $PsPP_1((ncoo)\alpha) \subseteq PsCF$. Given a P_{pp} system $\Pi = (V, \mu, u, R, M)$ from $PP_1((ncoo)pino_i)$, we show that one can construct a context-free grammar G such that the Parikh image of $L(G)$ is exactly $Ps(\Pi)$. Note that μ must be a single membrane. Let $Lab(R) = \{r_1, r_2, \dots, r_n\}$ be a labeling of the elements of R with $|R| = n$.

We now construct a context-free grammar $G = (N, T, P, S)$, dependent on Π , with

$$\begin{aligned} N &= \{a \in V \cup \{S\} \mid a \rightarrow \alpha \in P \text{ for some } \alpha\}, \text{ where } V \cap \{S\} = \emptyset, \\ T &= Lab(R) \cup (V - N), \\ P &= \{a \rightarrow r_j x \mid r_j : []_a \rightarrow []_x \in R, 1 \leq j \leq n\} \cup \{S \rightarrow u\}. \end{aligned}$$

In this way G faithfully simulates Π (ignoring the elements of $Lab(R)$), assuming that during each $pino_i$ rule, all proteins move to the inner membrane. We now define a nondeterministic gsm dependent on G which includes the possibility of “random splitting” of proteins (as commented already after Example 7).

It is possible to construct a nondeterministic gsm Γ_G , dependent on G , with input alphabet T and output alphabet T , such that the set of output strings on input $y \in T^*$, denoted by $\Gamma_G(y)$, is

$$\{w'x, w' \mid y = w_1 r_j x w_2, a \rightarrow r_j x \in P, w' \in Scub(w_1 w_2)\} \cup U,$$

where $U = \{u\}$ if $u \in T^+$, and $U = \emptyset$ otherwise. For $y \in L(G)$, each decomposition of y into $w_1 r_j x w_2$ with $r = a \rightarrow r_j x \in P$ corresponds to a derivation in which r is the last production applied. The other symbols, $w_1 w_2$ were nondeterministically distributed to the outer membrane and the inner membrane. Therefore, $w'x$ (w' , resp.) represents the set of markings of the inner (outer, resp.) membrane. In the special case when there is no such decomposition of y , we have $y = u \in T^+$ and u is a marking of a “halting” membrane.

Let h be the morphism which deletes the elements of $Lab(R)$, formally defined by $h(a) = \lambda$ for $a \in Lab(R)$, and $h(a) = a$ for $a \in T - Lab(R)$. Now,

$\Psi_T(L)$ with $L = h(\Gamma_G(L(G)))$ precisely represents the set of multisets marking the reachable halting membranes of Π . Since context-free languages are closed under gsm mapping and applications of morphisms, we have $L \in CF$. We now only need to select those multisets of L which contain proteins of M . Therefore, $\Psi_T(L') = Ps(\Pi)$ for context-free $L' = L \cap V^*MV^*$ (context-free languages are closed under intersection with regular languages). ■

The computational power of this class increases when one uses cooperative *pino_i*, *pino_e* or *drip* rules. In this case the systems can generate at least the family of Parikh images of languages generated by programmed grammars without appearance checking – it is known that $PsPR$ strictly contains $PsCF$ because it also contains non-semilinear vectors of natural numbers (see [11] for further details).

Formally, we have the following result.

Theorem 8

$PsPR \subseteq PsPP_*(\alpha)$, $\alpha \in \{pino_i, pino_e, drip\}$.

Proof

Consider a programmed grammar $G = (N, T, P, S)$ without appearance checking in the l_h -normal form (see Lemma 3).

We construct a P_{pp} system Π from $PP_*(pino_i)$ that generates exactly the Parikh image of $L(G)$ and it is defined as follows:

$$\Pi = (V, \mu, u_1, u_2, R, F),$$

where

- $V = N \cup T \cup \{E\} \cup Lab(G) \cup \{\#\}$,
- $\mu = [[]_{ESl_0}]_\lambda$,
- $F = T$.

The *pino* rules in R are grouped as follows (the grouping is done according to their intended use):

1. (simulation of the programmed grammar productions),
 $[]_{EAi} \rightarrow [[]_{Exj}]_i$, for $(i : A \rightarrow x, E_i) \in P, j \in E_i, i \neq l_h$,
2. (used if a production of G cannot be applied),
 $[]_A \rightarrow [[]_{E\#\#}]$, $A \in N$,
3. (used for non-halting),
 $[]_{E\#\#} \rightarrow [[]_{E\#\#}]_\#$,
4. (used to keep the symbols from a sentential form on the same membrane),
 $[]_{Xi} \rightarrow [[]_{E\#\#}]_i$, for $X \in (N \cup T), i \in Lab(G)$,

5. (used to halt the computation),
 $[]_{EZl_h} \rightarrow [[]]_{El_h}$.

The so-constructed system Π simulates G in the following way. The structure of the system contains at any time during the computation a unique innermost membrane. The marking of this membrane contains the object E (except in the last step of the computation), the label of the next production of G to simulate (at the beginning this label is l_0), and the objects corresponding to the current sentential form (at the beginning of a computation of Π only the object S).

The objects from N are called nonterminal objects, while the objects from T are called terminal objects.

The simulation of the application of a production in G is done by using one of the rules in group 1.

The object A is rewritten to x if $(i : A \rightarrow x, E_i) \in P$.

Also the label j of the next production is produced while the old one, i , is stored on the created external membrane. Note that we should “trash” the computation of Π if no productions of G can be simulated and there are nonterminal objects attached to the innermost membrane of Π . This is accomplished by the rules in groups 2 and 3.

Indeed, if no rule from group 1 can be applied, then a rule of group 2 must be applied, because of the maximal parallelism. If this rule is applied, then the membrane $[]_{E\#\#}$ is created and the rule in group 3 is applied forever; thus the computation does not halt – the computation is “trashed”.

We have to guarantee that, after a rule of group 1 is applied (simulating the application of a production from G), the objects not modified in the sentential form do not go to the created external membrane. To this aim the rules of group 4 are used. In fact, if any object of the sentential form is attached to a membrane not containing the object E (that is always attached only to the innermost membrane), then the computation of Π never halts.

To make the computation halting, the symbols E and l_h must be removed from the innermost membrane, and the nonterminal object Z must be erased. To this aim we use the rules of group 5. Notice that this should be done only when the sentential form is composed by only terminal objects. In fact, if this is not true, then in the next step, a rule of group 2 is applied, and then the computation will never halt.

Thus, from the above explanation, it follows that, any successful derivation of G producing w can be simulated by a successful computation in Π halting in a configuration containing a unique innermost membrane, which is also the unique output membrane which is marked by the multiset $\Psi_V(w)$.

On the other hand, unsuccessful derivations of G can be of the type $S \Rightarrow^* Z(N \cup T)^*$ or of the type $S \Rightarrow^* Z(N \cup T)^* N(N \cup T)^* \Rightarrow (N \cup T)^* N(N \cup T)^*$. The simulation in Π of these two types of derivations results in non-halting computations.

Therefore $Ps(\Pi)$ is exactly the Parikh image of the language generated by the grammar G .

The proof given can be easily adapted using only *pino_e* or using only *drip* rules. Therefore the theorem follows. ■

6.7 Using Protein-Membrane and Protein Movement Rules

We will investigate now membrane systems using both protein-membrane rules and protein movement rules. As we will demonstrate the ability to attach, detach, and move proteins across the system in a controlled fashion increases the generative power of the systems.

The first indications of the increased generative power is given by Theorem 9: P_{pp} systems from $PsPP_*((ncoo)\alpha, prot)$, $\alpha \in \{pino_i, pino_o, drip\}$, can generate at least the family of Parikh images of context-free languages (compare this result with Theorem 7).

Theorem 9

$PsCF \subseteq PsPP_*((ncoo)\alpha, prot), \alpha \in \{pino_i, pino_o, drip\}$.

Proof

Given a context-free grammar $G = (N, T, P, S)$ one can construct a P_{pp} system Π from $PP_*((ncoo)pino_i, prot)$ such that $Ps(\Pi)$ is exactly the Parikh image of $L(G)$. Without loss of generality we suppose that each nonterminal is at the left-hand side of at least one production of the grammar.

We construct $\Pi = (V, \mu, u_1, u_2, R, F)$ with $V = N \cup T \cup \{t, E\}$, $F = T$, and $\mu = [[]_{St}]_E$.

The rules of R are grouped according to their intended use:

1. (*Pino rules*),
 $[]_a \rightarrow [[]_{tx}]$ for $a \rightarrow x \in P$,
2. (*protein movement rules – movement of terminal objects*),
 $[]_{ta} \rightarrow []_t a$, for $a \in T$,
 $[a] \rightarrow [] a$, for $a \in T$,
 $[] a \rightarrow [a]$, for $a \in T$,
 $[a]_E \rightarrow []_a E$, for $a \in T$,
 $[]_E \rightarrow [] E$,
3. (*protein movement rules – movement for non-halting*),
 $[]_{tt} \rightarrow []_t t$,
 $[t] \rightarrow [] t$,
 $[] t \rightarrow [t]$.

Intuitively, Π simulates the context-free productions of G using the *pino* rules, and when terminal objects are created, they are collected on the skin membrane. We will show that during the computation of Π the membrane structure is such that the marking of the skin membrane contains exactly one copy of E and no copies of t , while the markings of the other membranes contain exactly one copy of t and no copies of E .

The system Π works in the following way. The rules of group 1 simulate the rewriting in G . Each time a pino rule is applied, then the “special” object t is also attached to the created internal membrane. If a membrane with two (or more) objects t attached is produced, then the computation will not halt because at least a membrane with a marking containing two objects t is produced, and then the rules from group 3 can be used forever.

This guarantees that each membrane present in the system, except the skin membrane, is marked with objects from $N \cup T$ and exactly one t . This object t is used to de-attach the terminal objects from the membranes and to make them migrate toward the skin membrane where they remain attached. This is done by using the rules from group 2 (this process of migration can start at any moment during the computation; it does not interfere with the result of the computation).

Finally, the object E attached to the skin is removed; it can be removed only when all objects from T present in Π have been moved and attached to the skin membrane, otherwise these objects move through the regions of the system forever and the computation will not halt.

In this way for any string w in $L(G)$ one can obtain, at the end of a halting computation, a marking of the skin membrane corresponding to the Parikh vector of w . In fact, this can be done by applying the pino rules and then moving all the objects from T to the skin membrane in the above described way.

On the other hand, each multiset w produced by Π is a marking of the skin membrane in a halting configuration, and it can be only obtained in the way described above – hence, there exists a derivation in G that produces a string with its Parikh vector corresponding to w .

The proof can be adapted for systems using $(ncoo)pino_e$ or $(ncoo)drip$ rules by adapting the protein movement rules. Hence, the theorem holds. ■

If P_{pp} systems are equipped with both protein-membrane and protein movement rules, then they are computationally complete, in the sense that they are able to generate the family of Parikh images of recursively enumerable languages.

So, informally, it seems that the ability to move the proteins (in a controlled way) through the regions of the system is important for reaching computational completeness. On the other hand, it is interesting to notice that the generative power of protein movement rules, when used alone, is very “weak” (Theorem 5).

By comparing the following proof with the proof of Theorem 8 we clearly notice similarities. The main difference is the second group of rules, used to simulate the appearance checking mechanism present in the programmed grammar.

Theorem 10

$PsPP_*(\alpha, prot) = PsRE$, $\alpha \in \{pino_i, pino_e, drip\}$.

Proof

We first prove the theorem for systems from $PP_*(pino_i, prot)$.

The inclusion $PsPP_*(pino_i, prot) \subseteq PsRE$ follows from the Church-Turing thesis. The reverse inclusion, $PsRE \subseteq PsPP_*(pino_i, prot)$ can be proved by sim-

ulating a programmed grammar with appearance checking G by a P_{pp} system Π from $PP_*(pino_i, prot)$.

To this aim, consider $G = (N, T, P, S)$ in the l_h -normal form. Let $Lab'(G) = \{i' \mid i \in Lab(G)\}$.

The P_{pp} system Π is defined as follows.

$$\Pi = (V, \mu, u_1, u_2, R, F), \text{ where}$$

- $V = N \cup T \cup \{E\} \cup Lab(G) \cup Lab'(G) \cup \{\#, d, d', h\} \cup \{h'_i, h''_i \mid i \in Lab(G)\}$;
- $\mu = [[]_{ESl_0h}]_\lambda$;
- $F = T$;
- The pino rules and the protein movement rules in R are given in groups, according to their intended use during the simulation of G by Π .

1. (simulation of the productions of the programmed grammar),

$$[]_{hEAi} \rightarrow [[]_{hExj}]_i, \text{ for } (i : A \rightarrow x, E_i, F_i) \in P, j \in E_i, i \neq l_h,$$

2. (simulation of the skipping of a production – appearance checking),

$$\begin{aligned} []_{Ehi} &\rightarrow [[]_{Eh'_i h''_i}]_i, i \in Lab(G), i \neq l_h, \\ []_{Eh'_i h''_i} &\rightarrow []_{Eh'_i h''_i}, []_{Eh'_i} \rightarrow [[]_{E\#\#}]_{h''_i}, i \in Lab(G), i \neq l_h, \\ []_{Eh'_i A} &\rightarrow [[]_{E\#\#}]_{h'_i A}, \text{ for } (i : A \rightarrow x, E_i, F_i) \in P, \\ [h''_i]_i &\rightarrow []_{h''_i i}, i \in Lab(G), \\ [h''_i i] &\rightarrow [[]_{j'd}]_i, i \in Lab(G), j \in F_i, \\ []_{j'd} &\rightarrow []_{dj'}, j \in Lab(G), \\ []_{Ej'} &\rightarrow []_{Ej'}, j \in Lab(G), j \neq l_0, j \neq l_h, \\ []_{Ej'h'_i} &\rightarrow [[]_{Ejh}]_{j'}, i, j \in Lab(G), \end{aligned}$$

3. (used to produce non-halting),

$$[]_{E\#\#} \rightarrow [[]_{E\#\#}]_\#,$$

4. (used to keep the symbols from a sentential form on the same membrane),

$$[]_{Xi} \rightarrow [[]_{E\#\#}]_i, X \in (N \cup T), i \in Lab(G) \cup Lab'(G),$$

5. (used to halt a computation),

$$\begin{aligned} []_{El_h} &\rightarrow []_{l_h} E, \\ []_{l_h Z} &\rightarrow []_{l_h} Z, \\ []_{l_h h} &\rightarrow []_{l_h} h, \\ []_{l_h} &\rightarrow []_{l_h}, \\ []_{El_h} &\rightarrow []_{El_h}, \\ []_{Zl_h} &\rightarrow []_{Zl_h}, \\ []_{hl_h} &\rightarrow []_{hl_h}, \\ []_{Xl_h} &\rightarrow []_{Xl_h}, X \in N. \end{aligned}$$

The so-constructed system Π works as follows.

Each computation starts from the initial configuration $[[]_{ESl_0h}]_\lambda$. At each step, there is a unique membrane of Π marked by a multiset composed by the object E , the objects corresponding to the current sentential form of G (only the object S at the beginning of a computation), the label of the next production of G to simulate (l_0 at the beginning of a computation), and the “support” object h .

The pino rules of group 1 simulate the application of a production ($i : A \rightarrow x, E_i, F_i$) of G , not used in the appearance checking mode (i.e., the nonterminal A is present in the multiset marking the membrane to which E is attached). In this case the pino rule corresponding to the production $A \rightarrow x$, with label i is applied, and together with the objects x also the label of the next production to simulate ($j \in E_i$) is produced.

If a production cannot be applied (because A is not present in the multiset marking the membrane to which E is attached), then the production has to be used in the appearance checking mode (i.e., has to be skipped) and, for this goal, the rules of group 2 are used. The system “guesses”, by applying a rule $[]_{Ehi} \rightarrow [[]_{Eh'_i h''_i}]_i$ from group 2, that the production of G with label i that should be currently simulated, cannot be executed.

For instance, consider the configuration $[\cdots []_{hExi} \cdots]$, with x a string representing a multiset over $N \cup T$ (it represents the current sentential form of G) and i the label of the next production of G to simulate. By applying $[]_{Ehi} \rightarrow [[]_{Eh'_i h''_i}]_i$ we obtain the configuration $[\cdots [[]_{Eh'_i h''_i}]_i \cdots]$. Then, the rule $[]_{Eh'_i h''_i} \rightarrow []_{Eh'_i h''_i}$ is applied (the other rules that could be applied lead to a non-halting computation). So the configuration $[\cdots [[]_{Eh'_i h''_i}]_i \cdots]$ is obtained. In the next step, in parallel, the rule $[]_{Eh'_i A} \rightarrow [[]_{E\#\#}]_{h'_i A}$ with $(i : A \rightarrow x, E_i, F_i) \in P$ is applied (if possible) and the rule $[h''_i]_i \rightarrow []_{h''_i i}$ is applied (this is certainly possible). If the first rule is applied, then this means that the production with label i could be simulated and the (guess) decision to skip it was wrong. In this case the proteins $E\#\#$ are attached to the created membrane and the computation never halts. If the first rule is not applied, then the next configuration reached is $[\cdots [[]_{Eh'_i x}]_{h''_i i} \cdots]$. Now only the pino rule $[]_{h''_i i} \rightarrow [[]_{j'd}]_i$, $j \in F_i$, can be applied. Therefore, the next configuration obtained is the following one (notice the movement of the contents in the pino operation): $[\cdots [[]_{j'd} []_{Eh'_i x}]_i \cdots]$. Now the protein j' is de-attached using the rule $[]_{j'd} \rightarrow []_{dj'}$. So the next configuration obtained is $[\cdots [[]_{dj'} []_{Eh'_i x}]_i \cdots]$. The protein j' is added to the marking of the membrane where the protein E is already attached, by using $[]_{Ej'} \rightarrow []_{Ej'}$. In this way the configuration $[\cdots [[]_{dj'} []_{j'Eh'_i x}]_i \cdots]$ is obtained. Finally, the pino rule $[]_{Ej'h'_i} \rightarrow [[]_{Ejh}]_{j'}$ can be applied. So the next configuration is $[\cdots [[]_{dj'} [[]_{Ejh}]_{j'}]_i \cdots]$. Therefore, the process can be iterated by applying (or skipping) the production of G with label j , in the way described above. Also, the rules of group 4 ensure that the sentential form is always entirely attached to the membrane where E is attached, and it is never divided randomly between the membranes created by the pino operation.

The correct halting of the computation is assured by applying the rules of

group 5. The computation can be halted when the production with label l_h should be simulated. In this case, it is necessary to remove the nonterminal Z and to check that the objects attached to the membrane containing the current sentential are terminals.

For instance, consider the configuration $[\cdots []_{hExl_h} \cdots]$, where x is a string representing a multiset over V (the current sentential form of G). First, the proteins E , Z (present in x) and h are de-attached, obtaining the configuration $[\cdots []_{x'l_h} hEZ \cdots]$. Then, finally, also l_h is released yielding to the configuration $[\cdots []_{x'l_h} hEZ \cdots]$. The release of l_h cannot be done earlier, since otherwise l_h would be re-attached because of the presence of E, Z or h . Once l_h has been released, it is attached again to the same membrane if and only if a nonterminal object is attached to this membrane (rule $[]_X l_h \rightarrow []_{Xl_h}$); this label is released again by the rule $[]_{l_h} \rightarrow []_{l_h}$ and this attachment/de-attachment runs forever. This guarantees that, when the computation halts, the unique output membrane (the one where is attached at least a flag) contains only objects from T .

From the above explanation it is easily seen that any successful derivation of G producing w can be simulated in Π by a successful computation halting in a configuration with a unique output membrane marked with the multiset $\Psi_T(w)$. The simulation in Π of this type of derivation leads to a non-halting computation.

Therefore it follows that $Ps(\Pi)$ is exactly the Parikh image of $L(G)$.

Moreover the proof given can be easily adapted by using *pino*_e or by using *drip* rules (by adjusting the protein movement rules). Thus the theorem follows.■

6.8 Decision Problems

Since the set of proteins attached to a membrane determines the set of rules that can be applied to this membrane, we will consider now the following decision problem: Is it decidable whether or not an arbitrary multiset w is a reachable marking for an arbitrary P_{pp} system?

We will demonstrate that this problem is decidable for P_{pp} systems using (i) only *pino* and/or *drip* rules, or (ii) only protein movement rules, while it is not decidable for P_{pp} systems using both *pino* (or *drip*) rules and protein movement rules.

Theorem 11

It is undecidable whether or not, for any P_{pp} system Π and any multiset w of proteins over V_Π , w is a reachable marking of Π .

Proof

Sketch. The result follows from the universality results proved in Theorem 10.

For any programmed grammar G with appearance checking it is possible to construct a P_{pp} system Π that can simulate the derivations in G . Consider now the construction given in Theorem 10. If there exists an algorithm to check if an arbitrary multiset w is a reachable marking of Π , then the same algorithm together with the construction from Theorem 10, could be used to decide whether or not

for an arbitrary Parikh vector v a sentential form z with Parikh vector equal to v can be generated in G . This, however, contradicts the universality of programmed grammars with appearance checking (which has been proved in a constructive way, see [11]). ■

If P_{pp} systems use only protein movement rules, only pino rules, or only drip rules, then the above problem becomes decidable.

Theorem 12

It is decidable whether or not, for any P_{pp} system Π from $PP_*(prot)$ and any multiset w of proteins over V_Π , w is a reachable marking of Π .

Proof

Given a P_{pp} system Π from $PP_*(prot)$ the number of possible distinct reachable configurations for Π is finite and therefore the problem is decidable (e.g., by using an exhaustive search). ■

Theorem 13

It is decidable whether or not, for any P_{pp} system Π from $PP_*(\alpha)$, $\alpha \in \{pino_i, pino_e, drip\}$, and any multiset w of proteins over V_Π , w is a reachable marking of Π .

Proof

We first show that the set of strings representing all the reachable markings for $\Pi \in PP_*(pino_i)$ can be generated by a programmed grammar G without appearance checking.

Let $\Pi = (V, \mu, p_1, p_2, \dots, p_m, R, F)$ where $R = \{r_1, \dots, r_k\}$.

In what follows, in order to avoid writing an entire pino rule $r_i : []_{uav} \rightarrow []_{ux}]_v$ we will often refer directly to the strings u , v , and x and to the symbol a . Thus, e.g., we may write “consider a string u of r_i ”.

We also use the morphism $h : V \longrightarrow V'$ defined by $h(x) = x'$, $x \in V$, and $V' = \{a' \mid a \in V\}$.

Let $G = (N, T, P, S)$ be a pure programmed grammar, thus $N = T$, where N is the set $V \cup V'$.

In what follows, in order to simplify the notation, we assume that several productions of G can have the same label (in this case the production to be applied is chosen nondeterministically among the ones with the same label). Note that this assumption is only a notational convenience: it is easy to see that for each such G there is an equivalent pure programmed grammar having an injective labeling of the productions.

For the sake of readability we will use l_{beg} to denote the set of labels of G which correspond to productions that are used to initiate the simulation of pino rules. Thus, l_{beg} (the set of beginning labels) is defined as

$$\begin{aligned} l_{beg} &= \{l'_{r_i,1} \mid r_i \text{ has } u \neq \lambda, 1 \leq i \leq k\} \\ &\cup \{l'_{r_i} \mid r_i \text{ has } u = \lambda, 1 \leq i \leq k\} \cup \{l''_{r_i,1} \mid r_i \text{ has } v \neq \lambda, 1 \leq i \leq k\} \\ &\cup \{l''_{r_i,r+1} \mid r_i \text{ has } v = \lambda, 1 \leq i \leq k\}. \end{aligned}$$

The label used to start the simulation of a pino rule r_i can be different according to the presence of strings u and v in the rule r_i . In fact, if in the chosen rule r_i , u or/and v are missing, then some of the productions of G need to be skipped.

The set of productions P is divided into several groups, according to their intended use during the simulation of Π .

1. (nondeterministic choosing of one membrane and of one pino rule),
 $(l_0 : S \rightarrow p_i, l_{beg}), \text{ for } 1 \leq i \leq n,$

If the pino rule $r_i : [\]_{uav} \rightarrow [[\]_{ux}]_v, 1 \leq i \leq k$, is present in R , with $u = u_1 u_2 \cdots u_j, v = v_1 v_2 \cdots v_r$ and $x = x_1 x_2 \cdots x_p$, then we add to P the following productions.

- 2(a). (prime the symbols of the string u),
 $(l'_{r_i,1} : u_1 \rightarrow h(u_1), \{l'_{r_i,2}\}),$
 $(l'_{r_i,2} : u_2 \rightarrow h(u_2), \{l'_{r_i,3}\}),$
 $\dots$
 $(l'_{r_i,j} : u_j \rightarrow h(u_j), \{l'_{r_i}\}),$

- 3(a). (prime the symbol a),
 $(l'_{r_i} : a \rightarrow h(a), \{l'_{r_i,j+1}\}), \text{ if } r_i \text{ has } v \neq \lambda,$
 $(l'_{r_i} : a \rightarrow h(a), \{l'_{r_i,j+r+1}, l_{1,i}\}), \text{ if } r_i \text{ has } v = \lambda,$

- 4(a). (delete the symbols of v),
 $(l'_{r_i,j+1} : v_1 \rightarrow \lambda, \{l'_{r_i,j+2}\}),$
 $(l'_{r_i,j+2} : v_2 \rightarrow \lambda, \{l'_{r_i,j+3}\}),$
 $\dots$
 $(l'_{r_i,j+r} : v_r \rightarrow \lambda, \{l'_{r_i,j+r+1}, l_{1,i}\}),$

- 5(a). (delete nondeterministically),
 $(l_{1,i} : d \rightarrow d, \{l'_{r_i,j+r+2}\}), d \in N, \text{ if } r_i \text{ has } u \neq \lambda,$
 $(l_{1,i} : d \rightarrow d, \{l'_{r_i,2j+r+2}\}), d \in N, \text{ if } r_i \text{ has } u = \lambda,$
 $(l'_{r_i,j+r+1} : d \rightarrow \lambda, \{l'_{r_i,j+r+1}, l'_{r_i,j+r+2}\}), d \in V, \text{ if } r_i \text{ has } u \neq \lambda,$
 $(l'_{r_i,j+r+1} : d \rightarrow \lambda, \{l'_{r_i,j+r+1}, l'_{r_i,2j+r+2}\}), d \in V, \text{ if } r_i \text{ has } u = \lambda,$

- 6(a). (de-prime the symbols of u),
 $(l'_{r_i,j+r+2} : u'_1 \rightarrow u_1, \{l'_{r_i,j+r+3}\}),$
 $(l'_{r_i,j+r+3} : u'_2 \rightarrow u_2, \{l'_{r_i,j+r+4}\}),$
 $\dots$
 $(l'_{r_i,2j+r+1} : u'_j \rightarrow u_j, \{l'_{r_i,2j+r+2}\}),$

- 7(a). (apply $a \rightarrow x$ and choose the next pino rule),
 $(l'_{r_i,2j+r+2} : a' \rightarrow x, l_{beg}),$

2(b). (prime the symbols of v),

$(l''_{r_i,1} : v_1 \rightarrow h(v_1), \{l''_{r_i,2}\}),$

$(l''_{r_i,2} : v_2 \rightarrow h(v_2), \{l''_{r_i,3}\}),$

...

$(l''_{r_i,r} : v_r \rightarrow h(v_r), \{l''_{r_i,r+1}\}),$

3(b). (prime the symbol a),

$(l''_{r_i,r+1} : a \rightarrow h(a), \{l''_{r_i,r+2}\}),$ if r_i has $u \neq \lambda$,

$(l''_{r_i,r+1} : a \rightarrow h(a), \{l''_{r_i,r+j+2}, l_{2,i}\}),$ if r_i has $u = \lambda$,

4(b). (delete the symbols of u),

$(l''_{r_i,r+2} : u_1 \rightarrow \lambda, \{l''_{r_i,r+3}\}),$

$(l''_{r_i,r+3} : u_2 \rightarrow \lambda, \{l''_{r_i,r+4}\}),$

...

$(l''_{r_i,r+j+1} : u_j \rightarrow \lambda, \{l''_{r_i,r+j+2}, l_{2,i}\}),$

5(b). (delete nondeterministically),

$(l_{2,i} : d \rightarrow d, \{l''_{r_i,r+j+3}\}), d \in N,$

$(l''_{r_i,r+j+2} : d \rightarrow \lambda, \{l''_{r_i,r+j+2}, l''_{r_i,r+j+3}\}), d \in V,$

6(b). (delete the symbol a'),

$(l''_{r_i,r+j+3} : a' \rightarrow \lambda, \{l''_{r_i,r+j+4}\}),$ if r_i has $v \neq \lambda$,

$(l''_{r_i,r+j+3} : a' \rightarrow \lambda, l_{beg}),$ if r_i has $v = \lambda$,

7(b). (de-prime the symbols of v and choose the next pino rule),

$(l''_{r_i,r+j+4} : v'_1 \rightarrow v_1, \{l''_{r_i,r+j+5}\}),$

$(l''_{r_i,r+j+5} : v'_2 \rightarrow v_2, \{l''_{r_i,r+j+6}\}),$

...

$(l''_{r_i,r+j+4+r-1} : v'_r \rightarrow v_r, l_{beg}),$

The so-constructed grammar G simulates Π in the following way.

The underlying idea is that G stores in its sentential forms the strings corresponding to reachable markings of Π (with one reachable marking stored in one sentential form).

The grammar simulates, by using its productions, the evolution of a single membrane from a reachable configuration of Π ; if a membrane has several possible evolutions then G “chooses” only one of them in a nondeterministic fashion. When a pino rule is simulated, then the grammar chooses, nondeterministically, to follow the evolution of either the created internal membrane or the created external membrane.

Initially, the grammar G applies one of the productions present in the group 1, having label l_0 . So the symbol S is rewritten in a nondeterministic way into one of the strings $p_1, p_2, \dots, p_m$ corresponding to the initial markings of Π . The choice can be done in a nondeterministic manner since in the systems we consider here,

the evolution of a membrane present in a certain configuration is independent from the evolution of the other membranes present in the same configuration.

The next production is selected by choosing, in a nondeterministic way, a label in the set l_{beg} associated with the production with label l_0 .

We will discuss the functioning of G when it simulates a pino rule in which both contexts u and v are nonempty (the reader can easily verify the functioning of G in the case when one or both contexts are empty).

Given a pino rule $r_i : []_{uav} \rightarrow [[]_{ux}]_v$, the label $l'_{r_{i,1}}$ is used to start the sequence of productions that simulate the pino rule r_i , in the case G follows the evolution of the created internal membrane; on the other hand $l''_{r_{i,1}}$ is used to start the sequence of productions that simulate the pino rule r_i in the case G follows the evolution of the created external membrane.

The productions in group (a) are used in the former case, while the productions in group (b) are used in the latter case.

(i): We now analyze the former case. Thus we suppose that, after applying the production labeled by l_0 , we have chosen the production with label $l'_{r_{i,1}}$ and r_i is the pino rule $[]_{uav} \rightarrow [[]_{ux}]_v$. Therefore, the grammar simulates the rule r_i and chooses to follow the evolution of the created *internal* membrane.

First, the productions of the group 2(a) are executed in sequence – they are used to mark all the symbols in the sentential form corresponding to the objects of the string u of the pino rule r_i .

Then the production $a \rightarrow h(a)$ is applied, and the object a is primed (group 3(a)).

After that, the productions of group 4(a) are applied in sequence (we suppose $v \neq \lambda$). This corresponds to the deletion from the sentential form of the objects from the string v of the pino rule r_i . These objects are deleted because the grammar has chosen to follow the evolution of the created internal membrane.

When this phase is completed, then some (possibly none) of the symbols in the sentential form are randomly deleted. This is used to simulate the random distribution of the objects between the two newly created membranes, and in particular the deletion simulates the nondeterministic distribution of some of the objects to the created external membrane.

This deletion can be stopped by choosing to execute the production with label $l'_{r_{i,j+r+2}}$ (notice that the deletion can be even totally skipped by choosing the special “dummy” production with label $l_{1,i}$).

When the deletion is stopped, then the introduced symbols of u are de-marked by applying, in sequence, the productions from group 6(a), and finally the symbols of the string x are introduced by using the rules of group 7(a). Moreover, when this last production is applied, a new pino rule is nondeterministically selected by choosing a label in l_{beg} and the above described process is repeated.

(ii): Now we analyze the latter case. Thus we suppose that, after executing the production with label l_0 , we have chosen the production with label $l''_{r_{i,1}}$ where the pino rule r_i is $[]_{uav} \rightarrow [[]_{ux}]_v$. Therefore, the constructed grammar simulates the application of the pino rule r_i with the choice to follow the evolution of the

created *external* membrane.

This is done by applying, in a way analogous to the one described above, the rules of the group (b) , in the order described by groups 1, $2(b)$, $3(b)$, $4(b)$, $5(b)$, $6(b)$, and $7(b)$.

Since G is pure, the language $L(G)$ consists of all reachable sentential forms. Notice that during the intermediate steps of the simulation of a pino rule there are always primed symbols in the sentential forms produced by G .

By Lemma 2, $L(G)$ can also be generated by a programmed grammar without appearance checking.

We are interested in the set of strings corresponding to reachable markings of Π and to obtain this set, we only need to intersect $L(G)$ with the regular set V^* , filtering out in this way the strings from $L(G)$ containing primed symbols (note that these are the only strings in $L(G)$ not corresponding to reachable markings of Π).

The family of languages generated by programmed grammars without appearance checking is closed under intersection with regular sets (see, e.g., [11]); therefore, the language $L_{reach} = L(G) \cap V^*$ can also be generated by such grammars (the proof of this closure property is constructive, i.e., we can construct the grammar generating L_{reach} starting from G and from the automaton for V^*).

Therefore to check if a multiset of proteins w is a reachable marking of Π , we only need to decide if (any permutation of) the string w is in L_{reach} , and this is decidable (see, e.g., [11]).

Since the membrane structure is not really important in the described simulation then it is easy to adapt the given proof for P_{pp} systems using only *pino_e* rules or using only *drip* rules. Therefore, the theorem holds. ■

We conclude this section by investigating two more decision problems. The first problem concerns the reachability of a configuration in P_{pp} systems. The second problem concerns the boundness of P_{pp} systems.

First, we observe that, given an arbitrary P_{pp} system Π and an arbitrary configuration C of Π , one can compute an upper bound $map_{\Pi}(C)$ on the number of applications of pino and drip rules that can be used in deriving C from the initial configuration of Π (in case that C is reachable in Π).

Clearly, one can generate in a systematic fashion all reachable configurations of Π containing no more than r membranes. Since each application of a pino or drip rule increases the number of membranes this generation process takes a bounded number of steps. If C appears among these configurations, then it is reachable, otherwise C is not reachable in Π .

Thus, we have the following result:

Theorem 14

It is decidable whether or not, for any P_{pp} system Π and any configuration C of Π , C is a reachable configuration of Π .

It is perhaps worthwhile to discuss Theorem 14 in the light of the universality result stated in Theorem 10. The reason that Theorem 14 holds is that, for a

given configuration C , one can, a priori, provide an upper bound m_c such that C is reachable in Π if and only if it is reachable by computations that do not exceed m_c steps.

On the other hand, if we want to check whether or not a particular multiset w is in the output of a successful computation of Π , then, in general, there is no upper bound m_w such that: $w \in Ps(\Pi)$ if and only if w is an output of a successful computation which takes no more than m_w steps.

In fact, in general, there is no relationship between the size of w and the maximal size of a halting configuration in which w is marking one of the output membranes.

A P_{pp} system Π is *bounded* if there exists an integer k , such that, any reachable configuration of Π has less than k membranes.

Theorem 15

It is decidable whether or not an arbitrary P_{pp} system Π from $PP_*(\alpha)$, $\alpha \in \{pino_i, pino_e, drip\}$, is bounded.

Proof

Given a P_{pp} system Π from $PP_*(\alpha)$ $\alpha \in \{pino_i, pino_e, drip\}$, one can construct a programmed grammar G without appearance checking such that $L(G)$ consists of strings corresponding to all the reachable markings of Π . Such a grammar G can be constructed in the way described in the proof of Theorem 13.

Since it is decidable whether or not the language of an arbitrary programmed grammar without appearance checking is finite (see, e.g., [11]), we can decide whether or not $L(G)$ is finite. If $L(G)$ is infinite, then, obviously, Π is not bounded. Assume now that $L(G)$ is finite. Note that still Π can be unbounded because, e.g., many membranes can have the same marking in a certain configuration. It is easy to see that $L(G)$ can be effectively constructed from G : by iteration we can find a k such that $L(G) \cap V^k V^* = \emptyset$, where V is the alphabet of G , and we now need to check the membership of w in $L(G)$ (which is decidable) for only a finite number of $w \in V^{k-1}$.

By analyzing $L(G)$ it is possible to decide if a pino (or a drip) rule can be applied an unbounded number of times. This is done by constructing a graph having nodes labeled by the strings of $L(G)$. We add directed edges between the nodes in the following way. If a node x is labeled by w_1 and a node y is labeled by w_2 , then there is an edge between the two nodes, directed from w_1 to w_2 , if and only if there is a pino (or a drip) rule in Π that applied to the membrane $[]_{w_1}$ can produce two membranes where at least one of the them is marked by w_2 .

Clearly, if the so-constructed graph has a loop, then Π is unbounded; otherwise Π is bounded. ■

6.9 Concluding Remarks

We have investigated membrane systems using operations involving membranes marked with multisets of proteins. These systems use two different kinds of oper-

	w/o <i>prot</i>	<i>prot</i>
w/o <i>pino_i</i>		PsFIN
(<i>ncoo</i>) <i>pino_i</i>	$\subset$ PsCF	$\supseteq$ PsCF
<i>pino_i</i>	$\supseteq$ PsPR	PsRE

Table 6.1: Computational power for P_{pp} systems using *pino_i* and protein movement rules (*prot*). The same table holds also for *pino_e* and *drip* operations.

ations: the ones that involve membranes and proteins (*pino* and *drip* operations) and the ones that attach, de-attach, and move the proteins across the regions of the system (protein movement operations).

Membrane systems using both types of operations are shown to be computationally complete. When the protein-membrane rules are restricted to be non-cooperative, then one generates at least the family of Parikh images of context-free languages.

We have also analyzed membrane systems whose evolution is based on only one of the two types of operations.

In particular we have shown that (in terms of Parikh sets) membrane systems using only *pino* (or only *drip*) rules are at least as powerful as programmed grammars without appearance checking.

Our current knowledge about the computational power of membrane systems considered in this chapter is summarized in Table 6.1.

A number of problems have to be settled in order to get a more complete understanding of membrane systems with marked membranes. Some of them are suggested by the results obtained in this chapter.

1. Is the inclusion of $PsCF \subseteq PsPP_*((ncoo)\alpha, prot)$ $\alpha \in \{pino_i, pino_e, drip\}$, strict?
2. Is the inclusion $PsPP_*((ncoo)\alpha, prot) \subseteq PsRE$, $\alpha \in \{pino_i, pino_e, drip\}$, strict?
3. Is the inclusion $PsPR \subseteq PsPP_*(\alpha)$, $\alpha \in \{pino_i, pino_e, drip\}$, strict?
4. Is the inclusion $PsPP_*(\alpha) \subseteq PsRE$, $\alpha \in \{pino_i, pino_e, drip\}$, strict?

Also the following “natural” decision problem should be settled for membrane systems with marked membranes: is it possible to decide whether or not an arbitrary multiset of proteins is a reachable marking for an arbitrary P_{pp} system from $PP_*((ncoo)\alpha, prot)$, with $\alpha \in \{pino_i, pino_e, drip\}$?

The problem is challenging since it is proved to be decidable for P_{pp} systems from $PP_*(prot)$, i.e., using only protein movement rules (see Theorem 12), and for P_{pp} systems from $PP_*(\alpha)$, $\alpha \in \{pino_i, pino_e, drip\}$, i.e., using only protein-membrane rules (see Theorem 13), while it is undecidable for arbitrary P_{pp} systems (Theorem 11).

A more general research line is to consider a more realistic model of the way that proteins are embedded in membranes. A possible starting point is to accommodate within our model the concept of the parametric regular spherical membrane presented in [2].

An interesting research topic is to consider protein rules with different execution times (following, for instance, the idea of timed P systems introduced in [7]). Also, it would be interesting to consider proteins marking only one of two sides of a membrane, similar as defined in [10]. In this way, these proteins would only have an effect on one side of the membrane.

Chapter 7

Membrane Systems with External Control

Abstract

We consider the idea of controlling the evolution of a membrane system. In particular, we investigate a model of membrane systems using promoted rules, where a string of promoters (called the control string) “travels” through the regions, activating the rules of the system. This control string is present in the skin region at the beginning of the computation – one can interpret that it has been inserted in the system before starting the computation – and it is “consumed”, symbol by symbol, while traveling through the system. In this way, the inserted string drives the computation of the membrane system by controlling the activation of evolution rules. When the control string is entirely consumed and no rule can be applied anymore, then the system halts – this corresponds to a successful computation. The number of objects present in the output region is the result of such a computation. In this way, using a set of control strings (a control program), one generates a set of numbers. We also consider a more restrictive definition of a successful computation, and then study the corresponding model.

In this chapter we investigate the influence of the structure of control programs on the generative power. We demonstrate that different structures yield generative powers ranging from finite to recursively enumerable number sets.

In determining the way that the control string moves through the regions, we consider two possible “strategies of traveling”, and prove that they are similar as far as the generative power is concerned.

7.1 Introduction

Membrane systems (also referred to as P systems) were introduced in 1998 by Gh. Păun as computing devices inspired by the structure and functioning of living cells. Since their introduction, several models of P systems have been investigated,

many of them being proved to be computationally complete. The reader is referred to the monograph [21], and to an up-to-date bibliography of this research area available at the P systems web-page, [1].

In nature, the behavior of cells can be influenced by the signals (controls) that they receive from the “outside”. Thus, it may be possible to drive the evolution of a living cell by providing the cell with a specific control.

With this motivation in mind, we introduce and investigate a model of P systems, called *string-controlled P systems* (in short, SC P systems). This model is based (with some modifications) on membrane systems with promoters, introduced in [3]. There, the presence of promoters is used to activate, during the computation, certain rules of the system. The biological motivation is the fact that chemical reactions in living cells can be promoted (or inhibited) by the presence of various enzymes.

A string of promoters (called the *control string*), “produced” by the environment, is present in the skin region of the system at the beginning of a computation. This string (that acts like an external control) travels through the regions of the system, possibly promoting (with its leftmost symbol) the rules of the region where it currently resides. Each time the string moves from one region to another, its leftmost symbol (used as a promoter) gets consumed. When the whole string is consumed, and no rule can be applied in any region, then the system halts, completing a successful computation. The output of such computation is the number of objects present in the output region when the system halts.

We shall also consider another sort of successful computation, which additionally has to satisfy a “clean ending condition” (which requires that an a priori specified “undesirable” object is not present in any region upon the completion of the computation).

In this way, an SC P system generates the set of numbers composed by the outputs of all its computations. Also, a membrane system with a collection of control strings (called the *control program*) generates a set of numbers, which is defined as the union of the sets generated for each single string.

In this chapter we pay special attention to SC P systems where all evolution rules of the system are promoted – hence, only the rules defined in the region where the control string currently resides, and whose promoter matches the leftmost symbol of the control string, may be active. In particular, we investigate how the structure of the control program influences the generative power of such systems, which are called *fully-promoted SC P systems*.

We show that if the control program is finite, then the generative power corresponds exactly to the family of finite sets of numbers. On the other hand, if the family of recursively enumerable languages is used as the control program, then, not surprisingly, the resulting generative power corresponds to the family of Turing computable sets of numbers. Several intermediate results are obtained by balancing the structure of the control program and the power of the evolution rules used by the system.

We consider two different ways (operating modes) for a control string to travel

through the regions of the system: either the string must move at each step (mode (1)), or it is allowed to remain in the same region for several consecutive steps until it decides (nondeterministically) to move again (mode (2)). We prove that, under some natural conditions on the control program, these two modes are similar as far as the generative power is concerned.

The chapter is organized as follows. Section 7.2 recalls some basic notions of formal languages theory used throughout the chapter. A formal definition of SC P systems is presented in Section 7.3. In Section 7.4 we show that the generative power of classes of fully-promoted SC P systems with a natural condition on the control program family are “almost” independent on the chosen operating mode of the movement of the control string. In Section 7.5 we consider structures of control that yield a generative power strictly weaker than *RE*, and in Section 7.6 structures that yield the computational completeness.

We conclude the chapter by suggesting a number of open problems and research directions.

7.2 Preliminaries

Let us briefly recall some notions and results of formal languages to the extent needed in this chapter—in this way we establish the basic notation and terminology needed later on. For more details the reader can consult standard books, such as [25], [11], and the handbook [24].

An *alphabet* V is a finite set of symbols. By V^* we denote the set of all strings over V , the empty string is denoted by λ , and $V^+ = V^* - \{\lambda\}$.

The *length* of a string $w \in V^*$ is denoted by $|w|$, while the number of occurrences of $a \in V$ in w is denoted by $|w|_a$. For a language $L \subseteq V^*$, the set $\text{length}(L) = \{|w| \mid w \in L\}$ is called the *length set* of L .

If FL is a family of languages then NFL is the family of length sets of languages in FL .

We denote by *FIN*, *REG*, *CF*, *CS* and *RE* the families of finite, regular, context-free, context-sensitive and recursively enumerable languages, respectively. Accordingly, for instance, the family of length sets of languages in *RE* is denoted by *NRE* (this is the family of all recursively enumerable sets of natural numbers).

A multiset over V is a mapping $M : V \rightarrow \mathbb{N}_0$; assigning to each $a \in V$ a multiplicity $M(a)$. Commonly, multisets are represented by strings of symbols. In this representation the order of symbols does not matter, because the number of copies of an object in a multiset is given by the number of occurrences of the corresponding symbol in the string. Hence, e.g., a^4b^3d denotes the multiset consisting of 4 occurrences of a , 3 occurrences of b , and one occurrence of d ; the same multiset is also represented by, e.g., $da^2ba^2b^2$.

An ETOL system is a construct $G = (\Sigma, T, H, w)$, where Σ is the (total) alphabet, $T \subseteq \Sigma$ is the terminal alphabet, $H = \{h_1, h_2, \dots, h_k\}$ is a finite set of finite substitutions (tables) over Σ , and $w \in \Sigma^*$ is the axiom; each $h_i \in H$,

$1 \leq i \leq k$, can be represented by a list of context-free productions $A \rightarrow x$, such that $A \in \Sigma$ and $x \in \Sigma^*$ (moreover, for each symbol A of Σ and each table h_i , $1 \leq i \leq k$, there is a production in h_i with A as the left hand side). Then G defines, for each $1 \leq i \leq k$, a derivation relation $\Rightarrow_{h_i}$ by $x \Rightarrow_{h_i} y$ iff $y \in h_i(x)$. We write $x \Rightarrow y$ if $x \Rightarrow_{h_i} y$ for some $1 \leq i \leq k$. As usual, $x \Rightarrow^* y$ denotes the reflexive and transitive closure.

The language generated by G is $L(G) = \{z \in T^* \mid w \Rightarrow^* z\}$. We denote by *ET0L* the family of languages generated by ET0L systems, and by *T0L* the family of languages generated by ET0L systems such that $\Sigma = T$.

A regularly (context-free, respectively) controlled ET0L system, E(rc)T0L system (E(cfc)T0L system, respectively) in short, is a pair $\Omega = (G, L)$ where $G = (\Sigma, T, H, w)$ is an ET0L system and L is a regular (context-free, respectively) language over H .

The language generated by Ω is

$$L(\Omega) = \{z \in T^* \mid w = w_0 \Rightarrow_{h_{i_1}} w_1 \Rightarrow_{h_{i_2}} \cdots \Rightarrow_{h_{i_m}} w_m = z, h_{i_1} \cdots h_{i_m} \in L\}.$$

We denote by *E(rc)T0L* the family of languages generated by E(rc)T0L systems, and by *E(cfc)T0L* the family of languages generated by E(cfc)T0L systems.

The following known inclusions between families of languages will be used in this chapter (see, e.g., [25]):

$$FIN \subset CF \subset ET0L \subset CS \subset RE.$$

From [14] we recall the following result.

$$ET0L = E(rc)T0L.$$

Moreover, it is known that for each $L \in ET0L$ there exists an ET0L system G , with only 2 tables, such that $L = L(G)$ (see, e.g., [23]).

A *regularly controlled grammar with appearance checking* is a tuple

$$G = (N, T, S, P, K, F)$$

where N, T, S , and P are the set of nonterminals, the set of terminals, the starting symbol and a finite set of context-free productions, respectively. Each production in P has a uniquely associated label, and the set of all these labels is denoted by $lab(P)$. K is a regular language over $lab(P)$ and $F \subseteq lab(P)$. Let $V = N \cup T$. We say that $x \in V^+$ *derives* $y \in V^*$ in the appearance checking mode by application of $A \rightarrow w$ with label p (written as $x \Rightarrow_p^{ac} y$) if either $x = x_1 A x_2$ and $y = x_1 w x_2$, or A does not appear in x , $p \in F$, and $x = y$.

The language $L(G)$, generated by G , consists of all strings $w \in T^*$ such that there is a derivation $S \Rightarrow_{p_{i_1}}^{ac} w_1 \Rightarrow_{p_{i_2}}^{ac} w_2 \Rightarrow_{p_{i_3}}^{ac} \cdots \Rightarrow_{p_{i_n}}^{ac} w_n = w$, for some $n \geq 1$ and $p_{i_1} p_{i_2} \cdots p_{i_n} \in K$.

By rC_{ac} we denote the family of languages generated by regularly controlled grammars with appearance checking and erasing productions, and by rC we denote the family of languages generated by regularly controlled grammars with erasing productions and without appearance checking (the set F is empty).

The following lemma holds (see, [11]):

Lemma 1

$$rC_{ac} = RE.$$

In what follows we assume that the reader is familiar with the membrane computing area, in particular with the class of P systems with rewriting rules and symbol-objects, and with the notions of P systems using promoters/inhibitors; for instance as presented in [3, 17, 22] or in Chapter 3 of [21].

7.3 String-Controlled P Systems

A string-controlled P system, as informally described in Introduction, is defined as follows.

Definition 2

A string-controlled P system (in short, *SC P system*) is a construct

$$\Pi = (V, C, P, L, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0),$$

where:

- V is the alphabet of Π ; its elements are called *objects*;
- $C \subseteq V$ is the set of *catalysts*;
- P is the set of *promoters*; $P \cap V = \emptyset$;
- $L \subseteq P^*$ is the *control program* (each string in L is a *control string*);
- μ is a *membrane structure* consisting of m membranes labeled $1, \dots, m$;
- w_i , $1 \leq i \leq m$, are strings that represent the multisets over V initially associated with the regions $1, 2, \dots, m$ of μ ;
- R_i , $1 \leq i \leq m$, are finite sets of *evolution rules* associated with the regions $1, 2, \dots, m$ of μ . Each evolution rule is either of the form $u \rightarrow v$ or of the form $u \rightarrow v|_p$, where $u \in V^+$, $p \in P$, and $v \in V_{tar}^*$ with $V_{tar} = V \times TAR$, for $TAR = \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$;
- $i_0 \in \{1, \dots, m\}$ specifies the output region of Π . ■

As usual, the *membrane structure* is a hierarchical arrangement of membranes, embedded in a *skin membrane*, which separates the system from the environment. A membrane without any membrane inside is called *elementary*. Each membrane defines a *region*. For an elementary membrane this is the space enclosed by it, while for a non-elementary membrane, is the space in-between the membrane and the membranes directly included in it. As usual, labels $1, \dots, m$ identify both membranes and their corresponding regions.

Evolution rules of the form $u \rightarrow v|_p$ are called *promoted*, and evolution rules of the form $u \rightarrow v$ are called *non-promoted*. An evolution rule is called *non-cooperative* if $u \in V$. Also, an evolution rule is called *catalytic* if it is either of the form $ca \rightarrow cv$ or of the form $ca \rightarrow cv|_p$, where $a \in (V - C)$, $c \in C$, $p \in P$, and $v \in ((V - C) \times TAR)^*$. The elements of TAR are called *targets*. It is convenient to denote $(a, t) \in V_{tar}$ by a if $t = here$, and by a_t otherwise.

A *configuration* of Π is a description of the membrane structure and of the contents of all the regions. An *initial configuration* of Π consists of the membrane structure μ , the objects initially present in the regions of the system, as described by $w_1, \dots, w_m$, and by one string from L , present in the skin region (this string is called *control string*). Notice that Π has a set of initial configurations, one for each element of L .

As standard, we suppose the existence of a global clock that marks the steps of the system.

At each step, the control string moves, in a nondeterministic way, across the regions of Π . We distinguish *two possible modes* of operation for Π : (1) at each step the string moves passing from one region to an adjacent one; (2) at each step the string may move to an adjacent region or remain in the same region. In both cases the control string cannot move to the environment, and when it moves from a region to another one, it loses its leftmost symbol. The leftmost symbol of the control string is called the *head*.

At each step the *head* of the current control string is used as a *promoter* for the rules present in the region where the string resides. A promoted rule is *active* if its promoter is present. The rules that are not promoted are always active.

A *transition* between two configurations of Π is obtained by applying in one step the active rules in each region of Π in a maximally parallel nondeterministic manner. More precisely, if a rule $u \rightarrow v \in R_i$ or $u \rightarrow v|_p \in R_i$ is active and the multiset u is present in region i , then the *application* of this rule means removing u from region i and adding the objects specified by v in the regions indicated by the corresponding target commands.

A sequence of transitions, starting from an initial configuration of Π , is called *computation*. A computation *halts* when there is no applicable rule in any region of Π and the control string is entirely consumed (Π has reached a *halting configuration*).

We shall consider two definitions of *successful* computation for Π :

- in the standard case, we say that all halting computations of Π are successful,
- in the $\#$ case, we consider that a halting computation of Π is successful if and only if a special *a priori* designated symbol $\# \in V$ is not present in the halting configuration in any region of Π .

The *result* of a successful computation ω is the number of objects present in the output region i_0 in the halting configuration of ω . Depending on the definition of *successful* computation that is considered, we shall say that the system collects the result in the standard way, or in the $\#$ way.

We use the notation $P_m(\alpha, FL)$, where $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}$ and FL is a family of languages, to denote the class of SC P systems which use at most m membranes, use only non-cooperative ($ncoo$), cooperative (coo), or catalytic with at most k catalysts (cat_k) evolution rules (promoted or not), and use a control program in FL . We call FL the *control program family* of the class. In the coo case, there is no restriction on the form of the evolution rules. The prefix (*pro*) is added if *only* promoted rules are used (such systems are called *fully-promoted* SC P systems).

We denote by $N^{(i)}(\Pi)$, $i \in \{1, 2\}$, the set of results of all successful computations of Π starting from any possible initial configuration, operating in mode (i), and collecting the result in the standard way. Similarly, we denote by $N_{\#}^{(i)}(\Pi)$, $i \in \{1, 2\}$, the set of results of all successful computations of Π operating in mode (i) and collecting the result in the $\#$ way. Moreover, $N^{(i)}P_m(\alpha, FL) = \{N^{(i)}(\Pi) \mid \Pi \in P_m(\alpha, FL), i \in \{1, 2\}\}$ denotes the family of sets of natural numbers generated by SC P systems from $P_m(\alpha, FL)$ operating in mode (i), $i = 1, 2$, and collecting the result in the standard way. The family $N_{\#}^{(i)}P_m(\alpha, FL)$ is similarly defined.

The following inclusions follow directly from the definitions.

Lemma 3

$$\begin{aligned}
 (pro)N^{(i)}P_m(\alpha, FL) &\subseteq (pro)N_{\#}^{(i)}P_m(\alpha, FL), \\
 (pro)N_{\#}^{(i)}P_m(\alpha, FL_1) &\subseteq (pro)N_{\#}^{(i)}P_m(\alpha, FL_2), \text{ if } FL_1 \subseteq FL_2, \\
 (pro)N_{\#}^{(i)}P_m(ncoo, FL) &\subseteq (pro)N_{\#}^{(i)}P_m(cat_j, FL) \\
 &\subseteq (pro)N_{\#}^{(i)}P_m(cat_{j+1}, FL) \subseteq (pro)N_{\#}^{(i)}P_m(coo, FL), \\
 \text{for } j \geq 1, i \in \{1, 2\}, \alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}, \text{ and } FL, FL_1, FL_2 \text{ families} \\
 &\text{of languages.}
 \end{aligned}$$

7.4 Fully-Promoted SC P Systems

In this section we start the investigation of fully-promoted SC P systems. Notice that for such systems in each time step there is activity in at most one region (the region where the control string currently resides). First we give an example that illustrates the functioning of an SC P system. Then we prove the equivalence (as far as the generative power is concerned) between modes (1) and (2).

The following example shows that a given SC P system Π can produce different results according to its functioning mode.

Example 1

Let Π be the SC P system:

$$\Pi = (V, C, P, L, \mu, w_1, w_2, R_1, R_2, i_0),$$

where:

- $V = \{A\}$,
- $C = \emptyset$,
- $P = \{a, b\}$,
- $L = \{ab\}$,
- $\mu = [{}_1 [{}_2 \]_2]_1$,
- $w_1 = \lambda; w_2 = A$,
- $R_1 = \emptyset$,
- $R_2 = \{A \rightarrow AA|_b\}$,
- $i_0 = 2$.

The system collects the result in the standard way.

When Π operates in mode (1), the unique control string of L is initially present in the skin region and moves, in the next step, to region 2, losing its head a . Therefore, now the rule $A \rightarrow AA|_b$ is activated. In the following step the control string exits region 2, entering region 1, and then its last symbol, b , is consumed. Therefore, there is only one successful computation and we have $N^{(1)}(\Pi) = \{2\}$.

If Π operates in mode (2), then the unique control string of L is initially present in the skin region and it may remain there for a certain number of steps; meanwhile nothing is produced in region 2. At a certain step the string moves into region 2, losing its head a . Then, the rule $A \rightarrow AA|_b$ is activated in region 2 and it will double the number of objects A at each step, until the string b moves back to region 1. When this happens the computation halts and the number of objects produced in region 2 is a power of two, that is, $N^{(2)}(\Pi) = \{2^n \mid n \geq 1\}$.

Example 1 illustrates that for a given fully-promoted SC P system the generated sets under operating modes (1) and (2) may differ (even drastically). However, the family of sets of numbers generated by a *class* of fully-promoted SC P systems with a control program family that is closed under non-erasing regular substitution is “almost” independent on the chosen operating mode. In fact, we show that any fully-promoted SC P system operating in mode (2) [(1), respectively] can be simulated (in a weak sense) by a fully-promoted SC P system operating in mode (1) [(2), respectively] using the same type of rules, the same type of control program, and using a double number of membranes.

Theorem 4

Let $\Pi \in (pro)P_m(\alpha, FL)$, where $m \geq 1$, $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}$, and FL is closed under non-erasing regular substitution. There exists $\Pi' \in (pro)P_{2m}(\alpha, FL)$, such that

$$N_{\#}^{(1)}(\Pi') = \{x + 1 \mid x \in N_{\#}^{(2)}(\Pi)\}.$$

Proof

Let $\Pi = (V, C, P, L, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0) \in (pro)P_m(\alpha, FL)$, and let us construct $\Pi' = (V', C, P', L', \mu', w'_1, \dots, w'_{2m}, R'_1, \dots, R'_{2m}, i_0) \in (pro)P_{2m}(\alpha, FL)$ as follows.

Let $V' = V \cup \{Z\}$, with $Z \notin V$ and $P' = P \cup \{d\}$, with $d \notin P$. We consider the regular substitution ϕ defined by $\phi(p) = p(dp)^*$ for each $p \in P$; we define $L' = \phi(L)$ (notice that the substitution is non-erasing and so every family of languages in $\{REG, CF, CS, RE\}$ is closed under this operation). The structure μ' has $2m$ membranes and is obtained from μ by adding, in each region i , $1 \leq i \leq m$, of μ an (elementary) membrane with label $m+i$. Furthermore we define $w'_i = w_i Z$, for $1 \leq i \leq m$, and $w'_i = Z$, for $m+1 \leq i \leq 2m$.

We define $R'_i = R_i \cup \{Z \rightarrow \#|_d\}$, for $1 \leq i \leq m$, and $R'_i = \{Z \rightarrow \#|_p \mid p \in P\}$, for $m+1 \leq i \leq 2m$.

We shall now show that for every successful computation $\mathcal{C}$ of Π with result x operating in mode (2) there exists a successful computation $\mathcal{C}'$ of Π' with result $x+1$ operating in mode (1).

Consider an arbitrary computation of Π and consider one of its configurations. Now, suppose that in such configuration the current control string w is in region i of Π and has p as its head. Then, there exists a computation in Π' , starting with an “appropriate” control string from L' in the skin, that reaches a configuration having the control string $w' = p(dp)^n x$ present in region i of Π' .

Suppose now that w does not move in Π (Π operates in mode (2)) but remains in the same region for several consecutive steps. This is simulated in Π' by moving w' back and forth between region i and the adjacent dummy region $m+i$, consuming for each movement a symbol p and a dummy promoter d . In this way, an arbitrary computation in Π can be simulated in Π' by a computation starting with an appropriate control string from L' .

On the other hand, Π' does not have other successful computations except those simulating successful computations of Π as described above. In fact, since there is a rule $Z \rightarrow \#|_d$ in every set R'_i , for $1 \leq i \leq m$, which guarantees that the dummy symbol d cannot be used to move the control string into non-dummy regions, otherwise the computation would not be successful. Moreover, if the promoter present immediately to the right of the head of the current control string is non-dummy (i.e., the string is of type pqx , with $p, q \in P$, and $x \in P^*$), then the string must move in a non-dummy region, because otherwise the rules $R'_i = \{Z \rightarrow \#|_p \mid p \in P\}$, for $m+1 \leq i \leq 2m$, would make the computation unsuccessful, if applied. From the above discussion it should be clear that the theorem holds. ■

Conversely, a fully-promoted SC P system operating in mode (1) can be simulated (in a weak sense) by a fully-promoted SC P system operating in mode (2), using a structure having a double number of membranes.

Theorem 5

Let $\Pi \in (pro)P_m(\alpha, FL)$, where $m \geq 1$, $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}$, and FL is closed under non-erasing morphism. There exists $\Pi' \in (pro)P_{2m}(\alpha, FL)$, such

that

$$N_{\#}^{(2)}(\Pi') = \{x + 2 \mid x \in N_{\#}^{(1)}(\Pi)\}.$$

Proof

Given $\Pi = (V, C, P, L, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0)$ we construct $\Pi' = (V', C, P', L', \mu', w'_1, \dots, w'_{2m}, R'_1, \dots, R'_{2m}, i_0)$ as follows.

Let $V' = V \cup \{c, c', Z\}$ and $P' = P \cup \{d, d'\}$, with $c, c', Z \notin V$, and $d, d' \notin P$. We consider the non-erasing morphism ϕ defined by $\phi(p) = pdd'$, for each $p \in P$ – then we set $L' = \phi(L)$ (notice that every family of languages in $\{FIN, REG, CF, CS, RE\}$ is closed under non-erasing morphisms). The membrane structure μ' has $2m$ membranes and is obtained from μ in the following way. In each region $i, 1 \leq i \leq m$, of μ an (elementary) membrane with label $m + i$ is added.

The initial multisets of Π' are $w'_i = cZw_i$, for $1 \leq i \leq m$, and $w'_i = Z$, for $m + 1 \leq i \leq 2m$.

Finally, the evolution rules of Π' are defined in the following way: $R'_i = R_i \cup \{c' \rightarrow c|_{d'}, Z \rightarrow \#|_d\} \cup \{c' \rightarrow \#|_p, c \rightarrow c'|_p \mid p \in P\}$, for $1 \leq i \leq m$. $R'_i = \{Z \rightarrow \#|_p \mid p \in P\}$, for $m + 1 \leq i \leq 2m$.

We will prove now that for every computation of Π operating in mode (1) and producing x , there exists a computation of Π' operating in mode (2) producing $x + 2$.

Consider an arbitrary computation of Π and suppose that, after a certain step k during that computation, the control string $p_{i_1}p_{i_2} \cdots p_{i_j}$, with $p_{i_1}, p_{i_2}, \dots, p_{i_j} \in P$, is present in region i of Π .

Then, there is a computation of Π' (starting with an “appropriate” control string from L') such that the control string $p_{i_1}dd'p_{i_2}dd' \cdots p_{i_j}dd'$ is present in region i of Π' after a given step k' .

In Π , at step $k + 1$, the string must exit region i (Π operates in mode (1)), entering one of the adjacent regions, chosen nondeterministically, losing the promoter p_{i_1} and getting the promoter p_{i_2} as its new head.

This single step of Π is simulated by Π' in the following consecutive steps. The rules activated by promoter p_{i_1} present in region i of Π' are executed at step k' , together with the rule $c \rightarrow c'$ present in every region of Π' and activated by any promoter of P . Therefore, at step $k' + 1$ the control string must exit region i , as otherwise in the next step the rule $c' \rightarrow \#|_{p_{i_1}}$ would be applied and the entire computation would not be successful.

The only region of Π' where the control string can go to is the dummy region $m + i$ present inside region i (otherwise the promoter d that follows p_{i_1} would activate the rule $Z \rightarrow \#|_d$ present in any of the non-dummy adjacent regions of region i and the computation would not be successful). Therefore, suppose the control string goes to region $m + i$, losing in this way the promoter p_{i_1} ; the control string may remain in region $m + i$ for an unbounded number of steps (no rule can be applied there). At a certain step k'' the control string comes back to region i , losing the promoter d and having now the promoter d' as its head; therefore, in the step $k'' + 1$ the rule $c' \rightarrow c|_{d'}$ is applied. The control string having now d'

as head may remain in region i for an unbounded number of steps (no rule can be applied). Eventually, the control string exits region i moving to an adjacent region, losing the promoter d' , and having the promoter p_{i_2} (the next non-dummy promoter) as its new head.

Thus, all possible movements of the control string in Π (i.e., all possible computations) are correctly captured by the functioning of Π' ; consequently, every successful computation of Π can be simulated by Π' .

Notice that, in Π' , if the promoter adjacent to the head of the control string is non-dummy (i.e., it belongs to the set P), then the control string must move in a non-dummy region; otherwise a rule from $R'_i = \{Z \rightarrow \#|_p \mid p \in P\}$, $m+1 \leq i \leq 2m$, is applied and that would make the computation unsuccessful.

Therefore there are no other successful computations of Π' except those that simulate, in the above described way, successful computations of Π . Thus, the theorem holds. ■

7.5 The Influence of the Control Program

Now we analyze in more detail the class of fully-promoted SC P systems operating in mode (1). We show how the structure of the control program and the type of evolution rules influence the generative power of the constructed membrane system. A series of results, ranging from finite power to computational universality, is obtained.

It is worth to remark that one can easily obtain the length set of any language L as output of an SC P system using non-cooperative rules and having L as the control program. Hence, the structure of the control program influences the generative power of SC P systems as the following theorem states.

Theorem 6

$NFL \subseteq (pro)N^{(1)}P_2(ncoo, FL)$.

Proof

Given an arbitrary language L over the alphabet $\Sigma = \{a_1, \dots, a_n\}$, let us consider a symbol $*$ $\notin \Sigma$, and let $L' = h(L)$ where h is the morphism defined by $h(a) = *a$, for every $a \in \Sigma$.

Now let us construct an SC P system that generates $length(L)$ as follows:

$$\Pi = (V, C, P, L', \mu, w_1, w_2, R_1, R_2, i_0),$$

where:

- $V = \{a'_1, \dots, a'_n\}$,
- $C = \emptyset$,
- $P = \Sigma$,
- $\mu = [{}_1 [{}_2 [{}_2]_2]_1]$,

- $w_1 = \lambda; w_2 = a'$,
- $R_1 = \emptyset$,
- $R_2 = \{a' \rightarrow a'_{out}a' | a \in \Sigma\}$,
- $i_0 = 1$.

At the beginning of the computation one of the strings from L' , nondeterministically chosen, is present in the skin region of Π (i.e., region 1). The string moves back and forth between region 1 and region 2 of the system, losing alternatively the symbol $*$ (when passing from region 1 to region 2) and a symbol $a \in \Sigma$ (when moving in the opposite direction). When the string is in region 2, its head $a \in \Sigma$ activates exactly the rule that produces and sends out the symbol a' . Therefore, the number of symbols contained in the output region when the computation halts (the string is entirely consumed) is equal to the number of symbols from Σ that occurred in the inserted control string. Thus Π generates exactly the $length(L)$.■

Now, from Corollary 3, Theorem 6 and the Turing-Church thesis, we have that the class of fully-promoted SC P systems using arbitrary RE languages as control program is universal, even when only non-cooperative rules are used. Hence, the following theorem holds.

Theorem 7

$$(pro)N^{(1)}P_2(ncoo, RE) = (pro)N_{\#}^{(1)}P_2(ncoo, RE) = NRE.$$

It is now natural to ask what happens if we increase the “power” of the evolution rules used by the P system and we decrease the “power” of the control program.

First we consider SC P systems that use cooperative evolution rules and finite control programs.

Theorem 8

$$(pro)N_{\#}^{(1)}P_*(coo, FIN) = (pro)N^{(1)}P_*(coo, FIN) = NFIN.$$

Proof

Given an SC P system Π , it is sufficient to notice that the number of distinct nondeterministic computations using only a finite number of steps is bounded by a constant that only depends on Π . Therefore, if Π has a finite control program, then the set of numbers produced is finite. The other inclusion follows from Theorem 6.■

Let us prove next that the class of fully-promoted SC P systems using arbitrary context-free (regular, respectively) languages as control program generates exactly the family $NE(cfc)T0L$ (or the family $NE(rc)T0L$, respectively), even with non-cooperative rules.

Theorem 9

$$(pro)N_{\#}^{(1)}P_2(ncoo, REG) \supseteq NE(rc)T0L = NET0L.$$

$$(pro)N_{\#}^{(1)}P_2(ncoo, CF) \supseteq NE(cfc)T0L.$$

Proof

Given $\Omega = (G, L)$ an arbitrary E(rc)T0L system (or E(cfc)T0L system, respectively) we construct a SC P system Π in $(pro)P_2(ncoo, REG)$ (in $(pro)P_2(ncoo, CF)$, respectively) such that $N_{\#}^{(1)}(\Pi) = length(L(\Omega))$ as follows.

Let $G = (\Sigma, T, H, w)$ with $H = \{h_1, \dots, h_k\}$. Let

$$\Pi = (V, C, P, L, \mu, w_1, w_2, R_1, R_2, i_0),$$

where:

- $V = \Sigma$,
- $C = \emptyset$,
- $P = \{t_1, \dots, t_k, d, p\}$, with $d, p \notin \{t_1, \dots, t_k\}$,
- $L' = \phi(L)dp$ with the morphism ϕ defined by $\phi(t_i) = dt_i, 1 \leq i \leq k$,
- $\mu = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}_1$,
- $w_1 = \lambda; w_2 = w$,
- $R_1 = \emptyset$,
- $R_2 = \{X \rightarrow \alpha|_{t_i} \mid X \rightarrow \alpha \in h_i, 1 \leq i \leq k\} \cup \{N \rightarrow \#|_p \mid N \in \Sigma - T\}$,
- $i_0 = 2$.

Now Π simulates in region 2 the productions of G , applying the tables according to the strings in L' , in such a way that each table h_i has an associated promoter t_i , for every $1 \leq i \leq k$.

The dummy promoter d is only used to be consumed while the control string moves from region 1 to region 2. In this way, the new head of the control string is a symbol t_i , for some $1 \leq i \leq k$. The final promoter p added as last symbol of any string in L' is used to check whether or not there are still nonterminals in region 2 in the last step of the computation. If this is the case, then the special object $\#$ is produced and the computation is not successful. Consequently, the theorem follows. ■

We continue now to prove that the reverse inclusions also hold.

Theorem 10

$$(pro)N_{\#}^{(1)}P_*(ncoo, REG) \subseteq NE(rc)T0L = NET0L.$$

$$(pro)N_{\#}^{(1)}P_*(ncoo, CF) \subseteq NE(cfc)T0L.$$

Proof

Consider a fully-promoted SC P system Π of the form

$$\Pi = (V, C, P, L, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0),$$

such that $C = \emptyset$ and L is a regular (context-free, respectively) language over $P = \{p_1, p_2, \dots, p_k\}$.

We consider the morphisms φ_i , $1 \leq i \leq m$, defined by $\varphi_i(X) = (X, i)$, for all $X \in V$, $1 \leq i \leq m$. By using these morphisms, we associate with each occurrence of any object X the index of the region where the occurrence resides.

We also use the morphisms φ_i^t , $1 \leq i \leq m$, defined by

$$\varphi_i^t(X_{tar}) = \begin{cases} (X, i) & \text{if } tar = here, \\ (X, j) & \text{if } tar = out, \\ (X, k) & \text{if } tar = in_k, \end{cases}$$

for all $X \in V$, where j is the label of the surrounding region of i .

We construct now an E(rc)T0L system (or an E(cfc)T0L system, respectively) $\Omega = (G, L')$ simulating the computations of Π .

First we construct G . Let $G = (\Sigma, T, H, w')$, where $\Sigma = \{(X, i) \mid X \in V, 1 \leq i \leq m\}$, $T = \Sigma - \{(\#, i) \mid 1 \leq i \leq m\}$ and $w' = \varphi_1(w_1) \cdots \varphi_m(w_m)$.

Each table $h_{i,p_j} \in H$, $1 \leq i \leq m$, $1 \leq j \leq k$, is constructed in the following way:

- for each $X \in V$, if $X \rightarrow \alpha|_{p_j} \in R_i$, for some $p_j \in P$, then the rule $(X, i) \rightarrow \varphi_i^t(\alpha)$ is added to the table h_i . Otherwise, if X is not present as the left hand side of any rule in R_i , then the rule $(X, i) \rightarrow (X, i)$ is added to the table h_i ;
- for each $X \in V$ and $1 \leq l \leq m$, $l \neq i$, the rule $(X, l) \rightarrow (X, l)$ is added to the table h_i .

Notice that H has mk tables and each one of them is complete.

Finally we construct L' . To this aim we define the finite substitution φ' by $\varphi'(p_j) = \{t_{(i,p_j)} \mid 1 \leq i \leq m\}$ for each $1 \leq j \leq k$. We also define the nonde-terministic finite state automaton $A = (Q, V_A, s_0, F, \delta)$, where $Q = \{0, 1, \dots, m\}$, $V_A = \{t_{(i,p_j)} \mid 1 \leq i \leq m, 1 \leq j \leq k\}$, $s_0 = 0$, $F = Q$ and δ is defined by $\delta(0, t_{(1,p_j)}) = 1$, $\delta(i_1, t_{(i_1,p_j)}) = \{i_2 \mid 1 \leq i_2 \leq m, \text{ and region } i_1 \text{ is adjacent to region } i_2 \text{ in } \mu\}$ for every $1 \leq j \leq k$ and $1 \leq i_1 \leq m$. Without loss of generality, we assume 1 to be the label of the skin membrane of Π .

Now, $L' = \varphi'(L) \cap L(A)$ is regular (context-free, respectively) since regular (context-free, respectively) languages are closed under intersection with regular languages, see e.g. [25].

The underlying idea of the proof is the following.

Each table $t_{(i,p_j)}$ of G with $1 \leq i \leq m, 1 \leq j \leq k$, simulates the rewriting in parallel of the objects present in region i of Π , by using rules activated by the promoter p_j . All the objects present in the same region that cannot be rewritten by any active rule, as well as those present in the other regions of the system, are left unchanged by the application of the table.

The language $\varphi'(L)$ is used to pass from one table to another, in the way described by the strings of promoters present in the control program L . More specifically, if the string $w = p_{j_1} \cdots p_{j_l}$ is present in L , then $\varphi'(L)$ contains all the strings of the set $S_w = \{t_{(i_1,p_{j_1})}, \dots, t_{(i_l,p_{j_l})} \mid i_1, \dots, i_l \in \{1, \dots, m\}\}$. In this way, each computation of Π starting with the control string $w = p_{j_1} \cdots p_{j_l}$ can be simulated in G by applying the tables following the order of an appropriate string in S_w . On the other hand, not every string in the set S_w simulates a correct computation in Π starting with the control string $p_{j_1} \cdots p_{j_l}$. In fact, the control string in Π can only move through adjacent regions – this has to be “encoded” in the way that the passage from one table of G to another one is done. For this reason the appropriate regular (context-free, respectively) language L' that controls G is obtained by intersecting the language $\varphi'(L)$ with the regular language $L(A)$.

From the above explanation it follows that each string in $L(\Omega)$ contains pairs (*object, region*) corresponding to the objects present in the halting configurations of successful computations of Π . In order to get the exact contents of the output region of Π , we apply to $L(\Omega)$ the morphism φ_o , defined by:

$$\varphi_o((X, i)) = \begin{cases} X & \text{if } i = i_0, \\ \lambda & \text{otherwise.} \end{cases}$$

Since the family $E(rc)T0L$ (or the family $E(cfc)T0L$, respectively) is clearly closed under arbitrary morphisms, it follows that $N_{\#}^{(1)}(\Pi)$ belongs to the family $NE(rc)T0L$ (or to the family $NE(cfc)T0L$, respectively). Thus the theorem holds. ■

From Theorems 9 and 10 we obtain

Corollary 11

$$\begin{aligned} (pro)N_{\#}^{(1)}P_*(ncoo, REG) &= NE(rc)T0L = NET0L. \\ (pro)N_{\#}^{(1)}P_*(ncoo, CF) &= NE(cfc)T0L. \end{aligned}$$

On the other hand, if SC P systems collect the result in the standard way, then one gets the following results.

Theorem 12

$$(pro)N^{(1)}P_2(ncoo, REG) \supseteq N(rc)T0L = NET0L.$$

$$(pro)N^{(1)}P_2(ncoo, CF) \supseteq N(cfc)T0L.$$

Proof

In the proof of Theorem 9 the special symbol $\#$ is only used to check if any nonterminal of G is still present when the computation of Π halts. Therefore this checking can be avoided during the simulation of a (rc)T0L system (or a (cfc)T0L system, respectively). Hence the theorem holds. ■

Analogously, note that in Theorem 10 the set of nonterminals used by the ET0L system constructed in the proof contains only the special object $\#$ included in the alphabet of the corresponding SC P system Π . Therefore if Π collects the output in the standard mode (i.e., it does not use $\#$), then one gets the following results.

Theorem 13

$$(pro)N^{(1)}P_*(ncoo, REG) \subseteq N(rc)T0L = NET0L.$$

$$(pro)N^{(1)}P_*(ncoo, CF) \subseteq N(cfc)T0L.$$

Theorems 12 and 13 yield the following corollary.

Corollary 14

$$N(rc)T0L = (pro)N^{(1)}P_*(ncoo, REG) = NET0L.$$

$$N(cfc)T0L = (pro)N^{(1)}P_*(ncoo, CF).$$

7.6 Fully-Promoted SC P Systems: Universality

If SC P systems use arbitrary regular control programs, and only one catalyst, then they generate the family of recursively enumerable sets of natural numbers.

In [13], P systems using two catalysts and two membranes have been proved to be universal. This proof can also be applied for non fully-promoted SC P systems to obtain the following universality result.

Corollary 15

$$N_{\#}^{(1)}P_2(cat_2, \{\{\lambda\}\}) = NRE.$$

In case of fully-promoted SC P systems, the computational universality can be obtained using arbitrary regular control programs and catalytic rules with only one catalyst.

Theorem 16

$$(pro)N_{\#}^{(1)}P_2(cat_1, REG) = NRE.$$

Proof

The inclusion in NRE follows from Church-Turing thesis. The opposite inclusion can be proved by simulating regularly controlled grammars with appearance checking, as follows.

Given a regularly controlled grammar with appearance checking $G = (N, T, S, P, K, F)$, we construct $\Pi \in (pro)P_2^{(1)}(cat_1, REG)$, collecting the output in the $\#$ way, that simulates G . Let

$$\Pi = (V, C, P', L, \mu, w_1, w_2, R_1, R_2, i_0),$$

where:

- $V = N \cup T \cup \{c, Z\}$,
- $C = \{c\}$,
- $P' = lab(P) \cup \{d, d'\}$, $d, d' \notin lab(P)$,
- $L = \phi(K)dd'$ with non-erasing morphism ϕ defined by $\phi(p) = dp$ for each $p \in lab(P)$,
- $\mu = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$,
- $w_1 = \lambda; w_2 = SZc$,
- $R_1 = \emptyset$,
- $R_2 = \{cA \rightarrow c\alpha|_p \mid p : A \rightarrow \alpha \in P\} \cup \{cZ \rightarrow c\#|_p \mid p \notin F\} \cup \{Z \rightarrow Z_{out}|_{d'}\}$,
- $i_0 = 2$.

We show that Π simulates the derivations of G . Note that, by definition, for every $p_{i_1} \cdots p_{i_k} \in K$, we have $dp_{i_1} \cdots dp_{i_k}dd' \in L$. The promoters d are dummies, they are only used to let the control string to enter and exit region 2, passing in this way from a promoter p_{i_j} as the current head to the promoter $p_{i_{j+1}}$, for $1 \leq j \leq k-1$. The derivations of G are simulated by the execution of rules from R_2 .

Notice that, because of the catalyst c that inhibits the parallelism, at most one rule is executed in region 2 when the control string resides in that region. If a rule cannot be applied and the label of the corresponding production is not in F , then the computation is unsuccessful ($\#$ is produced by applying the rule $cZ \rightarrow c\#$ that is activated by any promoter $p \in (lab(P) - F)$) and this is correct since the simulated derivation in G cannot be continued. On the other hand, if a rule cannot be applied and the label of the corresponding production is in F (so the production has to be used in the appearance checking mode), then no rule is applied in region 2, the control string leaves the region and the computation continues. The last promoter d' present for any control string in L is used to move, at the end of the computation, the symbol Z into region 1. It should be clear from the above description that $N_{\#}^{(1)}(\Pi)$ is exactly the length set of $L(G)$. Thus the theorem holds. ■

We conclude this section by presenting some preliminary results concerning the class of non fully-promoted SC P systems.

By definition, it is clear that

Lemma 17

$$\begin{aligned}(pro)N_{\#}^{(i)}P_m(\alpha, FL) &\subseteq N_{\#}^{(i)}P_m(\alpha, FL), \\ (pro)N^{(i)}P_m(\alpha, FL) &\subseteq N^{(i)}P_m(\alpha, FL),\end{aligned}$$

for $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}$, FL a family of languages, and $i \in \{1, 2\}$.

It is easy to notice that systems from $P_1(ncoo, FIN)$ can generate infinite sets of numbers when operating in mode (1) and collecting the result in the standard way. This observation and Theorem 8 yield the following result.

Theorem 18

$$(pro)N_{\#}^{(1)}P_*(\alpha, FIN) = (pro)N^{(1)}P_*(\alpha, FIN) \subset N^{(1)}P_*(\alpha, FIN),$$

for $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}$.

On the other hand, if one can use any RE language as the control program, then both classes of SC P systems have the same computational power. In particular, from Theorem 7 and Corollary 17, one gets the following result.

Theorem 19

$$(pro)N^{(1)}P_m(\alpha, RE) = N^{(1)}P_m(\alpha, RE) = NRE,$$

for $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}$.

7.7 Concluding Remarks and Open Problems

We have introduced and investigated SC P systems where the computations are driven by control strings (present in their skin region at the beginning of computations). We have mainly investigated fully-promoted SC P systems, where all the rules are promoted (hence controlled by the control strings). Most of the results proved in this chapter concern systems operating in mode (1), although this is just a matter of convenience, because we have proved the equivalence between both operating modes (under some conditions).

Table 7.1 gives an overview of the results obtained for fully-promoted SC P systems operating in mode (1) and collecting the result in the $\#$ way.

The results obtained for fully-promoted SC P systems operating in mode (1) and collecting the result in the standard way are summarized in Table 7.2.

Several problems, mainly concerning non fully-promoted systems, remain open. Are non-fully promoted SC P systems more powerful than fully-promoted SC P

	RE	CF	REG	FIN
<i>ncoo</i>	<i>NRE</i>	<i>NE(cfc)TOL</i>	<i>NETOL</i>	<i>NFIN</i>
<i>cat_i, i ≥ 1</i>	<i>NRE</i>	<i>NRE</i>	<i>NRE</i>	<i>NFIN</i>
<i>coo</i>	<i>NRE</i>	<i>NRE</i>	<i>NRE</i>	<i>NFIN</i>

Table 7.1: Computational power of fully-promoted SC P systems operating in mode (1) and collecting the result in the # way. Rows specify the types of evolution rules, and the columns specify the types of control programs.

	RE	CF	REG	FIN
<i>ncoo</i>	<i>NRE</i>	<i>N(cfc)TOL</i>	<i>N(rc)TOL</i>	<i>NFIN</i>
<i>cat_i, i ≥ 1</i>	<i>NRE</i>	$\supseteq N(cfc)TOL$	$\supseteq N(rc)TOL$	<i>NFIN</i>
<i>coo</i>	<i>NRE</i>	$\supseteq N(cfc)TOL$	$\supseteq N(rc)TOL$	<i>NFIN</i>

Table 7.2: Computational power for fully-promoted SC P systems operating in mode (1) and collecting the result in the standard way. Again, rows specify the types of evolution rules, and the columns specify the types of control programs.

systems? The answer is positive for SC P systems operating in mode (1) and having a finite control program (Theorem 18). We conjecture that the strict inclusion also holds when the control program is regular and the result is collected in the standard way.

Another open problem is to find a non-trivial upper bound for the generative power of fully-promoted SC P systems operating in mode (1), collecting the result in the standard way, and using cooperative or catalytic rules (see Table 7.2). We only know that these classes of systems can generate at least the family of length sets of languages from $(rc)TOL$ (if the control program is regular) and from $(cfc)TOL$ (if the control program is context-free). We doubt that these two classes are universal – as a matter of fact they may be incomparable with the classical Chomsky classes.

Finally, another interesting issue to be investigated is having the control programs produced by another bio-inspired generative device (as for instance, another membrane system, or a DNA-based system).

Chapter 8

Communication Membrane Systems with Active Symports

Abstract

We consider membrane systems where the generation/transformation of objects can take place only if it is linked to communication rules.

More specifically, all the rules move objects through membranes and, moreover, the membranes *can* modify the objects as they pass through. The intuitive interpretation of such rules is that a multiset of objects can move from a region to an adjacent one, and moreover objects can engage into (biochemical) reactions while passing through (are in “contact” with) a membrane. Therefore such “twofold” rules are called symport-rewriting (in short, *sr*) rules, where symport refers to a coordinated passage of a “team” of molecules through a membrane.

In this chapter we investigate the influence of the form of *sr* rules on the power of membrane systems that employ them (sometime in combination with simple antiport rules which allow a synchronized exchange, through a membrane, of two molecules residing in two adjacent regions). A typical restriction on the form of an *sr* rule requires that the passage described by the rule is such that the sort of exiting molecules is a subset of the sort of entering molecules (however the multiplicities of sorts do not have to be related).

We also compare the sequential passage mode with the maximally parallel passage mode.

8.1 Introduction

Membrane computing, introduced in [20], is a computational model inspired by the functioning of membranes in living cells. The biological membranes within a cell divide the cell in a number of compartments (regions). This is the basic feature of this model with each region containing its own set of evolution rules,

where each rule prescribes both the transformation and generation of molecules as well as the transport of molecules through membranes. In this way each evolution rule has both a rewriting and a communication component.

An important class of membrane systems allows only communication, i.e., their rules prescribe only the passage of objects (molecules) through membranes. Such systems are called symport/antiport membrane systems where both “symport” and “antiport” refer to types of rules that allow for a synchronized passage of molecules through a membrane. For symport rules this passage is unidirectional (a multiset of molecules is passing through a membrane together), while for antiport rules this passage is bidirectional (a passage of a multiset of molecules in one direction is synchronized with a passage of molecules in another direction through the same membrane).

In this chapter we enrich symport rules by coupling them with a generative component: a multiset of molecules passing a membrane synchronously in the unidirectional fashion can be changed to a different multiset. Such rules are called *symport-rewriting rules*, or *sr rules* for short – they are biologically motivated, as the biological membranes do not only allow for the passage of molecules, but can also change them *during* a passage. Membrane systems using sr rules and antiport rules are called *communication membrane systems with active symports*, or CAS P *systems* for short.

In this chapter we study the influence of the form of sr rules (sometimes in combination with antiport rules) on the generative power of resulting membrane systems. We also study the influence of the communication mode (sequential versus maximally parallel) on the generative power. In sequential mode, at any given time, at most one sr rule can be active for any given membrane, while (as usual) in maximally parallel mode an application of the rules is such that no more rules can be applied to the objects that are not already involved in the passage through membranes.

The chapter is organized as follows. Section 8.2 recalls some basic notions from language theory – more specifically those of matrix grammars and register machines. Section 8.3 formally defines CAS P systems. Section 8.4 considers “alphabetically restricted” CAS P systems which can use only sr rules in which the type of every generated object (i.e., an object occurring at the right hand side) is already present at the “entrance to the membrane” (i.e., occurring at the left hand side). In Section 8.5 we consider CAS P systems operating in sequential mode. Section 8.6 considers CAS P systems that use only unary rules, i.e., the rules that describe the passage of one single molecule and allow this molecule to multiply during this passage (technically, this corresponds to the rules $a \rightarrow v$, where $v \in a^*$).

A natural restriction on communication can be formulated through the use of uni-directional membranes, i.e., membranes such that, for any symbol a , if a may cross a given membrane in one direction (as specified by one of the rules), then a *cannot* cross this membrane in the opposite direction. These systems are considered in Section 8.7. Unary rules are a special case of non-cooperative rules,

i.e., rules that describe the passage of a single object (which during the passage can be changed into a multiset of objects). CAS P systems using only non-cooperative rules are considered in Section 8.8 both with and without the use of antiport rules. In Section 8.9 we consider *bounded* CAS P systems, i.e., CAS P systems for which there exists a positive integer k such that in any computation and any region the cardinality of the multiset of objects present in the region does not exceed k . In the last section we discuss the results obtained in this chapter and we formulate a number of open problems.

8.2 Preliminaries

We assume the reader to be familiar with basic notions of formal languages and automata theory (which can be found, e.g., in [25]). In this section we briefly recall some notions and results concerning matrix grammars and register machines that will be used in some proofs in this chapter.

8.2.1 Matrix Grammars

A *matrix grammar* is a construct $G = (N, T, S, M, F)$, where N is the nonterminal alphabet, T is the terminal alphabet ($N \cap T = \emptyset$), $S \in N$ is the axiom, M is a finite set of sequences (called *matrices*) of context-free productions ($A_1 \rightarrow x_1, \dots, A_n \rightarrow x_n$), $n \geq 1$ with $A_i \in N$, $x_i \in (N \cup T)^*$, $1 \leq i \leq n$, and F is a set of occurrences of rules in M (note that one may have the same rule in different entries of a matrix and only some of these entries in F).

Given two strings $w \in (N \cup T)^* N (N \cup T)^*$ and $z \in (N \cup T)^*$, we write $w \Rightarrow z$ if there is a matrix $(A_1 \rightarrow x_1, \dots, A_n \rightarrow x_n) \in M$ and there exist strings $w_i \in (N \cup T)^*$, for $1 \leq i \leq n+1$, such that $w = w_1$, $z = w_{n+1}$ and, for each $1 \leq i \leq n$, either $w_i = w'_i A_i w''_i$ and $w_{i+1} = w'_i x_i w''_i$ for some $w'_i, w''_i \in (N \cup T)^*$, or $w_i = w_{i+1}$, A_i does not appear in w_i , and (the given occurrence of) production $A_i \rightarrow x_i$ appears in F . Thus, the productions of a matrix are applied in the order in which they are listed, except that one skips the rules in F if they cannot be applied – therefore we say that these productions are applied in the *appearance checking mode*. If the set F is empty, then G is said to be *without appearance checking*. The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. The family of languages generated by matrix grammars with appearance checking is denoted by MAT_{ac} . The family of languages generated by matrix grammars without appearance checking is denoted by MAT . It is known (see [24, Chapter 12]) that $MAT \subset MAT_{ac} = RE$.

We say that a matrix grammar is *pure* if there is no distinction between terminals and nonterminals, i.e., each string derived from S belongs to $L(G)$. The family of languages generated by pure matrix grammars without appearance checking is denoted by $pMAT$. It is known (see [11, Lemma 5.1.1]) that $pMAT \subset MAT$.

8.2.2 Register Machines

Intuitively, a register machine is an automaton with a number of registers (each storing a natural number) that is executing labelled instructions of several simple types. More precisely:

A *register machine* is a construct $M = (k, \mathcal{P}, l_0, l_h)$, where:

- k is the number of registers,
- $\mathcal{P}$ is a set of labelled instructions (the *program*) that can be of the following forms:
 1. $(l : \text{add}(r), l_i, l_j)$,
 2. $(l : \text{sub}(r), l_i, l_j)$,
 3. $(l_h : \text{halt})$,
 with l, l_h, l_i, l_j from the set $\text{lab}(\mathcal{P})$ of labels associated with the instructions in a one-to-one manner,
- $l_0 \in \text{lab}(\mathcal{P})$ is the label of the initial instruction,
- $l_h \in \text{lab}(\mathcal{P})$ is the label of the halting instruction.

The execution of an instruction $(l : \text{add}(r), l_i, l_j)$ increments by one the value stored in register r and then the machine proceeds, in a nondeterministic way, either to the instruction with label l_i or to the instruction with label l_j . An instruction $(l : \text{sub}(r), l_i, l_j)$ is executed as follows. If the value stored in register r is not zero, then it subtracts one from this value, and the machine proceeds to the instruction labelled by l_i ; otherwise it proceeds to the instruction labelled by l_j . A halting instruction $(l_h : \text{halt})$ stops the machine; its label is always the final label l_h .

We say that a vector $(n_1, \dots, n_\alpha) \in \mathbb{N}^\alpha$ is *generated* by M (where α is fixed for M) if, starting from the instruction labelled by l_0 with the value of all registers equal to zero, it halts with value n_j in register j for all $1 \leq j \leq \alpha$, and with the values of the registers $\alpha + 1, \dots, k$ equal to zero. The set of all vectors obtained in this way constitutes the set generated by M . The family of all sets of vectors generated by register machines is denoted by RegM . It is known (see [19]) that register machines can generate the family of Turing computable sets of vectors of natural numbers, that is, $\text{RegM} = \text{PsRE}$.

A *register machine without checking for zero* is a register machine where all subtraction instructions are of the form $(l : \text{sub}(r), l_j, l_{h*})$, where l_{h*} is the non-successful halting label – the machine stops but the computation is not considered successful. The family of languages generated by register machines without checking for zero is denoted by $\text{RegM}_{\neq 0}$. The computational power of this model is the same, in terms of Parikh images of languages, as that of matrix grammars without appearance checking. That is, $\text{RegM}_{\neq 0} = \text{PsMAT}$ (see, e.g., [12]).

8.3 Communication Membrane Systems with Active Symports

In what follows we assume that the reader is familiar with the area of membrane computing, in particular with membrane systems using symport/antiport rules, see, e.g., [21]. We will still use a rather informal terminology describing the nested relationship between membranes or regions, such as, e.g., (immediately) inner membrane of membrane i or (immediately) outer membrane of membrane i , however, if needed, this can be always made precise by, e.g., considering a tree representation of the nested structure of membranes (thus an immediately inner membrane of membrane i becomes a *direct descendent* of membrane i).

The model of membrane systems studied in this chapter is defined as follows.

Definition 1

A *communication membrane system with active symports* (in short, a *CAS P system*) is a construct

$$\Pi = (\Gamma, \mu, w_1, \dots, w_m, R_1, \dots, R_m, R_1^a, \dots, R_m^a, i_0),$$

where:

- Γ is the alphabet of objects,
- μ is a tree structure representing a membrane structure with m membranes (labelled in a one-to-one manner by $1, \dots, m$),
- $w_1, \dots, w_m$ are the multisets of objects initially present in the regions of the system,
- $R_1, \dots, R_m$ are finite sets of symport-rewriting (in short, sr) rules associated with membranes $1, 2, \dots, m$, respectively; each rule is of the form (u, v, out) or (u, v, in) , where $u \in \Gamma^+, v \in \Gamma^*$,
- $R_1^a, \dots, R_m^a$ are finite sets of antiport rules associated with membranes $1, 2, \dots, m$, respectively; each rule is of the form $(u, in; v, out)$ where $u \in \Gamma^+, v \in \Gamma^+$,
- $i_0 \in \{1, \dots, m\}$ is the label of the output membrane of Π . ■

We also use Γ_Π to denote Γ . As usual, the root of μ is called the *skin membrane*. The skin membrane separates the system from the *environment*. The leaves of μ are called *elementary*. Each membrane i delimits a *region i*: it is the space between it and its direct descendants. The *surrounding region* of membrane i is the environment if i is the skin membrane, and otherwise it is the region of the direct ancestor of i .

We now define the semantics of symport-rewriting rules. An sr rule $r = (u, v, out) \in R_i$, $1 \leq i \leq m$, can only be applied if multiset u is present in

region i (we say then that rule r is *applicable*). The effect of applying this rule is as follows: multiset u is deleted from region i and simultaneously multiset v is added to its surrounding region. This formalizes the following intuition: multiset u is moved *out of* region i by crossing through membrane i and *while* crossing, it is transformed into multiset v . An sr rule (u, v, in) from the set R_i is applied analogously, however multiset u is moved *into* region i by crossing through membrane i and while crossing u is transformed into multiset v . Rules $(u, v, out) \in R_i$ and $(u, v, in) \in R_i$ will also be denoted by $u \dashv_i v$ and $u \dashv_i^+ v$, respectively.

The (standard) effect of applying an *antiport rule* $(x, in; y, out) \in R_i^a$, $1 \leq i \leq m$, is as follows: multiset x crosses membrane i from the surrounding region into region i , while, at the same time, multiset y moves in the opposite direction through membrane i .

Before going on, it is worth to remark that CAS P systems do not assume a potentially infinite supply of objects available in the environment, although this is the case in the classical definition of symport/antiport P systems, which have a set E of objects available in the environment in an unbounded number of copies.

In this chapter we consider several restrictions for the sr rules of CAS P systems, and so we give now notation and terminology to describe these restrictions. Symport-rewriting rules (u, v, out) and (u, v, in) are called *cooperative* if $|u| \geq 2$, and *noncooperative* if $|u| = 1$. Also, sr rules are called *alph-restricted* if $alph(v) \subseteq alph(u)$, where $alph(x)$ is the smallest alphabet $\Psi \subseteq \Gamma$ such that $x \in \Psi^*$. Thus, such sr rules cannot introduce new types of objects. Noncooperative alph-restricted sr rules are also called *unary* sr rules (because there is only one type of object, one symbol, present in these rules). The *weight* of an antiport rule $(u, in; v, out)$ is defined as $\max\{|u|, |v|\}$.

As usual, a *configuration* of a membrane system is an instantaneous description of the membrane structure and the contents of all the regions. The *initial configuration* consists of the membrane structure μ and the multisets of objects initially present in the regions of the system, given by $w_1, \dots, w_m$.

The system evolves from one configuration to another by performing a *transition step*. In one mode of operation, the most usual one for P systems, the transition steps are performed by applying the rules in a nondeterministic, maximally parallel manner. However, we will also consider another mode of operation, called *sequential*, where no antiport rules are present and *at most one* sr rule is applied at each step for each membrane, allowing a membrane to be inactive even when there is an applicable sr rule.

All the possible sequences (finite or infinite) of transition steps that the system is able to perform from the initial configuration are the *computations* of the system. A given configuration is called *reachable* if it results from a computation of the system. A reachable configuration is a *halting configuration*, if there is no rule applicable to it. The *output* of a CAS P system, denoted by $Ps(\Pi)$, is the set of vectors of natural numbers which are the Parikh images of the multisets present in region i_0 in all possible halting configurations.

We also want to make a comment concerning the outputs of computations in

membrane systems. They are traditionally either vectors expressing the multiplicities of various objects present in the output region at the conclusion of a successful computation or numbers expressing the cardinality of *all* objects present in the output region at the conclusion of a successful computation. The common dimension of the vectors is the cardinality of a fixed a priori alphabet of output objects. This alphabet may be different from the total alphabet Γ of objects, however it is not *explicitly* given. This also happens in this chapter, however as usual the output alphabet is always well understood from the context of considerations.

Example 1

It is easy to see that the (unique) halting configuration of the CAS P system Π from Figure 8.1 is (in both sequential and maximally parallel mode) $[[]_2 [aaaaaaa]_3]_1$ and so, we have $Ps(\Pi) = \{(8)\}$.

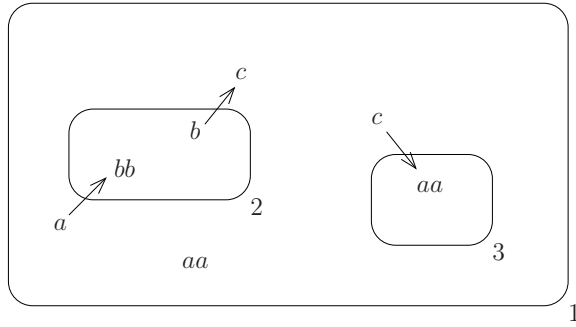


Figure 8.1: A graphic representation of a communication membrane system Π with active symports (CAS P system). The initial configuration is $[[]_2 []_3 a a]_1$. The output region is the one enclosed by membrane 3 and the output alphabet is $\{a\}$. The sets of sr rules associated with regions 2 and 3 are: $R_2 = \{b \rightarrow_2 c, a \rightarrow_2 bb\}$ and $R_3 = \{c \rightarrow_3 aa\}$, respectively.

We denote by $CAS(\alpha, ant_i)$ the class of CAS P systems with sr rules of type α , and antiports of weight at most i . In this chapter we consider $\alpha \in \{coo, ncoo, coo_{AR}, ncoo_U\}$ to denote general (non-restricted), noncooperative, cooperative alph-restricted, and unary sr rules, respectively. We use $PsCAS_m(\alpha, ant_i)$ to denote the family of sets of vectors computed by CAS P systems from the class $CAS(\alpha, ant_i)$ with at most m membranes. Besides, we denote $PsCAS_*(\alpha, ant_i) = \bigcup_{m \in \mathbb{N}} PsCAS_m(\alpha, ant_i)$. Moreover, when we consider systems working in the sequential mode we will add the prefix *seq* (obtaining in this way the notation $seqPsCAS_m(\alpha, ant_i)$).

In what follows, to simplify the notation, in the definition of CAS P systems that do not use antiport rules we will omit the specification of the sets of antiport rules.

8.4 Alphabetic Restriction

In this section we investigate systems using *cooperative alph-restricted* sr rules, coo_{AR} sr rules for short, *without* antiport rules. We show that this restriction does not decrease the generative power of the model – CAS P systems using coo_{AR} sr rules are computationally universal.

Theorem 2

$PsCAS_2(coo_{AR}, ant_0) = PsRE$.

Proof

In order to demonstrate the computational universality of the model, we will prove that any register machine can be simulated by it. To this aim, consider an arbitrary register machine $M = (k, \mathcal{P}, l_0, l_h)$, generating a set of vectors over $\mathbb{N}^\alpha$ (i.e., registers from 1 to α are *output registers* and registers from $\alpha + 1$ to k are *working registers*), where $lab(\mathcal{P})$ consists of n labels. We construct the CAS P system $\Pi = (\Gamma, \mu, w_1, w_2, R_1, R_2, i_0)$ satisfying the required conditions that generates the same set of vectors as M does, as follows.

- $\Gamma = \{c_i \mid 1 \leq i \leq k\} \cup \{l_i, l'_i, l''_i, l'''_i \mid 0 \leq i \leq n-1\}$.
- $\mu = [\quad]_2$.
- $w_1 = \lambda$ and $w_2 = l_0 c_1 \dots c_k l_0 l'_0 l''_0 l'''_0 \dots l_{n-1} l'_{n-1} l''_{n-1} l'''_{n-1}$.
- $R_1 = \emptyset$,
- The set R_2 of cooperative alph-restricted sr rules is defined as follows:
 - for every instruction $(l_{i_1} : add(j), l_{i_2}, l_{i_3}) \in \mathcal{P}$, R_2 contains the sr rules $l_{i_1} l_{i_1} l_{i_2} c_j \xrightarrow{-1_2} l_{i_2} l_{i_2} l_{i_1} c_j c_j$ and $l_{i_1} l_{i_1} l_{i_3} c_j \xrightarrow{-1_2} l_{i_3} l_{i_3} l_{i_1} c_j c_j$,
 - for every instruction $rs = (l_{i_1} : sub(j), l_{i_2}, l_{i_3}) \in \mathcal{P}$, R_2 contains the sr rules

$$rs_1 = l_{i_1} l_{i_1} l'_{i_1} l''_{i_1} \xrightarrow{-1_2} l_{i_1} l'_{i_1} l'_{i_1} l''_{i_1} l''_{i_1} l''_{i_1},$$

$$rs_2 = l'_{i_1} l'_{i_1} l'_{i_1} l'_{i_1} \xrightarrow{-1_2} l'_{i_1} l'_{i_1} l'_{i_1},$$

$$rs_3 = l''_{i_1} l''_{i_1} l''_{i_1} c_j c_j \xrightarrow{-1_2} l''_{i_1} l''_{i_1} l''_{i_1} c_j,$$

$$rs_4 = l'''_{i_1} l'''_{i_1} l'''_{i_1} l'''_{i_1} l_{i_2} \xrightarrow{-1_2} l_{i_2} l_{i_2} l'''_{i_1} l'''_{i_1},$$

$$rs_5 = l'''_{i_1} l'''_{i_1} l'''_{i_1} l'''_{i_1} l_{i_3} \xrightarrow{-1_2} l_{i_3} l_{i_3} l'''_{i_1}.$$
 - in addition, R_2 contains the following sr rules:

$$x \xrightarrow{-1_2} x, \text{ for every } x \in \Gamma, \text{ and}$$

$$rs_6 = l_h c_1 \dots c_k l_0 l'_0 l''_0 l'''_0 \dots l_{n-1} l'_{n-1} l''_{n-1} l'''_{n-1} \xrightarrow{-1_2} \lambda.$$
- $i_0 = 2$.

The simulation of M by Π is performed as follows. The current label of M is represented by exactly one symbol l_i that appears in region 2 with multiplicity two. Therefore, the initial label l_0 of M appears twice in w_2 . The number n_i stored in a register i of M is encoded in Π by multiplicity $n_i + 1$ of symbol c_i . The

reason for these encodings is that since only coo_{AR} sr rules are available, during the computation, the multiplicity of c_i and l_i may never be zero. Thus, only one copy of c_j corresponds to the case that register j is zero. Since in the initial configuration of M every register is zero, each symbol c_i appears in multiplicity one in w_2 .

The interpretation of the sr rules that simulate the addition instruction $(l_{i_1} : add(j), l_{i_2}, l_{i_3}) \in \mathcal{P}$ is quite straightforward: we modify the amount of objects l_{i_1} and l_{i_2} (or l_{i_3}) to simulate the transfer of control from l_{i_1} to l_{i_2} (or to l_{i_3} , respectively). We also generate one more copy of object c_j to simulate the increment by 1 of the value of register j . Note that by applying this sr rule we move objects from region 2 to region 1. In the next step all the objects are put back to region 2 without modifying their multiplicities. This is accomplished by sr rules $x \xrightarrow{+2} x$, for every $x \in \Gamma$.

The simulation of the subtraction instruction is more complicated since we need to check whether or not the value stored in the specified register is zero in order to select the next control label of the machine. Given $rs = (l_{i_1} : sub(j), l_{i_2}, l_{i_3}) \in \mathcal{P}$, we proceed in three steps (as before, after each of these steps all objects are put back into region 2) depending on whether or not the value of register j in $\mathcal{P}$ is zero (or, equivalently, on whether or not the multiplicity of c_j in Π is one):

step	register j is not zero	register j is zero
1	rs_1	rs_1
2	rs_2, rs_3	rs_2
3	rs_4	rs_5

The end result is that the multiplicity of l_{i_1} is changed from two to one, and the multiplicity of l_{i_2} (l_{i_3}) is changed from one to two when the value stored in register j was not zero (was zero, resp.). Moreover, when register j was not zero, the multiplicity of c_j was decreased by one.

Thus, Π simulates M step by step until arriving at the label l_h (in case of a successful computation). Then, Π performs one last step deleting all auxiliary objects through the application of sr rule rs_6 . Thus, in the halting configuration region 1 will be empty, and the multiplicity of objects c_j , $1 \leq j \leq \alpha$, in region 2 represents exactly the contents of the corresponding output registers of the machine in the halting state. Consequently, since $RegM = PsRE$ we have proved that $PsCAS_2(coo_{AR}, ant_0) \supseteq PsRE$. Invoking the Turing-Church thesis we obtain also the converse inclusion; hence $PsCAS_2(coo_{AR}, ant_0) = PsRE$. ■

Note that the maximal parallelism of the system is fundamental in the above proof, as it allows to simulate the zero checking in a register. Indeed, rule rs_3 is applicable in step 2 iff the value stored in the corresponding register is not zero.

Remark

The above proof can be modified in order to decrease the cardinality of the alphabet of Π . Actually, all the elements in the set $\Lambda = \{l_i, l'_i, l''_i, l'''_i \mid i = 0, \dots, n-1\}$ can be encoded by using only two symbols. The idea is to establish a correspondence between elements from Λ and pairs of numbers, each pair being represented

by the multiplicities of two symbols (e.g., a and b). This must be done in such a way that if $l_1 \in \Lambda$ is represented by a smaller multiplicity of a than used for representing l_2 , then the multiplicity of b in the representation of l_1 must be greater than the multiplicity of b in the representation of l_2 . In this way, a given multiplicity of a and b (encoding $l_1 \in \Lambda$), can trigger only l_1 .

Corollary 3

$PsCAS_2(coo, ant_0) = PsRE$.

Proof

Since the set of CAS P systems using cooperative sr rules includes the set of CAS P systems using coo_{AR} sr rules, we have $PsCAS_2(coo_{AR}, ant_0) \subseteq PsCAS_2(coo, ant_0)$. Thus, the corollary follows from the previous theorem and the Turing-Church thesis. ■

8.5 The Sequential Mode

The application of sr rules in a maximally parallel way is a powerful way to regulate the communication. This fact will become even more clear in this section where we consider CAS P systems working in the sequential mode: in each membrane *at most one* sr rule is applied at each step, allowing a membrane to remain inactive even when there is an applicable sr rule for it. The notion of a sequential mode of operation for P systems with carriers is studied in [15], and the computational power of these systems is shown to be equal to the family of Parikh images of the languages generated by matrix grammars without appearance checking. We show that the generative power of CAS P systems with cooperative sr rules working in the sequential mode is also equal to $PsMAT$. First, we prove the result for cooperative alph-restricted sr rules, and later we extend it to cooperative sr rules.

Lemma 4

$seqPsCAS_*(coo_{AR}, ant_0) \subseteq PsMAT$.

Proof

Let $\Pi = (\Gamma, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0)$ be a CAS system with cooperative alph-restricted sr rules and working in the sequential mode. We prove that there exists a matrix grammar without appearance checking generating a language whose Parikh image is $Ps(\Pi)$.

First of all, as matrix grammars can only handle strings, we need to find a string-representation for the configurations of Π (which indicates for each object the region in which it occurs). To this aim, we introduce a notation that assigns to each object a pair (*object*, *location*) explicitly mentioning the region where the object resides. That is, for each object x present in region i , we include the pair (x, i) in the string. Note that the so-constructed string may contain many occurrences of a pair (x, i) – this represents the multiplicity of x in region i .

Note also that the order of such pairs in a string is not relevant, so any permutation of the string could be used as well. We will use *pure matrix grammars* to generate every reachable configuration of Π , and later filter out the halting configurations.

In order to correctly simulate the transition steps in Π by means of matrices of context-free productions, we have to avoid that the pairs (x, i) corresponding to objects produced in a given step are used to trigger rules in the *same* transition step. Furthermore, we have to take into account all the sr rules that are applied in Π (recall that Π works in the sequential mode, so at most one sr rule is applied in each membrane at each step). We will introduce one matrix for each possible applicable multiset of sr rules. That is, we will have a specific matrix to simulate each one of the $(|R_1|+1) \cdot (|R_2|+1) \cdots (|R_m|+1)$ possible combinations of selecting at most one sr rule from each one of the sets of sr rules associated with the m membranes of Π .

We are ready now to specify the pure matrix grammar without appearance checking $G_\Pi = (N, S, M)$ that simulates Π . Let $N = (\Gamma \times \{1, \dots, m\}) \cup \{S\} \cup \{A_{j,i} \mid 1 \leq i \leq m, 1 \leq j \leq |R_i|\}$ and $S \notin \Gamma$. The set of matrices M is defined as follows. Let $R_i = \{r_1^i, \dots, r_{n_i}^i\}$, for $1 \leq i \leq m$, and consider an arbitrary set C consisting of $c \leq m$ sr rules taken from $R_1, \dots, R_m$ (at most one from each set), $C = \{r_{j_1}^{i_1}, \dots, r_{j_c}^{i_c}\}$. For each sr rule $r_j^i : u \xrightarrow{i} v$ in C we define

$$\begin{aligned} avail(r_j^i) &= \{(x_1, i) \rightarrow A_{j,i}; (x_2, i) \rightarrow \lambda; \dots; (x_k, i) \rightarrow \lambda\}, \\ prod(r_j^i) &= \{A_{j,i} \rightarrow (y_1, i')(y_2, i') \cdots (y_{k'}, i')\}, \end{aligned}$$

where $u = x_1 \dots x_k$, $v = y_1 \dots y_{k'}$, and i' is the surrounding region of i (we proceed analogously for sr rules $u \xrightarrow{i} v$, by just exchanging i and i'). Then, for each C as above, we let M contain the matrix

$$(avail(r_{j_1}^{i_1}); \dots; avail(r_{j_c}^{i_c}); prod(r_{j_1}^{i_1}); \dots; prod(r_{j_c}^{i_c}))$$

that simulates the application of rules in C . The set M also includes a matrix $(S \rightarrow w_0)$, where w_0 is a string-representation of the initial configuration of Π .

Thus, we simulate Π by G_Π , in such a way that the language generated by G_Π corresponds to the string-representation of the reachable configurations of Π . Now, taking into account the inclusion $pMAT \subset MAT$ (cf. Section 8.2), we know that there exists a matrix grammar without appearance checking G such that $L(G) = L(G_\Pi)$.

Next, we need to filter out words from $L(G)$ corresponding to the halting configurations of Π . Note that the number of sr rules of Π is finite, and each one of them has a finite number of symbols on its left hand side. Therefore, the set of all configurations to which no rules of Π are applicable forms a regular language. In this way, it is possible to obtain the halting configurations of Π as the intersection of $L(G)$ and a regular language. Since the family of languages generated by matrix grammars without appearance checking is closed under intersection with regular languages (see [11], Lemma 1.3.5), we deduce that there exists a ma-

trix grammar without appearance checking G' such that $L(G')$ contains all the string-representations of the halting configurations of Π .

Finally, we apply the following erasing morphism over the set of halting configurations:

$$\gamma((x, i)) = \begin{cases} \lambda & \text{if } i \neq i_0, \\ x & \text{if } i = i_0, \end{cases}$$

and we get exactly the contents of the output region for every halting configuration, the Parikh image of which are the vectors generated by Π . Thus, taking into account that MAT is closed under morphisms (see [11], Theorem 1.3.1), we conclude that there exists a matrix grammar without appearance checking G'' that generates, in the Parikh image sense, the same set of vectors as Π . Therefore, we have that $seqPsCAS_*(coo_{AR}, ant_0) \subseteq PsMAT$. ■

Lemma 5

$PsMAT \subseteq seqPsCAS_2(coo_{AR}, ant_0)$.

Proof

Since $RegM_{\neq 0} = PsMAT$ (see Section 8.2) we prove that any register machine without checking for zero can be simulated by a CAS P system.

Consider an arbitrary register machine $M = (k, \mathcal{P}, l_0, l_h)$ with k registers and without checking for zero, generating a set of vectors over $\mathbb{N}^n$ (that is, registers from 1 to n are *output registers* and registers from $n+1$ to k are *working registers*). We define the CAS P system $\Pi = (\Gamma, \mu, w_1, w_2, R_1, R_2, i_0)$ as follows:

- $\Gamma = \{c_1, \dots, c_k, l, l_0, \dots, l_{n-1}\}$,
- $\mu = [\quad]_2^1$,
- $w_1 = \lambda, w_2 = l_0 l l_1 \dots l_{n-1} c_1 \dots c_k$,
- $R_1 = \emptyset$,
- the set R_2 of cooperative alph-restricted sr rules is defined as follows:
 1. for every instruction $(l_{i_1} : add(j), l_{i_2}, l_{i_3}) \in \mathcal{P}$, R_2 contains the following sr rules:

$$\begin{aligned} l_{i_1} l_{i_1} l_{i_2} c_j &\xrightarrow{+2} l_{i_1} l_{i_2} l_{i_2} c_j c_j, \\ l_{i_1} l_{i_2} l_{i_2} c_j c_j &\xrightarrow{-2} l_{i_1} l_{i_2} l_{i_2} c_j c_j, \\ l_{i_1} l_{i_1} l_{i_3} c_j &\xrightarrow{+2} l_{i_1} l_{i_3} l_{i_3} c_j c_j, \\ l_{i_1} l_{i_3} l_{i_3} c_j c_j &\xrightarrow{-2} l_{i_1} l_{i_3} l_{i_3} c_j c_j. \end{aligned}$$
 2. for every instruction $(l_{i_1} : sub(j), l_{i_2}, l_{h*}) \in \mathcal{P}$, R_2 contains the following sr rules: $l_{i_1} l_{i_1} l_{i_2} c_j c_j \xrightarrow{-2} l_{i_1} l_{i_2} l_{i_2} c_j$ and $l_{i_1} l_{i_2} l_{i_2} c_j \xrightarrow{-2} l_{i_1} l_{i_2} l_{i_2} c_j$.
 3. in addition, R_2 contains the following sr rules:

$$\begin{aligned} c_j &\xrightarrow{+2} c_j \text{ and } c_j \xrightarrow{-2} c_j \text{ for every } j \in \{n+1, \dots, k\}, \\ l &\xrightarrow{+2} l, l \xrightarrow{-2} l, \\ c_1 \dots c_k l_h l l_0 l_1 \dots l_{n-1} &\xrightarrow{+2} \lambda. \end{aligned}$$
- $i_0 = 2$.

Since we again deal with coo_{AR} sr rules, we use the same encodings for the contents of the registers and for the current label of M as we did in the proof of Theorem 2. The current label of M is represented by exactly one symbol l_i that appears in region 2 with multiplicity two (the others have multiplicity one), and the number n_i stored in a register i of M is encoded in Π by multiplicity $n_i + 1$ of symbol c_i .

The intuition behind the construction of Π is as follows. The addition instruction is simulated by just generating one more object for the corresponding register and replacing a copy of l_{i_1} nondeterministically by a copy of either l_{i_2} or l_{i_3} . For the subtraction instruction, if the register to which we try to apply the instruction is zero, then the rule $l_{i_1} l_{i_1} l_{i_2} c_j c_j \rightarrow l_{i_1} l_{i_2} l_{i_2} c_j$ cannot be applied. Instead of getting a non-accepting halting label l_{h^*} , in Π we avoid halting and accepting in Π by using the infinite loop given by the execution of the rules $l \rightarrow l$, $l \rightarrow l$. The only case when the loop can be interrupted is when the corresponding computation of M is successful, that is, if the label l_h is reached. In this case, Π deletes l , together with the additional elements c_j for all the registers by applying the rule $c_1 \dots c_k l_h l l_0 l_1 \dots l_{n-1} \rightarrow \lambda$. Then, we can get the output (the contents of the registers) from the multiset of objects present in region 2 in the halting configuration. Notice that if any of the working registers r_j with $j \in \{n+1, \dots, k\}$ is not zero when l_h is reached, then Π will not halt, because the corresponding rules $c_j \rightarrow c_j$ and $c_j \rightarrow c_j$ will run forever. ■

Theorem 6

$$seqPsCAS_2(coo_{AR}, ant_0) = seqPsCAS_*(coo_{AR}, ant_0) = PsMAT.$$

Proof

By definition, the inclusion $seqPsCAS_2(coo_{AR}, ant_0) \subseteq seqPsCAS_*(coo_{AR}, ant_0)$ holds. By Lemma 4 and Lemma 5 we have the desired result. ■

Corollary 7

$$seqPsCAS_2(coo, ant_0) = seqPsCAS_*(coo, ant_0) = PsMAT.$$

Proof

The proof follows easily from the previous theorem. On the one hand, the same construction of the matrix grammar used for proving Lemma 4 can be used for the general cooperative case. On the other hand, it is clear that $seqPsCAS_2(coo_{AR}, ant_0) \subseteq seqPsCAS_2(coo, ant_0)$. ■

8.6 Unary Rules

We now consider unary sr rules instead of cooperative alph-restricted sr rules. Recall that unary sr rules are sr rules of the form $a \rightarrow v$ with $v \in a^*$. Hence, one object a can move from one region to another and, while crossing the membrane, it can produce several copies of itself. It turns out that if we also allow standard antiport rules of weight one, then we get computational universality.

Theorem 8

$PsCAS_*(ncoo_U, ant_1) = PsRE$.

Proof

In order to prove the computational universality of the model, we rely on the proof in [26] which provides a construction of a universal P system Π with symport and antiport rules of weight 1, and with an environment containing an unbounded supply of objects.

We construct a CAS P system Π' that simulates the computations of Π as follows. Since symport rules of weight 1 are a particular case of unary sr rules, we can include Π as a subsystem of Π' .

In the proof from [26], during the initial steps of a computation the system receives as an input arbitrarily many objects from the environment, but only one at each step. Thus, we simulate the “infinite environment” condition by using repetitive applications of rules that are able to generate an unbounded number of objects.

More precisely, we consider a new skin membrane surrounding Π that will play the role of the environment of Π , and two additional elementary membranes in the new skin region. Let e_0 , e_1 , and e_2 be the labels for these three membranes, respectively, and let $E = \{a_1, \dots, a_n\}$ be the alphabet of the environment of Π (we refer again to [26]). Finally, we include in our system the initial multiset $w_{e_1} = a_1 \bar{a}_1 \dots a_n \bar{a}_n$, and we consider the following sets of rules:

- $R_{e_1} = \{a_i \xrightarrow{+}_{e_1} a_i^2, \bar{a}_i \xrightarrow{-}_{e_1} \bar{a}_i \mid 1 \leq i \leq n\}$.
- $R_{e_1}^a = \{(a_i, in; \bar{a}_i, out) \mid 1 \leq i \leq n\}$.
- $R_{e_2} = \{\bar{a}_i \xrightarrow{-}_{e_2} \bar{a}_i \mid 1 \leq i \leq n\}$.

These rules behave as loops that are able to generate new objects a_i for every $1 \leq i \leq n$ as long as they continue (the loop for a given i halts when the object $\bar{a}_i$ gets inside membrane e_2). Therefore, such construction can produce nondeterministically arbitrarily many copies of objects a_i for $1 \leq i \leq n$ that will be used by the original system Π to implement the simulation described in the proof from [26].

We conclude that $PsRE \subseteq PsCAS_*(ncoo_U, ant_1)$, and since $PsCAS_*(ncoo_U, ant_1) \subseteq PsRE$ is assumed to be true, the theorem holds. ■

Based on the above theorem we have the following result.

Corollary 9

$PsCAS_*(\alpha, ant_1) = PsRE$, for $\alpha \in \{ncoo_U, ncoo, coo_{AR}, coo\}$.

We conclude this section with a preliminary result concerning the generative power of systems from $CAS(ncoo_U, ant_0)$.

Theorem 10

$PsCAS_*(ncoo_U, ant_0)$ is incomparable with $PsFIN$.

Proof

On one hand it is clear that $PsCAS_*(ncoo_U, ant_0)$ contains an infinite set of vectors. Moreover $PsCAS_*(ncoo_U, ant_0)$ does not contain, for instance, the finite set $\{(2, 3), (3, 4)\}$. Every system from $CAS(ncoo_U, ant_0)$ that generates the vectors $(2, 3)$ and $(3, 4)$ can also generate the vectors $(2, 4)$ and $(3, 3)$ because the rules producing distinct symbols are noncooperative and unary, hence independent. ■

8.7 Unidirectional Membranes

In this section we consider a constraint on the form of the rules for systems in $CAS(ncoo_U, ant_1)$, which strengthens the selective role of membranes in regulating the traffic of molecules through them. More specifically, this restriction makes the traffic strictly unidirectional: if a molecule can pass through a membrane in one direction, then it cannot pass through the same membrane in the opposite direction. Such a restriction is natural from the biological point of view: e.g., if a toxic substance is secreted from a cell through its membrane, then it should not be allowed to go back. This is interesting also from mathematical point of view, because it turns out to be a real restriction: we get a model that is not computationally universal.

This restriction is formally defined as follows. Let

$$\Pi = (\Gamma, \mu, w_1, \dots, w_m, R_1, \dots, R_m, R_1^a, \dots, R_m^a, i_0)$$

be a $CAS(ncoo, ant_1)$ P system. Let, for $i \in \{1, \dots, m\}$,

$$\begin{aligned} In_i &= \{a \in \Gamma \mid (a, v, in) \in R_i, v \in \Gamma^* \text{ or } (a, in; b, out) \in R_i^a, b \in \Gamma\}, \\ Out_i &= \{a \in \Gamma \mid (a, v, out) \in R_i, v \in \Gamma^* \text{ or } (b, in; a, out) \in R_i^a, b \in \Gamma\}. \end{aligned}$$

Then Π is called *unidirectional* if In_i and Out_i are disjoint for all $i \in \{1, \dots, m\}$. The class of all unidirectional systems in $CAS(ncoo, ant_1)$ is denoted $uniCAS(ncoo, ant_1)$. Moreover, for a set of sr rules R , we define $max(R) = \max\{|v| \mid (a, v, in) \in R \text{ or } (a, v, out) \in R\}$. For a configuration C we use $tob(C)$ to denote the total number of objects (thus with multiplicities counted) present in all the regions in C . The main consequence of unidirectionality is expressed in the following lemma.

Lemma 11

Let $\Pi \in uniCAS(ncoo_U, ant_1)$, let w be the multiset of objects initially present in Π , and let R be the set of all sr rules in Π . Then for every reachable configuration C in Π , $tob(C) \leq |w| \cdot (weight(R))^m$.

Proof

The unidirectionality of Π implies that no object in Π during any computation can cross the same membrane twice. Therefore, the upper bound on the total number of membranes crossed by an object in any computation is given by m . Since in one crossing each object cannot generate more than $weight(R)$ objects and $|w|$ is the number of objects in the initial configuration, $|w| \cdot (weight(R))^m$ is indeed an

upper bound on the total number of objects in every reachable configuration in Π . ■

The next theorem shows that unidirectional P systems with unary sr rules and antiport rules of weight 1 generate only *finite* sets of vectors.

Theorem 12

$$uniPsCAS_*(ncoo_U, ant_1) = PsFIN.$$

Proof

Consider an arbitrary $\Pi \in uniCAS(ncoo_U, ant_1)$. By Lemma 11 the total number of objects in Π is bounded, and so the number of possible reachable configurations is finite. Therefore, $uniPsCAS_*(ncoo_U, ant_1)$ is also finite.

In order to prove the reverse inclusion, consider a finite set A of vectors of dimension n , and let $m = |A|$. We construct $\Pi_A \in uniCAS(ncoo_U, ant_1)$ that generates A as follows.

$$\Pi_A = (\Gamma, \mu, w_1, \dots, w_{2m+2}, R_1, \dots, R_{2m+2}, R_1^a, \dots, R_{2m+2}^a, i_0),$$

where:

- $\Gamma = \{S, a_1, \dots, a_n\}$;
- $\mu = [[[]_3]_2 \dots [[]_{2m+1}]_{2m} []_{2m+2}]_1$;
- $w_1 = S$, $w_{2m+2} = \lambda$, and $w_{2j} = \lambda$, $w_{2j+1} = a_1 \dots a_n$, for $1 \leq j \leq m$;
- The output membrane is $i_0 = 2m + 2$;
- $R_1 = \emptyset$, $R_{2m+2} = \{(a_i, a_i, in) \mid 1 \leq i \leq n\}$, and $R_1^a = R_{2m+2}^a = \emptyset$. For $1 \leq j \leq m$ we have

$$R_{2j} = \{(S, S^n, in)\} \cup \{(a_i, a_i^{x_{ji}}, out) \mid 1 \leq i \leq n, v_j = (x_{j1}, \dots, x_{jn})\},$$

$$R_{2j}^a = \emptyset,$$

$$R_{2j+1} = \emptyset,$$

$$R_{2j+1}^a = \{(S, in; a_i, out) \mid 1 \leq i \leq n\}.$$

The alphabet Γ consists of one symbol a_i for each component of the vectors, plus an additional starting symbol S . The membrane structure is set up as follows. For each vector $v_j \in A$ we have two membranes (labelled by $2j$ and $2j + 1$) in the membrane structure, and we also have another membrane to collect the output. The initial configuration contains object S in the skin, and the multiset $a_1 \dots a_n$ placed in all the regions delimited by elementary membranes (except for region $2m + 2$).

The computation in Π_A proceeds as follows. In the first step, the object S enters a nondeterministically chosen membrane labelled by $2j$, $1 \leq j \leq m$, and it produces n copies of itself. Then, objects $a_1, \dots, a_n$ present in region $2j + 1$ are exchanged via antiport rules against the n copies of S . In the next step, all those objects a_i cross membrane $2j$ towards the skin region, producing several copies

of themselves, according to the components of vector v_j . Finally, all these objects go to the output region (via sr rules of weight 1), and the computation stops. Consequently, we have $PsFIN \subseteq uniPsCAS_*(ncoo_U, ant_1)$, and so the theorem follows. ■

The unidirectional restriction has the intuitive meaning of letting each object cross a membrane only in one direction. Note that it is easy to overcome this restriction in the case when the objects can be rewritten. Indeed, we will see in the next section that the unidirectional restriction does not decrease the power of the model if the sr rules are not unary.

8.8 Noncooperative Rules

We turn now to CAS P systems using noncooperative sr rules (without antiport rules). We will prove that such systems are not universal. In this proof we will use the class of P systems with symbol rewriting noncooperative rules using targets [21], which we denote by $OP(ncoo)$. Let $PsOP(ncoo)$ be the family of sets of vectors generated by the P systems in the class $OP(ncoo)$.

Theorem 13

$$uniPsCAS_2(ncoo, ant_0) = PsCAS_*(ncoo, ant_0) = PsCF.$$

Proof

First of all we recall (from [21]) that $PsOP(ncoo) = PsCF$. Using this result we need to prove two more inclusions in order to demonstrate the theorem.

1. $PsCAS_*(ncoo, ant_0) \subseteq PsOP(ncoo)$.

This inclusion holds because any $\Pi \in CAS(ncoo, ant_0)$ can be reformulated as a P system $\Pi' \in OP(ncoo)$ with symbol-objects and noncooperative rules using targets in such a way that $Ps(\Pi') = Ps(\Pi)$. This reformulation is done as follows.

The system Π' has the same alphabet, membrane structure and output region as Π . The sr rules $a \xrightarrow{1}_i v$ from Π are reformulated as $a \rightarrow (y_1, out)(y_2, out) \dots (y_k, out) \in R_i$ in Π' where $v = y_1 \dots y_k$. Analogously, sr rules $a \xrightarrow{-1}_i v$ from Π are reformulated as rules $a \rightarrow (y_1, in_i)(y_2, in_i) \dots (y_k, in_i) \in R_j$ in Π' , where j is the surrounding region of i .

Obviously, $Ps(\Pi') = Ps(\Pi)$, and hence the inclusion holds.

2. $PsOP(ncoo) \subseteq uniPsCAS_2(ncoo, ant_0)$.

Let $\Pi = (\Gamma, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0) \in OP(ncoo)$. Let then

$$\Pi' = (\Gamma', \mu', w'_1, w'_2, R'_1, R'_2, 2) \in CAS_2(ncoo, ant_0)$$

be defined as follows

- $\Gamma' = \Gamma \times \{1, \dots, m\}$.

- $\mu' = [\begin{smallmatrix} & \\ & \end{smallmatrix}]_2]_1$.
- $w'_1 = \lambda$ and $w'_2 = h_1(w_1) \cdots h_m(w_m)$, where for $1 \leq i \leq m$, h_i is the morphism defined by $h_i(x) = (x, i)$ for $x \in \Gamma$.
- There are no sr rules associated with the skin ($R'_1 = \emptyset$).
- For each rule $r_t : a \rightarrow u \in R_i$ of Π , we include in R'_2 two sr rules:
 - $(a, i) \xrightarrow{r_t} A_t$,
 - $A_t \xrightarrow{r_t} v$,
 where v is obtained by adding (x, i) for each $(x, \text{here}) \in u$, adding (x, k) for each $(x, \text{in}_k) \in u$, and adding (x, j) for each $(x, \text{out}) \in u$ where j is the surrounding region of i .
- $i'_0 = 2$.

Note that in Π' we have collapsed the membrane hierarchy of Π : for each object x in Π a pair (x, i) is introduced which specifies the region i of object x .

The simulation of Π by Π' proceeds as follows. First, we encode into w'_2 the multisets $w_1, \dots, w_m$ present in the initial configuration of Π using the morphisms h_i , $1 \leq i \leq m$. Then, each transition step of Π is simulated in Π' in two steps: a special object is sent out to region 1 for each rule from Π that has been triggered, and then this object returns into region 2 generating the products of the application of that rule in Π , taking into account the target indicators. This process is iterated until the system stops, and the output is collected in region 2 (the inner region). Consequently, we have that $PsOP(ncoo) \subseteq uniPsCAS_2(ncoo, ant_0)$. ■

8.9 Deciding Boundness

We say that a membrane system is *bounded* if there exists a positive integer k such that in any region of any reachable configuration the cardinality of the multiset of objects present in the region does not exceed k .

It is easily seen that the boundness property is undecidable for arbitrary CAS P systems. In this section we show that it is decidable for CAS P systems using noncooperative sr rules.

In the proof of this result we will use the notion of dependency graph. This idea comes from classical formal language theory, but it has not been used in the P systems framework until recently (see [9]).

Definition 14

Let $\Pi = (\Gamma, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0)$ be a CAS P system with noncooperative sr rules. The *dependency graph* associated with it, g_Π , is defined as follows:

- Nodes: $\Gamma \times \{1, \dots, m\}$ and a special node O .

- **Edges:** First, we include an edge $O \xrightarrow{p} (a, i)$ for every $a \in \Gamma$ and $1 \leq i \leq m$ such that $|w_i|_a = p$. In this way we include in the graph information about the initial configuration.

Then, we add an edge $(a, i) \xrightarrow{p} (b, j)$ for any sr rule $a \xrightarrow{+i} u \in R_i$, where $|u|_b = p > 0$ and j is the surrounding region of i .

Analogously, we add an edge $(a, j) \xrightarrow{p} (b, i)$ for any sr rule $a \xrightarrow{-i} u \in R_i$, where $|u|_b = p > 0$ and j is the surrounding region of i .

We also add one edge from (a, j) to itself with weight 1 $(a, j) \xrightarrow{1} (a, j)$, if there is no sr rule $a \xrightarrow{-j} u$ and no sr rule $a \xrightarrow{+i} u$ for any i such that j is its surrounding region. ■

Clearly, the labels of the edges represent the multiplicity of the created objects. Since sr rules are applied in parallel, it is natural to define the weight of a path in g_Π as the multiplication of the labels of its edges.

Theorem 15

Let $\Pi \in \text{CAS}(ncoo, ant_0)$, and let $D \subseteq \Gamma_\Pi$. It is decidable whether or not there exists $k \in \mathbb{N}$ such that no region in any reachable configuration of Π contains more than k objects.

Proof

We present a constructive proof, describing an algorithmic procedure to decide if there exists such k for an arbitrary region ρ of Π . To this aim we will use the dependency graph associated with Π , denoted by g_Π .

We say that in a given dependency graph there is a “growing loop” for objects from D in region ρ if there exists a circular path in g_Π such that:

1. at least one of its edges has a label greater than 1,
2. at least one node of the form (a, ρ) with $a \in D$ is reachable from any of the nodes in the circular path,
3. at least one node of such a circular path is reachable from the special node O of g_Π .

The algorithm consists of checking if there exists a “growing loop” that is able to eventually produce an unbounded number of objects from D in region ρ . Since g_Π is finite, this can be decided.

If such a loop exists, then we conclude that the number of objects from D in ρ is *not* bounded.

Conversely, if such a loop does not exist, then there exists an upper bound on the number of objects from D in region ρ .

Thus, applying the above described algorithm for all the regions of Π we can decide the existence of a bound on the number of objects. ■

Corollary 16

It is decidable whether or not an arbitrary $\Pi \in \text{CAS}(ncoo, ant_0)$ is bounded.

Proof

The proof follows from Theorem 15, by simply setting the entire alphabet of Π as the set D . ■

In this section we provided a direct proof that the boundness property is decidable for CAS P systems with noncooperative sr rules. Note that one can also construct an indirect proof of this result using Theorem 13. Indeed, one can *construct* for each such CAS P system a context-free grammar generating the same language. Now using well known results from context-free grammars one obtains the boundness result.

8.10 Discussion

In standard membrane systems, the evolution (of objects) takes place in regions, and communication happens across membranes. In this chapter we have defined a class of membrane systems where membranes play a more central role: both communication and evolution are associated with membranes, and moreover evolution happens *only* as a result of communication.

We have presented an investigation of the *generative* power of CAS P systems. In particular, we have attempted to find out, in a systematic way, how various features/mechanisms of such systems influence their power.

First of all, we have shown that if there is some *context-sensitivity* in the system (that is, if we allow cooperative sr rules) then one obtains the computational universality, even if only cooperative alph-restricted sr rules (coo_{AR}) are used:

$$PsCAS_2(coo_{AR}, ant_0) = PsRE.$$

Also, allowing antiport rules makes the CAS P systems computationally universal, irrespectively of the type of sr rules used:

$$PsCAS_*(ncoo_U, ant_1) = PsRE.$$

Since antiports are rules where two objects residing in different sides of a membrane *cooperate* to exchange their positions, this supports the “context-sensitive” intuition of universality.

Thus “cooperation” is crucial for the control of communication in order to reach computational universality (this cooperation can be achieved by using antiport rules or by using cooperative sr rules).

We turned then to other ways of controlling communication. First, we studied the consequences of not allowing the parallel application of several sr rules on the same membrane (*seq* mode), and then we also explored the case when the communication channels are only unidirectional for each object. In both cases the computational universality is lost:

$$\begin{aligned} seqPsCAS_*(coo_{AR}, ant_0) &= PsMAT, \\ uniPsCAS_*(ncoo_U, ant_1) &= PsFIN. \end{aligned}$$

Finally, we have shown that computational universality is also lost if we explicitly required that the sr rules are noncooperative. This remains true even if we control the communication through membranes by unidirectionality:

$$\begin{aligned} PsCAS_*(ncoo, ant_0) &= PsCF, \\ uniPsCAS_2(ncoo, ant_0) &= PsCAS_*(ncoo, ant_0). \end{aligned}$$

These results are summarized in Figure 8.2. To avoid too cumbersome notation we omit here the subscripts indicating the number of membranes used. The identities involving coo_{AR} are also omitted, since they are equivalent to the coo case.

In order to get a better understanding of various features/mechanisms used in membrane systems, an even more systematic study is needed. Hence, e.g., one could consider “basic models” such as CAS P systems with unary sr rules (and without antiports), and determine their generating power.

It is somehow surprising to find out that $PsCAS(ncoo_U, ant_0)$ is incomparable with $PsFIN$.

Another interesting line of research is to study different ways of restricting the parallelism. In Section 8.7 we have studied membrane systems where *at most one* sr rule is applied on each membrane, but we allow that a membrane remains inactive even if there are applicable sr rules for it (*sequential mode*). If we impose the *maximality* condition to such sequential mode, which forces that *at least one* sr rule is applied on each membrane (provided that there are applicable rules for it), then we get some control over the application of the sr rules in the systems, and we again reach computational universality for the *cooperative* case. Thus, $max-seqPsCAS(coo, ant_0) \neq seqPsCAS(coo, ant_0)$. What is the situation in the noncooperative case?

Finally, it would be interesting to investigate the effect of minimal parallelism (see, [8]) in the framework of CAS P systems.

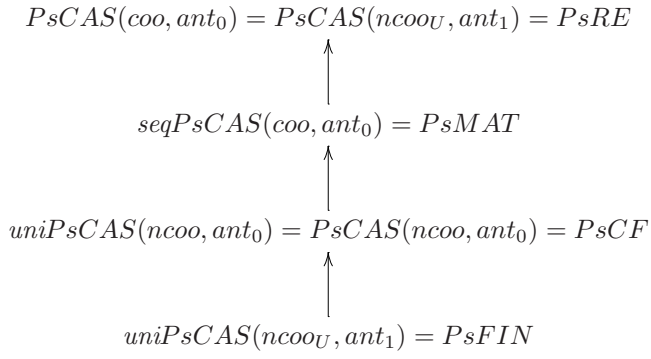


Figure 8.2: Hierarchy of families of sets of vectors computed by CAS P systems. Arrows indicate strict inclusions of the lower family in the upper family.

Bibliography

- [1] The P systems web page. <http://psystems.disco.unimib.it>.
- [2] D. Besozzi and G. Rozenberg. Formalizing spherical membrane structures and membrane proteins populations. In H.J. Hoogeboom, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Workshop on Membrane Computing*, volume 4361 of *Lecture Notes in Computer Science*, pages 18–41. Springer, 2006.
- [3] P. Bottoni, C. Martín-Vide, G. Paun, and G. Rozenberg. Membrane systems with promoters/inhibitors. *Acta Informatica*, 38(10):695–720, 2002.
- [4] N. Busi and R. Gorrieri. On the computational power of brane calculi. In C. Priami and G. Plotkin, editors, *Transactions on Computational Systems Biology VI*, volume 4220 of *Lecture Notes in Computer Science*, pages 16–43. Springer, 2006.
- [5] L. Cardelli. Brane calculi - Interactions of biological membranes. In V. Danos and V. Schachter, editors, *Computational Methods in System Biology (CSMB2004). Paris, France, May 2004. Revised Papers*, volume 3082 of *Lecture Notes in Computer Science*, pages 257–280. Springer, 2005.
- [6] L. Cardelli and Gh. Păun. An universality result for a (mem)brane calculus based on mate/drip operations. *International Journal of Foundations of Computer Science*, 17(1):49–68, 2005.
- [7] M. Cavaliere and D. Sburlan. Time-independent P systems. In Mauri et al. [18], pages 239–258.
- [8] G. Ciobanu, L. Pan, Gh. Paun, and M.J. Pérez-Jiménez. P systems with minimal parallelism. *Theor. Comput. Sci.*, 378(1):117–130, 2007.
- [9] A. Cordon-Franco, M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, and A. Riscos-Núñez. Exploring computation trees associated with P systems. In Mauri et al. [18], pages 278–286.
- [10] V. Danos and S. Pradalier. Projective brane calculus. In V. Danos and V. Schächter, editors, *CMSB*, volume 3082 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2004.

- [11] J. Dassow and Gh. Păun. *Regulated Rewriting in Formal Language Theory*. Springer, Berlin, 1984.
- [12] R. Freund, O.H. Ibarra, Gh. Păun, and H.-C. Yen. Matrix languages, register machines, vector addition systems. In M.A. Gutiérrez-Naranjo, A. Riscos-Núñez, F.J. Romero Campero, and D. Sburlan, editors, *Proceedings of the Third Brainstorming Week on Membrane Computing*, RGNC Report 01/2005, pages 155–168. Dept. of Computer Sciences and Artificial Intelligence, Univ. of Sevilla, 2005.
- [13] R. Freund, L. Kari, M. Oswald, and P. Sosík. Computationally universal P systems without priorities: Two catalysts are sufficient. *Theoretical Computer Science*, 330(2):251–266, 2005.
- [14] S. Ginsburg and G. Rozenberg. TOL schemes and control sets. *Information and Control*, 27(2):109–125, 1975.
- [15] H.J. Hoogeboom. Carriers and counters: P systems with carriers vs. (blind) counter automata. In M. Ito and M. Toyama, editors, *Developments in Language Theory*, volume 2450 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 2002.
- [16] J. Hopcroft and J. Ullmann. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [17] M. Ionescu and D. Sburlan. On P systems with promoters/inhibitors. *Journal of Universal Computer Science*, 10(5):581–599, 2004.
- [18] G. Mauri, Gheorghe Paun, M.J. Pérez-Jiménez, G. Rozenberg, and A. Salomaa, editors. *Membrane Computing, 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*, volume 3365 of *Lecture Notes in Computer Science*. Springer, 2005.
- [19] M. Minsky. *Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
- [20] Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000. Also, Turku Center for Computer Science-TUCS Report No. 208, 1998.
- [21] Gh. Păun. *Membrane Computing. An Introduction*. Springer, Berlin, 2002.
- [22] Gh. Păun and G. Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287(1):73–100, 2002.
- [23] G. Rozenberg and A. Salomaa. *The Mathematical Theory of L Systems*. Academic Press, New York, 1980.
- [24] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1–3. Springer, 1997.

-
- [25] A. Salomaa. *Formal Languages*. Academic press, New York, 1973.
- [26] G. Vaszil. On the size of P systems with minimal symport/antiport. In Mauri et al. [18], pages 404–413.

Nederlandse Samenvatting

Binnen het omvangrijke onderzoeksgebied Natural Computing kan men twee richtingen onderscheiden. Eén richting bestudeert processen in de natuur door ze op te vatten als rekenkundige bewerkingen. De andere onderzoeksrichting analyseert en ontwikkelt rekenkundige methoden en modellen geïnspireerd op de natuur. Dit proefschrift bestudeert uit beide bovengenoemde zijden van Natural Computing een representant.

Het onderwerp van Deel 1 van het proefschrift is gene assembly, een representant uit de eerste onderzoeksrichting. Gene assembly vindt plaats in ciliaten – een zeer oude groep eencellige organismen die in grote diversiteit van soorten verspreid over de aarde aanwezig is. Ciliaten hebben de opmerkelijke eigenschap dat ze twee typen celkernen hebben, de micronucleus en de macronucleus, die uitermate verschillend van elkaar zijn, zowel functioneel als fysiek. Tijdens de seksuele reproductie van ciliaten verandert een (nieuw gevormde) micronucleus in een macronucleus. Dit proces heet gene assembly. Gedurende het proces wordt ieder gen op een complexe wijze geknipt en geplakt. Het onderliggende principe van het knippen en plakken wordt recombinatie genoemd. Door middel van recombinatie wordt ieder gen in de micronucleus getransformeerd in een gen in de macronucleus.

Centraal in Deel 1 staat een graaf die, op basis van een beschrijving van een gen in de micronucleus (de beginsituatie), het resultaat na recombinatie weergeeft. Dit eindresultaat bevat naast een lineaire DNA-structuur mogelijk diverse cirkelvormige DNA-structuren. Deze structuren zijn in de graaf eenvoudig terug te vinden. Het corresponderende gen in de macronucleus bevindt zich op een van deze structuren en wordt daardoor ook door de graaf gerepresenteerd. In Hoofdstuk 3 blijkt bovendien dat door middel van de graaf de mogelijke volgorde waarin de cirkelvormige DNA-structuren ontstaan gedurende gene assembly gekarakteriseerd kunnen worden. Dit is opmerkelijk aangezien de graaf enkel op basis van de beginsituatie geconstrueerd wordt. In Hoofdstuk 4 worden de ontstane lineaire en cirkelvormige DNA-structuren als uitgangspunt genomen, en worden de mogelijke beginsituaties beschreven die deze DNA-structuren opleveren. Het blijkt dat meerdere beginsituaties hetzelfde eindresultaat kunnen hebben. Tevens blijkt dat de beginsituaties met hetzelfde eindresultaat in elkaar over te voeren zijn door middel van string-herschrijfgeregels. Opvallend daarbij is dat deze her-

schrijfgeregels veel gelijkenis vertonen met een bekend model van gene assembly dat drie typen van recombinatie onderscheidt. Ook worden in dit hoofdstuk de mogelijke grafen die eindsituaties representeren beschreven in termen van eenvoudig te verifiëren condities. In Deel 1 dient gene assembly als motivatie; er wordt gewerkt binnen een abstracte setting gebaseerd op strings. Uitzondering hierop is Hoofdstuk 5, waarin de bovengenoemde graaf op equivalente wijze gedefinieerd is in termen van grafen in plaats van strings.

Het onderwerp van Deel 2 van het proefschrift is membrane computing, een representant uit de tweede onderzoeksrichting. Binnen deze onderzoeksrichting worden systemen geïnspireerd op de werking van membranen binnen een cel (of tussen cellen) bestudeerd. Membranen verdelen een cel in diverse compartimenten en laten bepaalde moleculen of ionen alleen op een gecontroleerde wijze van een compartiment naar de andere door. Deze moleculen (of ionen) zijn daardoor in een juiste hoeveelheid aanwezig in de compartimenten. Binnen membrane computing wordt de werking van membranen opgevat als een rekenkundig proces. De membranen vormen hier een hiërarchische structuur die de cel in diverse compartimenten verdeelt. Ieder van deze compartimenten bevat een aantal objecten, en deze kunnen door middel van voorgeschreven regels getransformeerd worden in andere objecten en/of verplaatst worden naar andere compartimenten. Belangrijke eigenschap hierbij is dat de regels op een maximaal parallelle wijze worden uitgevoerd. Hieronder wordt verstaan dat de verzameling regels (preciezer: regelinstanties) die worden gebruikt in een tijdstap maximaal is – er is geen regel meer die (een gedeelte van) de ongebruikte objecten had kunnen transformeren. Deze maximale parallelle wijze van regeluitvoering biedt het membraansysteem soms onverwachte rekenkracht.

Men kan diverse klassen van membraansystemen onderscheiden. In Hoofdstuk 6 worden membraansystemen bekeken waarbij objecten niet alleen in de compartimenten kunnen voorkomen, maar ook op de membranen. De objecten beïnvloeden op deze wijze het functioneren van de membranen. Een membraan kan bijvoorbeeld de verplaatsing van bepaalde objecten alleen toestaan als er op het membraan een specifiek object bevindt. Ook kan een membraan al dan niet gesplitst worden afhankelijk van de aanwezigheid van objecten op het membraan. Vervolgens beschrijft Hoofdstuk 7 membraansystemen waarbij de evolutie van het systeem afhangt van signalen van buiten. Het signaal bestaat hierbij uit een reeks objecten die één voor één actief worden en daarmee de activatie van de (evolutie)regels beïnvloedt. Het signaal verplaatst zich door het systeem totdat alle objecten van het signaal verbruikt zijn. Tenslotte bestudeert Hoofdstuk 8 een specifieke klasse van symport/antiport membraansystemen. Essentieel bij (standaard) symport/antiport systemen is dat iedere evolutieregel precies twee voorgeschreven objecten tegelijkertijd door een membraan heen laat gaan. In dit hoofdstuk staan we binnen een dergelijk membraansysteem toe dat de twee objecten tijdens het verplaatsen tevens kunnen veranderen. Motivatie hiervoor is dat moleculen gedurende de doorgang van een membraan betrokken kunnen zijn bij chemische reacties, waardoor de moleculen op cruciale wijze kunnen veranderen.

Het centrale onderzoeksonderwerp van Deel 2 is het bepalen van de rekenkracht van de bovengenoemde klassen van membraansystemen (en van natuurlijke restricties op deze klassen) door ze zowel onderling als met andere bekende en uitvoerig bestudeerde klassen te vergelijken. Daaruit blijkt dat de rekenmogelijkheden van de verschillende membraansystemen aanzienlijk kunnen verschillen. Bijvoorbeeld blijkt uit Hoofdstuk 6 dat er geen algoritme bestaat die bepaalt of gedurende de evolutie van het systeem er een membraan is die een gegeven verzameling objecten bevat. Echter, als we de splitsing van membranen niet toestaan, dan bestaat een dergelijk algoritme wel.

Curriculum Vitae

Robert Brijder is geboren in Delft op 5 november 1980. Van 1993 tot 1999 doorliep hij het atheneum (VWO) op het Zandeveld College (ISW) te 's-Gravenzande. In september 1999 begon hij de studie Wiskunde aan de Universiteit Leiden. Na het afleggen van het propedeutisch examen Wiskunde, begon hij in september 2000 aan de studie Informatica aan diezelfde universiteit. In 2001 legde hij het dubbel propedeutisch examen Wiskunde-Informatica af, en in 2003 deed hij een half-jarige stage bij Philips Medical Systems. In april 2004 slaagde hij cum laude voor het doctoraalexamen Informatica met afstudeerrichting Theoretische Informatica. Vanaf september 2004 was hij verbonden aan het Leiden Institute of Advanced Computer Science waar hij zijn promotieonderzoek uitvoerde in het kader van NWO project “VIEWS” en betrokken was bij onderwijs.

Publication List

Patents

- [1] *High angular resolution diffusion weighted MRI*.
Inventors: F.G.C. Hoogenraad, R.F.J. Holthuisen, and R. Brijder.
International patent number WO2005076030,
also EP1714164, and CN1918481 (2005).

Journal papers

- [1] F. Bernardini, R. Brijder, G. Rozenberg, and C. Zandron. Multiset-based self-assembly of graphs. *Fundamenta Informaticae*, 75(1-4):49–75, 2007.
- [2] R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, and D. Sburlan. Communication membrane systems with active symports. *Journal of Automata, Languages and Combinatorics*, 11(3):241–261, 2006.
- [3] R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, and D. Sburlan. Membrane systems with proteins embedded in membranes. *Theoretical Computer Science*, 404:26–39, 2008.
- [4] R. Brijder and H.J. Hoogeboom. Combining overlap and containment for gene assembly in ciliates. Submitted, 2008.
- [5] R. Brijder and H.J. Hoogeboom. The fibers and range of reduction graphs in ciliates. *Acta Informatica*, 45:383–402, 2008.
- [6] R. Brijder and H.J. Hoogeboom. Perfectly quilted rectangular snake tilings. To appear in *Theoretical Computer Science*, 2008.
- [7] R. Brijder, H.J. Hoogeboom, and M. Muskulus. Strategies of loop recombination in ciliates. *Discrete Applied Mathematics*, 156:1736–1753, 2008.
- [8] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. Reducibility of gene patterns in ciliates using the breakpoint graph. *Theoretical Computer Science*, 356:26–45, 2006.

- [9] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. How overlap determines the macronuclear genes in ciliates. Submitted, also LIACS Technical Report 2007-02, [arXiv:cs.LO/0702171], 2008.
- [10] R. Brijder, M. Langille, and I. Petre. Extended strings and graphs for simple gene assembly. To appear in *Theoretical Computer Science*, 2008.
- [11] M. Muskulus, D. Besozzi, R. Brijder, P. Cazzaniga, S. Houweling, D. Pescini, and G. Rozenberg. Cycles and communicating classes in membrane systems and molecular dynamics. *Theoretical Computer Science*, 372(2-3):242–266, 2007.
- [12] M. Muskulus and R. Brijder. Complexity of bio-computation: symbolic dynamics in membrane systems. *International Journal of Foundations of Computer Science*, 17(1):147–165, 2006.

Peer-reviewed conference and workshop papers

- [1] R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, and D. Sburlan. Membrane systems with external control. In H.J. Hoogeboom, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Workshop on Membrane Computing*, volume 4361 of *Lecture Notes in Computer Science*, pages 215–232. Springer, 2006.
- [2] R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, and D. Sburlan. Membrane systems with marked membranes. *Electronic Notes Theoretical Computer Science*, 171(2):25–36, 2007.
- [3] R. Brijder and H.J. Hoogeboom. Applicability of loop recombination in ciliates using the breakpoint graph. In A. Siebes et al., editors, *CompLife '07*, volume 940 of *AIP Conference Proceedings*, pages 50–59, 2007.
- [4] R. Brijder and H.J. Hoogeboom. Characterizing reduction graphs for gene assembly in ciliates. In T. Harju, J. Karhumäki, and A. Lepistö, editors, *Developments in Language Theory (DLT) 2007*, volume 4588 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2007.
- [5] R. Brijder and H.J. Hoogeboom. Perfectly quilted rectangular snake tilings. In J. Kari et al., editors, *Proceedings of the Satellite Workshops of DLT 2007, Part 3: Workshop on Tilings and Self-Assembly*, volume 45 of *TUCS General Publication*, 2007.
- [6] R. Brijder and H.J. Hoogeboom. Extending the overlap graph for gene assembly in ciliates. In C. Martín-Vide, F. Otto, and H. Fernau, editors, *LATA 2008*, volume 5196 of *Lecture Notes in Computer Science*, pages 137–148. Springer, 2008.

- [7] R. Brijder, H.J. Hoogeboom, and M. Muskulus. Applicability of loop recombination in ciliates using the breakpoint graph. In M.R. Berthold et al., editors, *CompLife '06*, volume 4216 of *Lecture Notes in Computer Science*, pages 97–106. Springer, 2006.
- [8] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. The breakpoint graph in ciliates. In M.R. Berthold et al., editors, *CompLife '05*, volume 3695 of *Lecture Notes in Computer Science*, pages 128–139. Springer, 2005.
- [9] R. Brijder, H.J. Hoogeboom, and G. Rozenberg. From micro to macro: How the overlap graph determines the reduction graph in ciliates. In E. Csuhaj-Varjú and Z. Ésik, editors, *Fundamentals of Computation Theory (FCT) 2007*, volume 4639 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 2007.
- [10] R. Brijder, M. Langille, and I. Petre. A string-based model for simple gene assembly. In E. Csuhaj-Varjú and Z. Ésik, editors, *Fundamentals of Computation Theory (FCT) 2007*, volume 4639 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2007.
- [11] M. Muskulus and R. Brijder. First steps towards a geometry of computation. In M.A. Gutiérrez Naranjo et al., editors, *Proceedings of the Third Brainstorming Week on Membrane Computing*, pages 197–218. Fénix Editora, 2005.
- [12] M. Muskulus, S. Houweling, G. Rozenberg, D. Besozzi, P. Cazzaniga, D. Pescini, and R. Brijder. Reaction cycles in membrane systems and molecular dynamics. In C. Graciani Diaz et al., editors, *Proceedings of the Fourth Brainstorming Week on Membrane Computing*, volume 2, pages 185–208. Fénix Editora, 2006.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL .* Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing.* Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components.* Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method.* Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löb. *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenbergh. *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Soft-*

ware Architectures. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling.* Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertionial Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

- G. Lenzini.** *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07
- I. Kurtev.** *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08
- T. Wolle.** *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09
- O. Tveretina.** *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10
- A.M.L. Liekens.** *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11
- J. Eggermont.** *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12
- B.J. Heeren.** *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13
- G.F. Frehse.** *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14
- M.R. Mousavi.** *Structuring Structural Operational Semantics.* Faculty of Mathematics and Computer Science, TU/e. 2005-15
- A. Sokolova.** *Coalgebraic Analysis of Probabilistic Systems.* Faculty of Mathematics and Computer Science, TU/e. 2005-16
- T. Gelsema.** *Effective Models for the Structure of π -Calculus Processes with Replication.* Faculty of Mathematics and Natural Sciences, UL. 2005-17
- P. Zoetewij.** *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18
- J.J. Vinju.** *Analysis and Transformation of Source Code by Parsing and Rewriting.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19
- M. Valero Espada.** *Modal Abstraction and Replication of Processes with Data.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20
- A. Dijkstra.** *Stepping through Haskell.* Faculty of Science, UU. 2005-21
- Y.W. Law.** *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22
- E. Dolstra.** *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01
- R.J. Corin.** *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02
- P.R.A. Verbaan.** *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03
- K.L. Man and R.R.H. Schiffelers.** *Formal Specification and Analysis of*

Hybrid Systems. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for stream-*

ing DSP applications. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development*

Processes. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting*

Controversy. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automation Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of

Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23