Uniformity conditions for membrane systems

Uncovering complexity below P

Niall Murphy

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science,
Faculty of Science and Engineering,
National University of Ireland Maynooth
Maynooth, Ireland

Supervisors: Dr. Damien Woods and Thomas J. Naughton
Head of Department: Dr. Adam C. Winstanley

May, 2010

A thesis submitted for the degree of Doctor of Philosophy

To my parents.

Acknowledgments

First of all I thank my parents for their patience, support, and encouragement over the past four years. Extra special thanks to Satoko Yoshimura (吉村聡子) for the love, care, happiness she has given me.

My supervisors Damien Woods and Tom Naughton also deserve special mention. This document would have been impossible if not for Damien's super-human efforts to beat some *smacht* into me. He tried his best to make me into a careful researcher and I hope some of his attitude and dedication have rubbed off on me. His enthusiasm and optimism always helped reignite my research spark when I was plagued with doubts or entered the doldrums. While I deviated away from Tom's area of expertise very early on in my research, he was always ready to help (spectacularly) whenever I needed him.

I would also like to thank Irish Research Council for Science, Engineering and Technology funding me.

In Seville I would like to thank Mario de Jesús Pérez-Jiménez for allowing me to squat in the offices of the Research Group on Natural Computing in the University of Seville and Agustín Riscos-Núñez for checking that my overly Byzantine definitions were equivalent to what other researchers had in mind. I would also like to thank the members of the group for making me feel welcome in Spain, especially Miguel* Martínez-del Amor and Ignacio Pérez-Hurtado.

In Maynooth I would like to thank the staff and students of the Computer Science Department for some fun years and interesting conversations, especially Lukas Ahrenberg, Stuart Butler, Karen Molony, Andrew Page, Des Traynor, and Turlough Neary who can make me laugh so much I vomit.

I would like to thank Shanie Weissman (שני ויסמן) who taught me a new way of thinking and who encouraged me to start this project in the first place.

Also thanks to all my friends who helped me relax, special mentions for Cathal Browne, Hai-Ying Chen (陳海穎), Xi Chen (陳曦), Niall Kelly, Adam Lyons, Carlos Montaño-Tanco, Lei Pan (潘蕾), Caoimhin Scully, Melanie Stuertz, Shekman Tang (鄧碩文), and Fergal Walsh.

Abstract

We characterise the computational complexity of biological systems to assess their utility as novel models of computation and learn how efficiently we can simulate these systems in software. In this work we focus on the complexity of biological cells by using several established models of cell behaviour collectively known as Membrane Systems or P-Systems.

Specifically we focus on analysing the power of cell division and membrane dissolution using the well established active membrane model. Inspired by circuit complexity, researchers consider uniform and semi-uniform families of recogniser membrane systems to solve problems. That is, having an algorithm that generates a specific membrane system to compute the solutions to specific instances of a problem.

While the idea of uniformity for active membrane systems is not new, we pioneer uniformity conditions that are contained in P. Previously, all attempts to characterise the power of families of membrane systems used polynomial time uniformity. This is a very strong uniformity condition, sometimes too strong. We prove three major results using tighter uniformity conditions for families of recogniser active membrane systems.

First, by tightening the uniformity condition slightly to log space (L) we improve a P upper-bound on a semi-uniform family of membrane systems to a NL characterisation. With new insight into the nature of semi-uniformity we explore the relation between membrane systems and problems complete for certain classes. For example, the problem STCON is NL-complete; by restricting the problem slightly it becomes L-complete. This restriction in turn suggests a restriction to the NL characterising model which produces a new, L characterising, variation of recogniser membrane systems.

The second and most significant of our results answers an open question in membrane computing: whether the power of uniform families and semi-uniform families are always equal. The answer to this question has implications beyond membrane computing, to other branches of natural computing and to circuit complexity theory. We discovered that for AC^0 uniformity, the problems solved by uniform families of systems without dissolution rules or charges are a strict subset of those problems solved by AC^0 semi-uniform families of the same type.

Finally we present a result contributing to another open question known as the P-conjecture. We provide a surprising P characterisation of a model that can generate exponential space in linear time using cell division. The algorithms that we use to compress this exponential information are of interest to those wishing to simulate cell behaviour or implement these models *in silico*.

Tighter uniformity conditions allow researchers to study a range of complexity classes contained in P using the language of membrane systems. We argue that our stricter definition of uniformity is a more accurate one for characterising recogniser membrane systems because it allows researchers to see more clearly the actual power of systems, while at the same time all pre-existing results for classes that contain P (e.g. PSPACE, NP) still hold.

Contents

Chapter 1

Introduction

Living cells do an amazing amount of information processing every second. They read, splice, and recombine DNA, generate polypeptides and fold them into beautiful and complex proteins using a fraction of the time and energy that silicon based computers use to simulate the same tasks. Harnessing this computing power would be revolutionary in the history of computing. We contribute to this aim by studying the theoretical computing power of membrane computers.

There is a point of view, pioneered by Fredkin in the 1970's and 1980's, that computation is inherent in Nature, that every physical action can be interpreted as preforming some computation. For example, a stone falling to the earth for t seconds calculates t times 9.81, or electrons moving through silicon can be interpreted as computing a Boolean function.

The materials used to construct a computer are chosen because their intrinsic properties force them to behave in a certain ways due to the laws of nature. Until the 1930's the majority of computers were based on rotating wheels and cogs of different diameters. The ratios of the component's diameters allows this technology to compute large sums and solve differential equations. It was most famously pushed to its limits by Charles Babbage and his planned "Analytical Engine". These mechanical computers were programed by punch cards, patterns of holes punched in pieces of card.

Around the end of the 19^{th} century, the insulating properties of cardboard were discovered to break an electric circuit, the punch cards of mechanical computers were repurposed to control electrical circuits in the next generation of computer. These electric computers were further enhanced in the 1940's and 1950's by using the electrons emitted by a hot metal in a vacuum to control electric currents, these were the vacuum tube computers.

In the 1960's it became more popular to use the behaviour of electrons in silicon to compute Boolean functions, these circuits in wafers of crystal silicon are called integrated circuits. Integrated circuits have been rampantly successful ever since due to their low energy usage and because they allow for the construction of significantly smaller computing devices.

However, the search for properties of nature to use as new and powerful methods of

computing continues. The fields of quantum computing, chemical computing, optical computing, and biological computing all seek to improve the speed, accuracy, and frontiers of our computing ability. These fields also contribute to our knowledge about the fundamental nature of the universe by testing the "Church–Turing–Deutsch" principle [20]. That is, can all of physics be simulated by a finite machine and whether our current abstract computing models are the most powerful possible given the fundamental laws of physics.

Biology has been a rich source of ideas for future computing technology. Several different directions have emerged in the field which is collectively known as biological computing.

- Studying the way biological systems solve problems and taking inspiration to design new algorithms (e.g. Neural Networks [43], Genetic algorithms, (Ant) Swarm Intelligence).
- Building physical computers out of biological components, (e.g. Adleman [1], Winfree [73], Beneson [11], Fromherz [23]).
- Directly simulating biological systems in software and hardware (e.g. the Blue Brain project [41]).

Membrane computing defines computation models that are inspired by and abstracted from the structure and function of living cells. The first formal computational models of membrane systems* were introduced by George Păun [50]. This area has since expended greatly and straddles all areas of biological computing, however, the implementation [25] of such systems still seems far away. In anticipation of future implementations, the majority of the research done in membrane computing to date has been analysing the computing power of the systems using the tools of computational complexity theory.

The formal study of computation began when Turing [71] and Church [14] crystallised the idea of a computation in the Turing machine and λ-calculus respectively and showed a limit to what is computable. A Turing machine with the ability to simulate any other Turing machine is known as a universal Turing machine. Contemporaneously with the rise of integrated circuit based computers, researchers focused on examining what problems computers could solve with restrictions on the time and memory available to them. The class of problems solvable in polynomial time, known as P, was defined in the early sixties [15, 22, 57] and the class of problems solved by non-deterministic machines in polynomial time (NP) in the early seventies [17, 37, 40]. It remains to be shown if $\mathsf{P} \neq \mathsf{NP}$, this has become one of the most famous problems in computer science and has important consequences outside of computer science to mathematics, and even philosophy.

We study the theoretical limits of integrated circuit based computers using the Boolean circuit model. This model closely resembles the finite hardware of a computer, both silicon and Boolean circuits have a fixed maximum input length. Borodin [12]

* "P-system" is the more usual term for these devices, however, to avoid confusion with the complexity class P we refer to them as membrane systems.

and Cook [18] introduced infinite uniform families to allow Boolean circuits to solve all instances of a problem and not just those instances of a certain length. In a uniform family there is an algorithm that constructs each member of the family for a specific length. This algorithm is conceptually similar to a machine in a factory building physical circuits.

Circuits have provided complexity theory with some of its deepest results, for example PARITY cannot be solved by uniform constant depth polynomial size circuits [24]. Also many class separations have been proved using monotone circuits, most significantly monotone P and monotone NP [58]. However, it is thought unlikely to be possible to extend this proof to general circuits [59, 60] and thus the $\mathsf{P} \neq \mathsf{NP}$ conjecture.

1.1 Motivations and contributions

The native ability of bacteria and other cells to make copies of themselves inspires us with visions of exponentially multiplying processors all churning away, solving intractable problems. But can a cell actually be used to solve such hard problems? To answer this we abstract away from the mind-boggling intricacy of a living cell and create a model that strips away everything but the features that we want to analyse. Mitosis (cell division) and apoptosis (cell dissolution) seem to have great potential for computation, accordingly there is a variant of membrane systems known as *active membranes* [51] designed to study the computational complexity of these processes. An instance of the model consists of a number of (possibly nested) membranes, or compartments, which themselves contain objects which represent chemicals or molecules. The objects become other objects by rules (representing chemical reactions) which are applicable depending on the compartment they are in. Objects can also pass through membranes, trigger division, or dissolve a membrane.

We say that a family of active membrane systems solves a problem if there is a member of the family to recognise every instance of the problem. In a semi-uniform family each member of the family recognises a single problem instance. In a uniform family each member of the family recognises all problem instance a certain length.

It was shown that semi-uniform (Sosík [66]) and uniform families (Alhazov et al. [4]) of active membrane systems could solve PSPACE-complete problems in linear time, which indicates that active membrane systems are a powerful parallel model of computation [26]. Later, Sosík and Rodríguez-Patón showed [67] that active membranes are no more powerful than parallel machines by proving a PSPACE upper-bound on the model. This result holds for a very general definition of membrane systems and so can be interpreted as an upper-bound on systems with dividing membranes.

The original active membrane model allows for membranes to be "charged" (representing an electrical charge or polarization). This in effect allows a membrane to relabel itself and seems to be too powerful to allow a system to characterise classes contained in PSPACE. Alhazov et al. proposed active membrane systems without membrane charges [6] which is denoted $\mathcal{AM}^0$. Using active membranes without charges it seems that solving PSPACE-complete problems necessitated the use of non-elementary

division [7], that is, a copy of all sub-membranes is made when a membrane divides. This idea was formulated as the "P-conjecture" which states that active membrane systems without charges and non-elementary division characterise P [53]. There have been many attempts to resolve this problem (see Mauri et al. [42] for a survey) and we present a solution for a restriction of the problem in Chapter 5. Another attempt to resolve the P-conjecture made the discovery that membrane dissolution is a key ingredient to active membrane systems and that all systems without dissolution rules are upper-bounded by P [30]. However in Chapters 3 and 4 of this thesis we show that the situation is even more subtle and interesting.

Previously (almost) all complexity results in membrane computing used uniform and semi-uniform families whose members were constructable in polynomial time. We argue that polynomial time uniformity is to strong to discuss the complexity class P and below and that in general if is best to use a weak uniformity condition to get a clear understanding of the computational complexity of a membrane system. All results in this thesis (apart from Section 5.2) are clarifications of the true computational complexity of various active membrane systems that are only possible due to us choosing more appropriate uniformity condition. Figure 1.1 is a representation of the best known results related to the P-conjecture excluding the work in this thesis. The effects of restricting the semi-uniformity and uniformity conditions of P-conjecture systems to be computable FAC^0 can be seen in Figure 1.2.

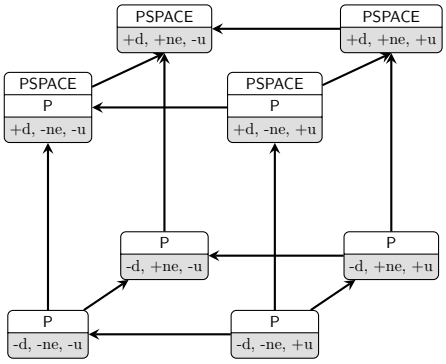

Figure 1.1: A diagram showing the best known upper and lower-bounds on families of some variations of active membrane systems without charges (excluding the work in this thesis). Arrows represent inclusions. The lower part of each node indicates the properties that system: the parameter "-d", rules of type (d) are prohibited, "-ne" that rules of types (e_w) and (f) are prohibited. The parameter "+u" indicates P-uniform and "-u" P-semi-uniform families. If the node is split, the top part represents the best known upper-bounds and the lower part, the best known lower-bounds. A node with a single complexity class represents a characterisation.

By studying the computational complexity of membrane systems we express existing problems in a new context. For example, thanks to the results in Chapters 3 and Chapters 5 we can describe the difference between NL and P in terms of dissolution and evolution in membrane systems.There is also the possibility that studying complexity

theory in the framework of membrane systems will yield solutions to general complexity problems more easily than in other frameworks.

Complexity results for membrane computing allow those implementing membrane computers to know in advance what their experimental devices are capable of computing and how efficiently. Similarly these results also guide those who seek to simulate in software, models of cells or use membrane systems to simulate other complex systems such as eco-systems or protein channels, since a simulator cannot compute any faster or with less memory than the theoretical limits imposed by the lower-bound of a good model.

Now we list the contributions that we have made in this thesis.

1. *Tighter uniformity conditions for families of membrane systems.* Previously, only polynomial time constructable families of membrane systems were considered. By introducing families constructable in constant parallel-time (FAC^0) we discovered that several upper-bounds for various models could be improved and allowed the characterisation of classes inside P using membrane systems.

2. *Improved the* P *upper-bound on semi-uniform* $\mathcal{AM}^0_{-d}$ *to* NL. Previously (with P uniformity) the upper-bound on all active membrane systems without dissolution was P [30]. Using more appropriate semi-uniformity conditions we showed that these systems actually characterise NL.

3. *Discovered that the* C*-uniform* $\mathcal{AM}^0_{-d}$ *characterises* C *for a range of complexity classes* C. P-uniform families of $\mathcal{AM}^0_{-d}$ characterise P [30], however we discovered that if the uniformity condition is computable in some other class C, the resulting family characterises the class C! We have observed this effect down as far as $\mathsf{C} = \mathsf{AC}^0$.

4. *Proved that uniform and semi-uniform families are not equal.* A trend of results seemed to indicate that semi-uniform and uniform families of systems would always have the same computing power. However, we have shown that this is not the case, resolving Open question C in [56].

5. *Introduced semi-uniformity for circuits.* Our result showing uniform and semi-uniform families have different computing power is applicable for all models which use uniform families. To demonstrate this we proved a similar result for uniform and semi-uniform families of circuits.

6. *Resolved a biologically motivated restriction of the P-conjecture.* We show a partial result for the still-open P-conjecture. We restricted the division rules so that both resulting membranes must have the same contents and showed that such systems characterises P.

7. *Introduced a new technique for constructing semi-uniform families.* We observed that a semi-uniform family of model M to solve problem X is closely related to a reduction from X to the prediction problem for the model M.

8. *Introduced a more general recogniser condition for membrane systems.* The standard restrictions on recogniser membrane systems can be generalised in a way that makes them easier to program yet does not increase their lower-bound.

9. *Introduced a new way to restrict the computing power of a system.* By restricting the definition of recogniser systems, we restricted the power of semi-uniform $\mathcal{AM}^0_{-d}$ systems to L.

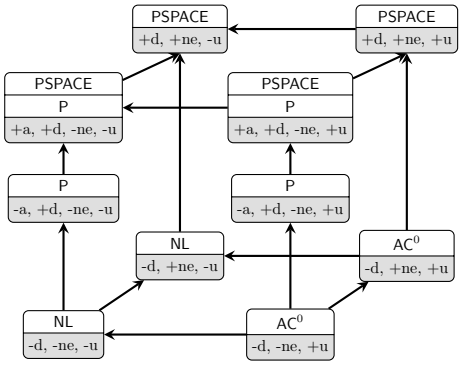

Figure 1.2: A diagram, including the results in this thesis, showing the currently known upper and lower-bounds on the variations of active membrane systems without charges. The lower part of each node indicates the properties that system: the parameter "-a", rules of type "(e)" are prohibited, "-d", rules of type (d) are prohibited, "-ne" that rules of types (e_w) and (f) are prohibited. The parameter "+u" indicates AC^0-uniform and "-u" AC^0-semi-uniform families. If the node is split, the top part represents the best known upper-bounds and the lower part, the best known lower-bounds. A node with a single complexity class represents a characterisation.

1.2 Basic terminology

First we clarify some notation then define graphs, multisets, complexity classes and reductions.

- Let $\mathbb{N}_0$ be the set of non-negative integers $\{0, 1, 2, \ldots\}$.
- The set of all subsets of S, known as the powerset of S, is denoted $\mathcal{P}(S)$.
- All logarithms are to base 2, unless specified otherwise.

1.2.1 Graphs

A finite graph $G = (V, E)$ is a pair of vertices V and edges $E \subseteq V \times V$. (We do not allow multi-edges). If the edges are of the form $E \subseteq \{(p, c) \mid c, p \in V\}$ we say the graph is directed. If the edges are of the form $E \subseteq \{\{p, c\} \mid c, p \in V\}$ we say the graph is undirected. In an edge (p, c) we call p the parent and c the child node.

Let $G = (V, E)$ be a directed graph with $x, y, z \in V$. Then let path(x, y) be true if $x = y$, or $\exists\ z$ s.t path(x, z) and path(z, y). Otherwise path(x, y) is false. A cycle is where a node x in the graph has a path via 1 or more edges from node x back to x again. A graph without cycles is said to be *acyclic*.

A tree, T, is an acyclic graph where each vertex has at most 1 parent and there is a single special parentless vertex (called the root) from which there is a unique path to each other vertex in T. The function subtree : $T \times V \to T$ returns the subtree of the tree T rooted at v. If v is a leaf then v is returned, if $v \notin V$ then the empty set is returned.

Let parent : $T \times V \to V$ such that when given a vertex, c, in a directed tree T, the parent of that vertex (or null) is returned, that is

$$\text{parent}(T, c) = p \text{ such that, } \exists (p, c) \in E$$

In this thesis when a graph is represented in binary, the set of edges is encoded as an adjacency matrix. The *adjacency matrix* A with entries $(a_{i,j})_{n \times n}$ of a graph G (with $n = |V|$ vertices) is defined by

$$a_{i,j} := \begin{cases} 1 \text{ if } (v_i, v_j) \in E \\ 0 \text{ otherwise.} \end{cases}$$

Let a depth first ordering (DFO) be a list of the vertices in a tree such that no vertex is listed before its parent. That is there is an ordering $\leq_{\text{DF}}$ on the vertices of a tree such that $v_i \leq_{DF} v_j$ (where $v_i, v_j \in V$) iff v_i is an element of the vertices of subtree(T, v_j). This ordering is computed by the well known Depth First search algorithm in time $\mathcal{O}(|V| + |E|)$ [70].

1.2.2 Multisets

A *multiset* [38, 68] over a finite alphabet Σ is a mapping $Q : \Sigma \to \mathbb{N}_0$. The multiplicity of s in the multiset $\mathcal{M}$ is $Q_{\mathcal{M}}(s)$ (where $s \in \Sigma$), this is expressed as a tuple $(s, Q(s)) \in \mathcal{M}$. The set of all multisets over Σ is written as MS(Σ). If A and B are multisets, the binary operator $\uplus$ returns a new multiset such that an element occurring a times in A and b times in B occurs exactly $a + b$ times in $A \uplus B$. The binary operator $\ominus$ returns a new multiset such that an element occurring a times in A and b times in B occurs exactly $\max(0, a - b)$ times in $A \ominus B$. The empty multiset is denoted $\emptyset$.

The support of multiset $\mathcal{M}$ is the set support$(\mathcal{M}) = \{x \in \Sigma \mid Q_{\mathcal{M}}(x) > 0\}$. We say that $x \in \mathcal{M}$ if $x \in$ support$(\mathcal{M})$.

A multiset $\mathcal{M}$ can be represented as an explicit collection of elements containing $Q_{\mathcal{M}}(x)$ occurrences of x. For example, the multiset $\mathcal{M}$ over $\{a, b, c\}^*$ with $Q_{\mathcal{M}}(a) = 3, Q_{\mathcal{M}}(b) = 1, Q_{\mathcal{M}}(c) = 2$ is represented as $\widetilde{\mathcal{M}} = [a, b, a, c, c, a]$. As with sets, the order of the elements in this representation is not fixed. When we especially wish for the multiset to be expressed in this format we refer to it using the notation $\widetilde{\mathcal{M}}$. Note that to distinguish multisets from sets we use brackets [and] in the place of { and }. Let $|\mathcal{M}|$ be the total number of objects in a multiset. Let $\|\mathcal{M}\|$ be the number of distinct objects in the multiset, that is $\|\mathcal{M}\| = |\text{support}(\mathcal{M})|$.

1.2.3 Computational complexity

The following summary of complexity classes is based on those by Kabanets [36]. In this thesis, by *Turing machine* we mean a standard multitape Turing machine that consists of a finite control, three semi-infinite tapes (input tape, work tape, and output tape) divided into cells, and a tape head for each of the three tapes.

A Turing machine is *deterministic* if at each time-step it has at most one choice for its next move. We say that a Turing machine M *decides* a language $L \subseteq \Sigma^*$ if on every input string $x \in \Sigma^*$ where $x \in L$, M halts in the accepting state, and if $x \notin L$ the M halts in the rejecting state.

We define $\mathsf{TIME}(f)$ to be the class of languages that can be decided by a deterministic Turing machine which is *time* bounded by some function in $\mathcal{O}(f)$. We define $\mathsf{SPACE}(f)$ to be the class of languages that can be decided by a deterministic Turing machine which is *space* (accessed cells of work tape) bounded by some function in $\mathcal{O}(f)$.

A Turing machine is *non-deterministic* if it can have more that one choice for its next move. A language $L \subseteq \Sigma^*$ is *decided* by a non-deterministic Turing machine M if, for every input string $x \in \Sigma^*$, if $x \in L$, there is at least one legal computation path that takes M to the accepting state, else if $x \notin L$, no computation path will take M to the accepting state.

We define $\mathsf{NTIME}(f)$ to be the class of languages that can be decided by a non-deterministic Turing machine which is *time* bounded by some function in $\mathcal{O}(f)$. We define $\mathsf{NSPACE}(f)$ to be the class of languages that can be decided by a non-deterministic Turing machine which is *space* bounded by some function in $\mathcal{O}(f)$.

We now define the classes of problems

$$\mathsf{P} = \cup_{k>0}\mathsf{TIME}(n^k)$$

$$\mathsf{NP} = \cup_{k>0}\mathsf{NTIME}(n^k)$$

$$\mathsf{PSPACE} = \cup_{k>0}\mathsf{SPACE}(n^k)$$

$$\mathsf{L} = \mathsf{SPACE}(\log n)$$

$$\mathsf{NL} = \mathsf{NSPACE}(\log n)$$

We also note that a language $\overline{L} = \Sigma^* \backslash L$ is in the complexity class coC iff its complement language L is in the complexity class C. Immerman and Szelepcsény [31, 69] have shown that $\mathsf{coNSPACE}(f) = \mathsf{NSPACE}(f)$ where $f(n) \geq \log n$, hence $\mathsf{coNL} = \mathsf{NL}$ and $\mathsf{coPSPACE} = \mathsf{PSPACE}$.

A function f that maps from $\{1,0\}^*$ to $\{1,0\}^*$ can be expressed as a language problem L_f, where the word $\langle x, i\rangle \in L_f$ if the i^{th} bit of $f(x)$ is 1. Let FP be the set of all functions from $\{1,0\}^*$ to $\{1,0\}^*$ computable in deterministic polynomial time. Let FL be the set of all functions from $\{1,0\}^*$ to $\{1,0\}^*$ computable in deterministic logarithmic space. Let FNL be the set of all functions from $\{1,0\}^*$ to $\{1,0\}^*$ computable in non-deterministic logarithmic space.

1.2.3.1 Uniform families of Boolean circuits

A *Boolean circuit* α is a labeled finite directed acyclic graph. Each vertex v (known as a gate) is of type AND ($\wedge$), OR ($\vee$), NOT ($\neg$), INPUT, or OUTPUT. An INPUT gate has indegree 0 (the indegree of a gate is referred to as fan-in). The inputs of α are given by a tuple $(x_1, \ldots, x_n)$ (where each $x_i \in \{1, 0\}$) of distinct input gates. A vertex with out-degree 0 is called an OUTPUT. The outputs of α are given by a tuple $(y_1, \ldots, y_m)$ (where each $y_i \in \{1, 0\}$) of distinct vertices.

A Boolean circuit α with inputs $(x_1, \ldots, x_n)$ and outputs $(y_1, \ldots, y_m)$ computes a function $f : \{0,1\}^n \to \{0,1\}^m$ as follows: input x_i is assigned the value of the i^{th} bit of the argument to the function. Every other gate v is assigned a value from $\{0, 1\}$ calculated (depending on its type) from the values of the gates connected from v's incoming edges, starting at the inputs and them moving through the circuit. The value of the function f is the values of the output gates $(y_1, \ldots, y_m)$ in which output y_j contributes the j^{th} bit of the output.

The *size* of a circuit α (size(α)) is the number of vertices in α. The *depth* of a circuit α (depth(α)) is the longest path from an input vertex to an output vertex in α. The time required by parallel model of computation to compute a function has a direct relationship with depth of a circuit required to compute the same function.

Boolean circuits are a *non-uniform model* of computation since the circuit is fixed for a given input length. To overcome this we can consider an infinite collection (known as a family) of circuits for each input length.

A *Boolean circuit family* $\{\alpha_n\}$ is a collection of circuits, each α_n computing a function $f^n : \{0,1\}^n \to \{0,1\}^{m(n)}$. The function computed by the family $\{\alpha_n\}$, denoted $f_\alpha : \{0,1\}^* \to \{0,1\}^*$ is defined by $f_\alpha(x) \equiv f^{|x|}(x)$. If the length of the output of each member of the family is 1, we call it a *language decider*. Note that each circuit in the family may be totally different from every other circuit. Without any restriction on the relationship between the circuits, a family may accept any binary language. To restrict this power we impose a condition on the elements of the family, this is known as a *uniformity condition*. For example a family $\{\alpha_n\}$ of Boolean circuits is logarithmic space (L) uniform if the transformation $1^n \to \alpha_n$ can be computed in $\log(\text{size}(\alpha_n))$ space on a Turing machine.

First order, FO, uniformity [33] is currently regarded [10, 33, 34] as the best form of uniformity for circuits especially when working with complexity classes contained in NL. In an FO-uniform circuit family, member α_n is first order definable from 1^n.

The complexity class AC^i is the set of problems solved by a FO-uniform circuit family of depth $\mathcal{O}(\log^i n)$ and size poly(n) where the gates in each circuit have unbounded fan-in. The set FAC^i is the set of functions computed by a FO-uniform circuit family of depth $\mathcal{O}(\log^i n)$ and size poly(n) where the gates in each circuit have unbounded fan-in. Thus AC^0 is the set of problems solved by a uniform family of polynomial sized circuits, with unbounded gate fan-in, in constant depth. We define $\mathsf{AC} = \cup_{i>0}\mathsf{AC}^i$.

The complexity class NC^i is the set of problems solved by a FO-uniform circuit family of depth $\mathcal{O}(\log^i n)$ and size poly(n) where the gates in each circuit have fan-in at most 2. The set FNC^i is the set of functions computed by a FO-uniform circuit family of depth $\mathcal{O}(\log^i n)$ and size poly(n) where the gates in each circuit have fan-in

at most 2. We define $\mathsf{NC} = \cup_{i>0}\mathsf{NC}^i$.

Note that for all $i \geq 0, \mathsf{NC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{NC}^{i+1}$, the former inclusion trivially holds and the latter follows from the fact that each logic gate with (a maximum) fan-in n can be replaced with a tree of gates with fan-in 2 and depth $\log n$. We can summarise the known relations between these classes as follows:

$$\mathsf{NC}^0 \subseteq \mathsf{AC}^0 \subsetneq \mathsf{NC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} = \mathsf{coNL} \subseteq \mathsf{NC}^2 \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$$

Also note that $\mathsf{L} \subsetneq \mathsf{PSPACE}$ [48].

1.2.3.2 Reductions and completeness

A language A is *reducible* to language B, (written as $A \leq B$) if there is a computable function $f : \Sigma^* \to \Sigma^*$, where $w \in A \iff f(w) \in B$. The function f is called a *reduction* from A to B. We say that L is C-*hard* (where C is a complexity class) if L can be reduced ($L \leq B$) to $B \in \mathsf{C}$. We say that L is C-*complete* if L is C-hard and $L \in \mathsf{C}$.

We must restrict the reduction to ensure that the problem is not made easier or harder by the reduction. In general it is desirable to have as weak a reduction as possible. In this thesis most reductions are computable in FAC^0 (denoted $\leq_{\mathsf{AC}^0}$).

1.2.4 Constant parallel time computations

Concurrent Random Access Machines (CRAMs) [32] are a parallel model of computation with many processors all operating on a global memory. They are synchronous (that is all the processors work in lock step) and concurrent (several processors may read and write from the same location at the same time-step). Each processor is identical except they each have a unique processor number (if several processors try to write to same bit of memory, then the lowest processor is the only one that succeeds). Each processor has a finite set of registers to store the following: the index of the processor, an address of global memory, and the line number of the instruction to be executed next. The instruction types and operations of a CRAM are: read, write, move, branch, assignment, addition, subtraction, and branch-on-less-than. Also permitted is a shift function, such that $\text{shift}(x, y)$ shifts the word x by y bits to the right, this allows each processor to access a part of the input word in constant time. There is a specific section of memory designated as the output. These basic instructions and operations can build up into useful functions that are used to allow us to specify the operation of the CRAM in a more general way.

Let $\mathsf{CRAM}(\mathcal{O}(1))$ be the set of problems solved by Concurrent Random Access Machines in constant time.

Theorem: 1.1 ([32])**.** $\mathsf{CRAM}(O(1)) = \mathsf{FO}$*–uniform* AC^0

Instead of providing a FO-uniform circuit for a reduction or uniformity condition, it is often easier to give a constant time CRAM algorithm.

Chapter 2

Membrane systems

Membrane systems with active membranes were first outlined [51] by Păun with their characteristic features of membranes charges (+,−,0) and division rules. Păun showed that such systems could solve SAT by mapping problem instances to membrane systems, this mapping would be later be defined as semi-uniformity. Obtułowicz [47] refined this mapping by providing a logspace Turing machine that constructed a membrane system to solve a given input SAT instance. Pérez-Jiménez et al. [55] formalised the definitions of polynomial time uniform and semi-uniform families, and so provided a framework to reason about the computational complexity of membrane systems. The active membrane model has been formally defined previously by Romero-Jiménez and Riscos-Núñez [61, 62]. These definitions were designed so that several varieties of membrane system could use the same framework. However, in this thesis we focus exclusively on active membrane systems and so have more specific definitions based on, or inspired by, those in the literature [30, 51, 52, 55, 61, 62, 67].

2.1 Active membrane systems without charges

In this thesis we exclusively discuss *active membrane systems without charges*. It is necessary to refer to the absence of charges since they are a defining characteristic in the original model definition [51].

Definition: 2.1. *An active membrane system without charges is a 6-tuple* $\Pi = (O, \mu, M, H, L, R)$ *where,*

1. O *is the alphabet of objects;*
2. $\mu = (V_\mu, E_\mu)$ *where* $V_\mu \subseteq \mathbb{N}_0$ *and* $E_\mu \subsetneq V_\mu \times V_\mu$ *is a tree representing the membrane structure;*
3. $M : V_\mu \to \mathrm{MS}(O)$ *maps membranes to their multisets;*
4. H *is the finite set of membrane labels;*
5. $L : V_\mu \to H$ *maps membranes to their labels;*

6. *R is a finite set of developmental rules of the following types (where $o, u, v \in O \times \{1\}$, $m \in \mathrm{MS}(O)$, $h, g, l \in H$):*

   (a) (a, o, h, m),

   (b) (b, o, h, u),

   (c) (c, o, h, u),

   (d) (d, o, h, u),

   (e) $(\mathrm{e}, o, h, \{u, v\})$,

   ($\mathrm{e_s}$) $(\mathrm{e_s}, o, h, \{u, u\})$,

   ($\mathrm{e_w}$) $(\mathrm{e_w}, o, h, \{u, v\})$,

   (f) $(\mathrm{f}, \{g, l\}, h, \{\})$.

   Note that

   - we use V_μ to refer to the set of membranes in a system Π.
   - The root vertex 0 of μ represents the ultimate parent of all membranes in a membrane structure. It is referred to as the "environment" and has label $0 \in H$. Since the environment has no parent, only type (a) rules are applicable to it.

The notation of rules in Definition 2.1 is in a non-standard and less readable format, this is to aid the formal definition of active membrane systems. In other chapters of this thesis we use the standard formating when specifying a membrane system. The semantics and formats of the rules are defined in Section 2.1.1.

To prevent extra information being included in the encoding of a membrane system, we define what is a permissible encoding. Previous membrane encoding schemes [55] specified for numbers to be encoded in binary. If a multiset is specified in the format a^3, b^2 instead of $[\, a, a, a, b, b \,]$ it becomes possible to encode an exponential number of objects in linear time. We consider this undesirable and so enforce that the multiplicity of objects in multisets must be encoded in unary (denoted $\widetilde{M}$) though the object types may be in binary. This restriction does not affect any previous results related with this thesis.

Definition: 2.2. *A* permissible encoding $\langle \Pi \rangle = \langle O, \mu, M, H, L, R \rangle$ *of a membrane system* Π *is a string from* $\{0, 1\}^*$ *with the following restrictions:*

- *O is encoded as a list of binary strings,*
- *$\mu = (V_\mu, E_\mu)$ is encoded as the adjacency matrix A_μ,*
- *Each multiset M is encoded explicitly in unary, that is in the form $\widetilde{M}$,*
- *H is encoded as a list of binary strings,*
- *L is encoded as set of tuples in binary,*
- *R is written as a sequence of 4-tuples, objects and labels may be written in binary however the multisets $\widetilde{m}$ in type (a) rules must be written out explicitly in unary*

- *We also augment the encoding scheme with the characters* $(,), \{, \}$, *and* , *which are used to delineate the above string a reasonable manner. The special space character* $\llcorner$ *may be inserted anywhere with no effect on the system's meaning.*

A configuration of a membrane system contains a complete description of all the information regarding the current state of the membrane system.

Definition: 2.3. *A* configuration $\mathcal{C}$ *of a membrane system is a tuple* (μ, M, L) *where*

- $\mu = (V_\mu, E_\mu)$, *where* $V_\mu \subseteq \mathbb{N}_0$ *and* $E \subsetneq V_\mu \times V_\mu$ *is a tree representing the current membrane structure,*
- $M : V_\mu \to \mathrm{MS}(O)$, *the multisets of objects in each membrane,*
- $L : V_\mu \to H$, *maps membranes to labels.*

A permissible encoding scheme of a configuration $\langle \mathcal{C} \rangle$ is defined in the same way as outlined in Definition 2.2.

2.1.1 Rules of active membrane systems

In this section we define the exact behaviour of each type of rule and how it affects a membrane configuration. For each rule type we define a predicate that is true if a rule of that type is applicable to a membrane in a configuration. For each rule type we define a function that when given a rule and a configuration, $\mathcal{C}$, returns the appropriate modified configuration, $\mathcal{C}'$. If the rule is not applicable to the membrane, the configuration is unchanged.

2.1.1.1 Rules of type (a)

Rules of type (a) are known as *object evolution rules* and are a tuple (a, o, h, m) where $o \in \{(x, 1)\}, x \in O, m \subseteq \mathrm{MS}(O), h \in H$. These rules are more commonly written in the form $[\, x \,]_h \to [\, \widetilde{m} \,]_h$ or $[\, x \to \widetilde{m} \,]_h$. A single object of type x in a membrane with label h is replaced by a multiset of new objects specified in m.

We define the predicate $\mathrm{applicable}_{\mathrm{a}}(\mathcal{C}, i, r)$ to be true iff $\exists i \in V_\mu$ such that $o \in M(i)$ and $L(i) = h$ where $r = (\mathrm{a}, o, h, m)$. We say a type (a) rule r is *applicable* in membrane i of a configuration $\mathcal{C}$ iff $\mathrm{applicable}_{\mathrm{a}}(\mathcal{C}, i, r)$ holds.

Given a configuration $\mathcal{C} = (\mu, M, L)$ and a rule $r = (\mathrm{a}, o, h, m)$ of type (a) applicable to membrane i, we say that this rule has been *applied* if a new configuration $\mathcal{C}' = \mathrm{apply}_{\mathrm{a}}(r, i, \mathcal{C})$ is generated.

$\mathcal{C}' = (\mu', M', L') = \mathrm{apply}_{\mathrm{a}}(r, i, \mathcal{C})$ such that
$\mu' = \mu$
$M' = \{(j, m') \mid j \in V_\mu,$ if $i = j$ then $m' = (M(j) \ominus o) \uplus m$ else $m' = M(j)\}$
$L' = L$.

The membrane structure and labels are unchanged by these rules. However in membrane i, an object o is removed and we add the multiset of objects m.

2.1.1.2 Rules of type (b)

Rules of type (b) are known as *communication in* rules and are a tuple (b, o, h, u) where $o \in \{(x,1)\}, u \in \{(y,1)\}, x, y \in O, h \in H$. These rules are more commonly written in the form $x\,[\,]_h \rightarrow [\,y\,]_h$. A single object of type x outside a membrane with label h is sent through the membrane to join that membrane's multiset, it may change in the process to an object of type y.

We define the predicate $\mathrm{applicable}_{\mathrm{b}}(\mathcal{C}, i, r)$ to be true iff $\exists i, j \in V_\mu$ such that $L(i) = h$ and $o \in M(j)$ and $j = \mathrm{parent}(\mu, i)$, where $r = (\mathrm{b}, o, h, u)$. We say a type (b) rule r is *applicable* in membrane i of a configuration $\mathcal{C}$ iff $\mathrm{applicable}_{\mathrm{b}}(\mathcal{C}, i, r)$ holds.

Given a configuration $\mathcal{C} = (\mu, M, L)$, and a rule $r = (\mathrm{b}, o, h, u)$ of type (b) applicable to membrane i, we say that this rule has been *applied* if a new configuration $\mathcal{C}' = \mathrm{apply}_{\mathrm{b}}(r, i, \mathcal{C})$ is generated.

$$
\begin{aligned}
&\mathcal{C}' = (\mu', M', L') = \mathrm{apply}_{\mathrm{b}}(r, i, \mathcal{C}) \text{ such that} \\
&\quad \mu' = \mu \\
&\quad M' = \{(j, m') \mid j \in V_\mu\} \\
&\qquad \text{where } m' = \begin{cases} M(j) \uplus u & \text{if } j = i \\ M(j) \ominus o & \text{if } j = \mathrm{parent}(\mu, i) \\ M(j) & \text{otherwise} \end{cases} \\
&\quad L' = L.
\end{aligned}
$$

This rule does not affect the membrane structure nor labels. In the parent of membrane i an object of type x is removed, and an object of type y is added to membrane i.

2.1.1.3 Rules of type (c)

Rules of type (c), known as *communication out* rules, are a tuple (c, o, h, u) where $o \in \{(x,1)\}, u \in \{(y,1)\}, x, y \in O, h \in H$. These rules are more commonly written in the form $[\,x\,]_h \rightarrow [\,]_h\, y$. A single object of type x inside a membrane with label h is sent through the membrane to join the parent membrane's multiset, x may be changed in the process to become an object of type y.

We define the predicate $\mathrm{applicable}_{\mathrm{c}}(\mathcal{C}, i, r)$ to be true iff $\exists i \in V_\mu, i \neq 0$ such that $L(i) = h$ and $o \in M(i)$, where $r = (\mathrm{c}, o, h, u)$. We say a type (c) rule r is *applicable* in membrane i of a configuration $\mathcal{C}$ iff $\mathrm{applicable}_{\mathrm{c}}(\mathcal{C}, i, r)$ holds.

Given a configuration $\mathcal{C} = (\mu, M, L)$, and a type (c) rule $r = (\mathrm{c}, o, h, u)$ applicable to membrane i, we say that this rule has been *applied* if a new configuration $\mathcal{C}' =$

$\text{apply}_\text{c}(r, i, \mathcal{C})$ is generated.

$$\begin{aligned}\mathcal{C}' = (\mu', M', L') = \text{apply}_\text{c}(r, i, \mathcal{C}) \text{ such that}\\ \mu' = \mu\\ M' = \{(j, m') \mid j \in V_\mu\},\\ \text{where } m' = \begin{cases} M(j) \ominus o & \text{if } i = j \\ M(j) \uplus u & \text{if } j = \text{parent}(\mu, i) \\ M(j) & \text{otherwise} \end{cases}\\ L' = L.\end{aligned}$$

The membrane structure and set of labels are unchanged by this rule. All membrane contents are the same except for membrane i which has o removed from it and the parent of i which has u added to it.

2.1.1.4 Rules of type (d)

Rules of type (d) are known as *dissolution* rules and are a tuple (d, o, h, u) where $o \in \{(x, 1)\}, u \in \{(y, 1)\}, x, y \in O, h \in H$. These rules are more commonly written in the form $[\, x \,]_h \rightarrow y$ where $x, y \in O$ and $h \in H$. A single object of type x inside a membrane with label h dissolves that membrane, the entire contents (child membranes and objects) of the dissolved membrane, are moved to the parent membrane, the *triggering* object x may be changed in the process to become an object of type y.

We define the predicate $\text{applicable}_\text{d}(\mathcal{C}, i, r)$ to be true iff $\exists i \in V_\mu, i \neq 0$ such that $L(i) = h$ and $o \in M(i)$, where $r = (\text{d}, o, h, u)$. We say a type (d) rule r is *applicable* to membrane i of a configuration $\mathcal{C}$ iff $\text{applicable}_\text{d}(\mathcal{C}, i, r)$ holds.

Given a configuration $\mathcal{C} = (\mu, M, L)$, and a type (d) rule $r = (\text{d}, o, h, u)$ applicable to membrane i, we say that this rule has been *applied* if a new configuration $\mathcal{C}' = \text{apply}_\text{d}(r, i, \mathcal{C})$ is generated.

$\mathcal{C}' = (\mu', M', L') = \text{apply}_\text{d}(r, i, \mathcal{C})$ and is defined as follows. The dissolved membrane i is removed from the membrane structure and the edge (p, i) from the parent p to i is removed along with the edges $E_c = \{(i, c) \in E_\mu \mid c \in V_\mu\}$ to the children of i. We then add a set of new edges that makes the parent of i become the parent of i's former children: $E_{c,p} = \{(p, c) \mid p, c \in V_\mu \text{ and } (p, i), (i, c) \in E_\mu\}$.

$$\begin{aligned}\mu' = (V'_\mu, E'_\mu) \text{ where}\\ V'_\mu = V_\mu \backslash i,\\ E'_\mu = E_\mu \backslash (\{(p, i)\} \cup E_c) \cup E_{c,p}\end{aligned}$$

The multiset of the dissolved membrane i is added to its parent membrane, removing the triggering object o and replacing it with u:

$$\begin{aligned}M' = \{(j, m') \mid j \in V'_\mu\}\\ \text{where } m' = \begin{cases} M(j) \uplus (M(i) \ominus o) \uplus u & \text{if } j = \text{parent}(\mu, i) \\ M(j) & \text{otherwise} \end{cases}\end{aligned}$$

The dissolved membrane i is removed from the set of membrane-label relations:

$$L' = L \backslash (i, L(i)).$$

2.1.1.5 Rules of type (e), elementary division

Rules of type (e), known as *elementary division* rules, are a tuple $(\mathrm{e}, o, h, \{u, v\})$ where $o \in \{(x, 1)\}, u \in \{(y, 1)\}, v \in \{(z, 1)\}, x, y, z \in O, h \in H$. These rules are commonly written in the form $[\,x\,]_h \rightarrow [\,y\,]_h [\,z\,]_h$. An elementary membrane has no child membranes. When this rule is applied to an elementary membrane with label h, the membrane is copied (multiset included) to create a new membrane with the same label. In the original membrane an object of type x is replaced by one of type y and in the copy with one of type z.

We define the predicate $\mathrm{applicable}_{\mathrm{e}}(\mathcal{C}, i, r)$ to be true iff $\exists i \in V_\mu, i \neq 0$ such that $L(i) = h$ and $o \in M(i)$ and i has no children $(\nexists c \in V_\mu, (i, c) \in E_\mu)$, where $r = (\mathrm{e}, o, h, \{u, v\})$. We say a type (e) rule r is *applicable* to membrane i of a configuration $\mathcal{C}$ iff $\mathrm{applicable}_{\mathrm{e}}(\mathcal{C}, i, r)$ holds.

Given a configuration $\mathcal{C} = (\mu, M, L)$, and a type (e) rule $r = (\mathrm{e}, o, h, \{u, v\})$ applicable to membrane i, we say that this rule has been *applied* if a new configuration $\mathcal{C}' = \mathrm{apply}_{\mathrm{e}}(r, i, \mathcal{C})$ is generated.
$\mathcal{C}' = (\mu', M', L') = \mathrm{apply}_{\mathrm{e}}(r, i, \mathcal{C})$ where

$$\mu' = (V_\mu \cup i', E_\mu \cup \{(\mathrm{parent}(\mu, i), i')\})$$

We add a new membrane i' with the same parents as i to the membrane structure.

$$M' = \{(j, m') \mid j \in V'_\mu\} \text{ where } m' = \begin{cases} (M(i) \ominus o) \uplus u & \text{if } j = i \\ (M(i) \ominus o) \uplus v & \text{if } j = i' \\ M(j) & \text{otherwise} \end{cases}$$

The multisets of i' and i are identical except o is replaced with u in one and with v in the other.

$$L' = L \cup (i', L(i))$$

The membrane i' has the same label as membrane i.

2.1.1.6 Rules of type ($\mathrm{e_s}$), symmetric elementary division

Rules of type ($\mathrm{e_s}$) are known as *symmetric elementary division* rules and are a tuple $(\mathrm{e}, o, h, \{u, u\})$ where $o \in \{(x, 1)\}, u \in \{(y, 1)\}, x, y \in O, h \in H$. These rules are commonly written in the form $[\,x\,]_h \rightarrow [\,y\,]_h [\,y\,]_h$. An elementary membrane has no child membranes. When this rule is applied an elementary membrane with label h is copied (multiset included) to create a new membrane with the same label. In the both membranes an object of type x is replaced by one of type y.

We define the predicate $\mathrm{applicable}_{\mathrm{e_s}}(\mathcal{C}, i, r)$ to be true iff $\exists i \in V_\mu, i \neq 0$ such that $L(i) = h$ and $o \in M(i)$ and i has no children $(\nexists c \in V_\mu, (i, c) \in E_\mu)$, where $r = (\mathrm{e}, o, h, \{u, u\})$. We say a type (e) rule r is *applicable* in membrane i of a configuration $\mathcal{C}$ iff $\mathrm{applicable}_{\mathrm{e_s}}(\mathcal{C}, i, r)$ holds.

Given a configuration $\mathcal{C} = (\mu, M, L)$, and a type ($\mathrm{e_s}$) rule $r = (\mathrm{e_s}, o, h, \{u, u\})$ applicable to membrane i, we say that this rule has been *applied* if a new configuration $\mathcal{C}' = \mathrm{apply}_{\mathrm{e_s}}(r, i, \mathcal{C})$ is generated.
$\mathcal{C}' = (\mu', M', L') = \mathrm{apply}_{\mathrm{e}}(r, i, \mathcal{C})$ where

$$\mu' = (V_\mu \cup i', E_\mu \cup (\mathrm{parent}(\mu, i), i'))$$

We add a new membrane i' with the same parents as i to the membrane structure.

$$M' = \{(j, m') \mid j \in V'_\mu\} \text{ where } m' = \begin{cases} (M(i) \ominus o) \uplus u & \text{if } j = i \vee j = i' \\ M(j) & \text{otherwise} \end{cases}$$

The multisets of i' and i are identical, o is replaced with u in both.

$$L' = L \cup (i', L(i))$$

The membrane i' has the same label as membrane i.

2.1.1.7 Rules of type ($\mathrm{e_w}$), weak non-elementary division

Rules of type ($\mathrm{e_w}$) are known as *weak non-elementary division* rules and are a tuple $(\mathrm{e_w}, o, h, \{u, v\})$ where $o \in \{(x, 1)\}, u \in \{(y, 1)\}, v \in \{(z, 1)\}, x, y, z \in O, h \in H$.

They are more commonly written in the format $[\,x\,]_h \rightarrow [\,y\,]_h[\,z\,]_h$. These rules are like rules of type (e) except they are permitted to operate on non-elementary membranes (that is, membranes with child membranes). The rule makes an isomorphic duplicate of an entire subtree of the membrane structure rooted at a membrane with label h including multisets and child membranes. Each membrane in the new subtree has the same label as its isomorphic pair membrane in the original structure. An object of type x is replaced by one of type y in the root membrane of one subtree and by one of type z in the other.

We define the predicate $\mathrm{applicable}_{\mathrm{e_w}}(\mathcal{C}, i, r)$ to be true iff $\exists i \in V_\mu, i \neq 0$ such that $L(i) = h$ and $o \in M(i)$, where $r = (\mathrm{e_w}, o, h, \{u, v\})$. We say a type ($\mathrm{e_w}$) rule r is *applicable* in membrane i of a configuration $\mathcal{C}$ iff $\mathrm{applicable}_{\mathrm{e_w}}(\mathcal{C}, i, r)$ holds.

Given a configuration $\mathcal{C} = (\mu, M, L)$ and a type ($\mathrm{e_w}$) rule $r = (\mathrm{e_w}, o, h, \{u, v\})$ applicable to membrane i, we say that this rule has been *applied* if a new configuration $\mathcal{C}' = \mathrm{apply}_{\mathrm{e_w}}(r, i, \mathcal{C})$ is generated.

Let "mark" be a function that when given a tree T, gives an isomorphic tree where each vertex k is renamed $\overline{k}$. We use the function subtree that was defined in Section 2.1. Let $\overline{T} = (\overline{V}, \overline{E}) = \mathrm{mark}(\mathrm{subtree}(\mu, i))$.
$\mathcal{C}' = (\mu', M', L') = \mathrm{apply}_{\mathrm{e_w}}(r, i, \mathcal{C})$ where

$$\mu' = (V'_\mu, E'_\mu) = (V_\mu \cup \overline{V}, E_\mu \cup \overline{E} \cup \{(\mathrm{parent}(\mu, i), \overline{i})\})$$

We augment the membrane structure by adding a copy of the subtree rooted at i to

the parent of i.

$$M' = \{(j, m') \mid j \in V'_\mu\} \text{ where } m' = \begin{cases} (M(j) \ominus o) \uplus u & \text{if } j = i \\ (M(j) \ominus o) \uplus v & \text{if } j = \overline{i} \\ M(k) & \text{if } j = \overline{k} \in \overline{V} \\ M(j) & \text{otherwise} \end{cases}$$

The multisets of unaffected membranes are the same. The multisets of the membranes of the new subtree are identical to the membrane they were copied from. In membrane i, an object of type o is replaced with an object of type u, and in membrane $\overline{i}$ an object of type o is replaced with one of type v.

$$L' = L \cup \{(\overline{j}, L(j)) \mid \overline{j} \in V'_\mu\}$$

The label of each new membrane is identical to its isomorphic pair membrane.

2.1.1.8 Rules of type (f), strong non-elementary division

Rules of type (f) are known as *strong non-elementary division* rules and are a tuple $(\mathrm{f}, \{g, l\}, h, \{\})$ where $h, g, l \in H$.

They are more commonly written in the format $[\,[\,]_g\,[\,]_l\,]_h \rightarrow [\,[\,]_g\,]_h\,[\,[\,]_l\,]_h$. These rules are of a different nature to all the other rules discussed here as they are triggered by membranes and not objects. The rule duplicates the entire subtree of the membrane structure rooted at a membrane with label h containing a membrane with label g and another with label l. One duplicate does not contain the subtree rooted by the membrane with label g and the other does not contain the subtree rooted by the membrane with label l. Each membrane in the new subtree has the same label as its isomorphic pair membrane in the original structure (except those in the g and l subtrees).

We define the predicate $\text{applicable}_{\mathrm{f}}(\mathcal{C}, i, r)$ to be true iff $\exists i \in V_\mu, i \neq 0$ such that $L(i) = h$ and $\exists j, k \in V_\mu$ such that $i = \text{parent}(\mu, j)$ and $i = \text{parent}(\mu, k)$ and $g = L(j)$ and $l = L(k)$, where $r = (\mathrm{f}, \{g, l\}, h, \{\})$. We say a type (f) rule r is *applicable* in membrane i of a configuration $\mathcal{C}$ iff $\text{applicable}_{\mathrm{f}}(\mathcal{C}, i, r)$ holds.

Given a configuration $\mathcal{C} = (\mu, M, L)$, and a type (f) rule $r = (\mathrm{f}, \{g, l\}, h, \{\})$ applicable to membrane i, we say that this rule has been *applied* if a new configuration $\mathcal{C}' = \text{apply}_{\mathrm{f}}(r, i, \mathcal{C})$ is generated.

Let mark be a function that when given a tree T, gives an isomorphic tree where each vertex k is renamed $\overline{k}$. We use the function "subtree" that was defined in Section 2.1. Let $\overline{T} = (\overline{E}, \overline{V}) = \text{mark}(\text{subtree}(\mu, i))$. Let $T_g = (\overline{E}_g, \overline{V_g}) = \text{subtree}(\mu, j)$ where $g = L(j)$ and $i = \text{parent}(\mu, j)$. Let $\overline{T}_l = (\overline{E}_l, \overline{V_l}) = \text{subtree}(\overline{T}, \overline{k})$ where $l = L(\overline{k})$ and $\overline{i} = \text{parent}(\mu, \overline{k})$.
$\mathcal{C}' = (\mu', M', L') = \text{apply}_{\mathrm{e_w}}(r, i, \mathcal{C})$ where

$$\mu' = (V'_\mu, E'_\mu) = ((V_\mu \backslash V_g) \cup (\overline{V} \backslash \overline{V}_l), \\ (E_\mu \backslash E_g) \cup (\overline{E} \backslash \overline{E}_l) \cup \{(\text{parent}(\mu, i), \overline{i})\})$$

We augment the membrane structure by making an isomorphic copy of the membrane structure subtree rooted at i, we refer to it as $\overline{T}$. We remove from children of i a membrane (and its subtree) with label g and from $\overline{T}$ we remove a membrane with label l (and its subtree). We join the subtree $\overline{T}$ back to the membrane structure by setting its parent to be the parent of i.

$$M' = \{(j, m') \mid j \in V'_\mu\} \text{ where } m = \begin{cases} M(j) & \text{if } j = \overline{j} \in \overline{V} \\ M(j) & \text{otherwise} \end{cases}$$

The multisets of all unaffected membranes are the same. The multisets of the membranes of the new subtree are identical to the membrane they were copied from.

$$L' = L \cup \{(\overline{j}, L(j)) \mid j \in V'_\mu\}$$

The label of each new membrane is identical to its original equivalent membrane.

2.2 Computation of a membrane system

We now explain how the rules of a membrane system are applied to a configuration. A multiset of rules that are all applicable to this configuration in this time-step is non-deterministically selected such that it is impossible to add another applicable rule to the multiset. There is a copy of a rule in the multiset for each individual object it is applicable to, also this multiset contains at most one rule of types (b)–(f) for each membrane. This is known as a *maximal* multiset of rules.

Definition: 2.4 (Maximal multiset of rules)**.** *A* multiset of rules *with respect to a configuration $\mathcal{C}$ of an active membrane system is a multiset $\mathcal{R}(\mathcal{C})$ such that $\mathcal{R}(\mathcal{C}) = [(i, r) \mid i \in V_\mu, r \in R$, r is a rule of type τ and* applicable$_\tau(\mathcal{C}, r, i)$ *holds*].

This multiset $\mathcal{R}(\mathcal{C})$ is said to be a maximal multiset $\mathcal{R}_{\max}(\mathcal{C})$ *for a configuration if it satisfies the following conditions.*

1. *For each tuple $(i, r) \in \mathcal{R}(\mathcal{C})$, there is a single unique object in $\mathcal{C}$ that r is applied to (unless r is a type (f) rule).*
2. *For each membrane i in $\mathcal{C}$, there is at most one $(i, r) \in \mathcal{R}(\mathcal{C})$ where r is of types (b)–(f).*
3. *All the rules can be applied in a single time-step.*
4. *It is impossible to augment the multiset with another rule that does not violate the above conditions.*

We then apply these rules to each membrane in the system, starting with the most deeply nested. The rules are applied in the order a, b, c, d, e, e_s, e_w, f.

Given a configuration $\mathcal{C}_t$ and a maximal set of rules $\mathcal{R}_{\max}(\mathcal{C})$ for that configuration, we move to configuration $\mathcal{C}_{t+1}$ via a parallel application of the rules in the multiset. This is complicated process and we define a number of functions which we use to define a time-step of the computation.

Note that by calculating the multiset of applicable rules from the configuration at the start of the computation step we ensure that no object generated by the application of a rule is used by another rule in the same computation step.

First, we define function $\text{applyType}(\tau, \mathcal{R}, i, \mathcal{C})$ which applies all rules from the multiset $\mathcal{R}_{\max}(\mathcal{C})$ of type $\tau \in \{\text{a}, \text{b}, \text{c}, \text{d}, \text{e}, \text{e}_\text{s}, \text{e}_\text{w}, \text{f}\}$ associated with membrane i. That is we construct a new multiset of rules $\mathcal{R}_i = [r_0, r_1, \ldots, r_{x-1}, r_x]$ which are all the rules that are applicable to membrane i and its contents. Formally: $Q_{\mathcal{R}_i} = \{(r, q) \mid ((i, r), q) \in Q_{\mathcal{R}}\}$.

$$\begin{aligned}\text{applyType}(\tau, \mathcal{R}, i, \mathcal{C}) = \text{apply}_\tau(r_x, i,\\ \text{apply}_\tau(r_{x-1}, i,\\ \ldots\\ \text{apply}_\tau(r_1, i,\\ \text{apply}_\tau(r_0, i, \mathcal{C}))\ldots))\end{aligned}$$

Next we define the function $\text{applyRules}(\mathcal{R}, i, \mathcal{C})$ which applies all the rules in the multiset $\mathcal{R}$ that are applicable to the membrane i and outputs the resulting configuration. The rules are applied in the order a, b, c, d, e, e_s, e_w, f.

$$\begin{aligned}\text{applyRules}(\mathcal{R}, i, \mathcal{C}) = \text{applyType}(\text{f}, \mathcal{R}, i,\\ \text{applyType}(\text{e}_\text{w}, \mathcal{R}, i,\\ \text{applyType}(\text{e}_\text{s}, \mathcal{R}, i,\\ \text{applyType}(\text{e}, \mathcal{R}, i,\\ \text{applyType}(\text{d}, \mathcal{R}, i,\\ \text{applyType}(\text{c}, \mathcal{R}, i,\\ \text{applyType}(\text{b}, \mathcal{R}, i,\\ \text{applyType}(\text{a}, \mathcal{R}, i, \mathcal{C}))))))))\end{aligned}$$

We define the function $\text{takeStep}(\mathcal{R}, \mathcal{C})$ which applies all applicable rules in $\mathcal{R}$ to each membrane in a configuration $\mathcal{C}$ and gives the resulting configuration. The rules are applied using the function applyRules on each membrane in a depth first ordering of the membrane structure, that is, starting from the most deeply nested membrane and working outwards towards the skin. Let $(i_0, i_1, \ldots, i_{|V_\mu|-1})$ be the depth first ordering of μ.

$$\begin{aligned}\text{takeStep}(\mathcal{R}, \mathcal{C}) = \text{applyRules}(\mathcal{R}, i_{|V_\mu|-1},\\ \text{applyRules}(\mathcal{R}, i_{|V_\mu|-2},\\ \ldots\\ \text{applyRules}(\mathcal{R}, i_1,\\ \text{applyRules}(\mathcal{R}, i_0, \mathcal{C}))\ldots))\end{aligned}$$

Definition: 2.5. *Given $\mathcal{C}_t$, a configuration of a membrane system, we say that $\mathcal{C}_t$* yields *configuration $\mathcal{C}_{t+1}$, denoted $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$, when a maximal multiset of rules $\mathcal{R}_{\max}(\mathcal{C}_t)$ is applied to $\mathcal{C}_t$. That is $\mathcal{C}_{t+1} = \text{takeStep}(\mathcal{R}_{\max}(\mathcal{C}_t), \mathcal{C}_t)$.*

Definition: 2.6. *A* computation *of a membrane system is a sequence of k configurations, denoted $\mathcal{C}_t \vdash^k \mathcal{C}_{t+k}$, such that each configuration yields the successor. We denote an unspecified number of configurations as $\vdash^*$.*

Note that when constructing the maximal multiset of rules for an active membrane system, the choices are non-deterministic, therefore on a given input there are multiple possible computations.

A computation halts when a configuration is reached where no more rules are applicable:

Definition: 2.7. *A computation* halts *when $|\mathcal{R}_{\max}(\mathcal{C})| = 0$.*

Definition: 2.8. *Given a configuration $\mathcal{C}_s$ of an $\mathcal{AM}^0_{-d}$ system Π containing an object of type object o_s in a membrane labeled h_s. We say an object type o_s in a membrane labeled h_s* eventually evolves *object type o_t in membrane labeled h_t if there is a computation where an unbroken sequence of rules act link the object type o_s as it changes type and membrane until it appears as object type o_t in a membrane labeled h_t. In other words the predicate* $\text{evEv}(\mathcal{C}_s, o_s, h_s, o_t, h_t)$ *holds.*

$$\text{evEv}(\mathcal{C}_s, o_s, h_s, o_t, h_t) = \begin{cases} \textit{true} & \textit{if } o_s = o_t, \textit{ and } h_s = h_t \\ \textit{false} & \textit{if } |\mathcal{R}_{\max}(\mathcal{C}_s)| = 0 \textit{ and } (o_s \neq o_t, \textit{ or } h_s \neq h_t) \\ \text{evEv}(\mathcal{C}_i, o_i, h_i, o_t, h_t) & \textit{if } \exists \mathcal{R}_{\max}(\mathcal{C}_s) \textit{ s. t. } \mathcal{C}_s \vdash \mathcal{C}_i \textit{ via } \mathcal{R}_{\max}(\mathcal{C}_s) \textit{ where} \\ & \quad o_i \in O_i, \; h_i \in H_i \textit{ and} \\ & \quad \text{matches}(o_s, h_s, o_i, h_i, \mathcal{R}_{\max}(\mathcal{C}_s)). \end{cases}$$

The predicate $\text{matches}(a, h, b, g, \mathcal{R})$ *holds if an object a in membrane h becomes object b in membrane g via a rule in a given multiset $\mathcal{R}$.*

$$\text{matches}(a, h, b, g, \mathcal{R}) = \begin{cases} \textit{true} & \textit{if } (\text{a}, a, h, m) \in \mathcal{R} \; \wedge b \in m \wedge h = g \\ \textit{true} & \textit{if } (\text{b}, a, g, b) \in \mathcal{R} \; \wedge h = L(\text{parent}(\mu, y)) \wedge g = L(y) \\ \textit{true} & \textit{if } (\text{c}, a, h, b) \in \mathcal{R} \; \wedge g = L(\text{parent}(\mu, y)) \wedge h = L(y) \\ \textit{true} & \textit{if } (\text{e}, a, h, B) \in \mathcal{R} \wedge h = h' \wedge b \in B = \{b, c\} \\ \textit{true} & \textit{if } (\text{e}_\text{s}, a, h, B) \in \mathcal{R} \wedge h = h' \wedge b \in B = \{b, b\} \\ \textit{true} & \textit{if } (\text{e}_\text{w}, a, h, B) \in \mathcal{R} \wedge h = h' \wedge b \in B = \{b, c\} \\ \textit{false} & \textit{otherwise.} \end{cases}$$

2.3 Confluent recogniser membrane systems

When we treat membrane systems as language deciding devices to solve decision problems, we call these *recogniser membrane systems*. Each computation of a recogniser system outputs either object `yes` or `no` to a specified output membrane in the system (normally the environment, the root of the membrane structure with label 0). The output of a computation is read only when it has reached a halting configuration.

Definition: 2.9. *A* recogniser membrane system *is a membrane system* Π *such that*

1. *all computations halt,*
2. $\texttt{yes}, \texttt{no} \in O$*,*
3. *the object* `yes` *or object* `no` *(but not both) appear in the multiset of the membrane with label* 0 *(the environment),*
4. *and this happens only in the halting configuration.*

If the `yes` object arrives in the membrane with label 0 (the environment) the computation is accepting. If the `no` object arrives in the membrane with label 0 (the environment) the computation is rejecting.

A language is a set $X = \{x_1, x_2, \ldots\} \subseteq \Sigma^*$ where Σ is some finite alphabet.

We consider infinite *families* $\mathbf{\Pi}$ of active membrane systems. We say that a family $\mathbf{\Pi}$ of membrane systems *decides* X if for any string $x \in \Sigma^*$

$$\mathbf{\Pi}(x) = \begin{cases} \text{accept} & \text{if } x \in X, \\ \text{reject} & \text{if } x \notin X. \end{cases}$$

That is that each instance of the problem is solved by some family member.

The family $\mathbf{\Pi}$ is *sound* with respect to X when, for each $x \in \Sigma^*$, if there exists an accepting computation of $\mathbf{\Pi}(x)$ then $x \in X$. The family $\mathbf{\Pi}$ is *complete* with respect to X when for each $x \in \Sigma^*$ if $x \in X$ then every computation of $\mathbf{\Pi}(x)$ is an accepting computation.

A membrane system is *deterministic* if for each input there is a unique computation.

Definition: 2.10. *A membrane system is said to be* confluent *if it is both sound and complete. That is, a membrane system* Π *is* confluent *if all computations of* Π *with the same input* x *(properly encoded) give the same result; either always "accept" or else always "reject".*

All membrane systems in this thesis are confluent.

2.4 Uniformity and semi-uniformity

Without further restrictions, recogniser active membrane systems are a non-uniform model of computation, that is there may be a different device for each input size. This means, like circuits, we consider an infinite family of recogniser active membrane systems to cover all potential input strings. However, if we can invest unbounded amounts of computation in order to construct each member of the family, then it can potentially solve uncomputable problems. To ensure that the algorithm (or function) that constructs each member of the family does not artificially increase the set of problems decided by the family we impose that the constructing algorithm (or function) is computable within certain restrictions on its time and/or space resource usage.

When the function maps a single input length to a membrane system that decides all inputs of that length, then the function is called a *uniformity condition.* When

the function maps a single input word to a membrane system that decides that input, then the function is called a *semi-uniformity condition.*

The notions of uniformity and semi-uniformity were first formally specified for membrane systems in 2003 [55] the most recent version is to be found in [54].

Definition: 2.11 (Class of problems solved by a uniform family)**.** *Let $\mathcal{R}$ be a class of recogniser membrane systems and let $t : \mathbb{N} \to \mathbb{N}$ be a total function. Let E, F be classes of functions. The class of* problems solved by an (E, F)-uniform family of membrane systems *of type $\mathcal{R}$ in time t, denoted (E, F)–$\mathsf{MC}_{\mathcal{R}}(t)$, contains all problems X such that:*

- *There exists an* F-uniform *family of membrane systems, $\mathbf{\Pi} = \{\Pi_1, \Pi_2, \ldots\}$ of type $\mathcal{R}$: that is, there exists a function $f \in \mathsf{F}$, $f : \{1\}^* \to \mathbf{\Pi}$ such that $f(1^n) = \Pi_n$, where $|x| = n$.*
- *There exists an input encoding function $e \in \mathsf{E}$, $e : X \cup \overline{X} \to \mathrm{MS}(I)$ such that $e(x)$ is the input multiset, which is placed in a specific input membrane of Π_n, where $|x| = n$ and $I \subsetneq O$ is the set of input objects.*
- *$\mathbf{\Pi}$ is t-efficient: Π_n always halts in at most $t(n)$ steps.*
- *The family $\mathbf{\Pi}$ is* sound *with respect to (X, e, f); that is, for each $x \in X$ there exists an accepting computation of the system $\Pi_{|x|}$ on input multiset $e(x)$.*
- *The family $\mathbf{\Pi}$ is* complete *with respect to (X, e, f); that is, for each input $x \in X$, then every computation of the system $\Pi_{|x|}$ on input multiset $e(x)$ is accepting.*

We define the set of languages decided by a uniform family of membrane systems in polynomial time to be

$$(\mathsf{E}, \mathsf{F})\text{–}\mathsf{PMC}_{\mathcal{R}} = \bigcup_{k \in \mathbb{N}_0} (\mathsf{E}, \mathsf{F})\text{–}\mathsf{MC}_{\mathcal{R}}(n^k).$$

In this thesis we exclusively use active membrane systems without charges ($\mathcal{AM}^0$) so letting $\mathcal{R} = \mathcal{AM}^0$ we refer to the set of languages decided by uniform families of active membrane systems in polynomial time as (E, F)–$\mathsf{PMC}_{\mathcal{AM}^0}$.

Semi-uniformity is a generalisation of uniformity. Here a single function (rather than two) is used to construct the semi-uniform membrane family, and the problem instance is encoded using objects, membranes, and rules. In this case, for each instance of $x \in X \cup \overline{X}$ we have a (possibly unique) membrane system which does not need a separately constructed input.

Definition: 2.12 (Class of problems solved by a semi-uniform family)**.** *Let H be a class of functions. The class of* problems solved by a (H)-semi-uniform family of membrane systems *of type $\mathcal{R}$ in time t, denoted (H)–$\mathsf{MC}^*_{\mathcal{R}}(t)$, contains all problems X such that:*

- *There exists a* H-semi-uniform *family $\mathbf{\Pi} = \{\Pi_{x_1}, \Pi_{x_2}, \ldots\}$ of membrane systems of type $\mathcal{R}$: that is, there exists a function $h \in \mathsf{H}$, $h : X \cup \overline{X} \to \mathbf{\Pi}$ such that $h(x_i) = \Pi_{x_i}$.*

- **Π** *is t-efficient:* Π_{x_n} *always halts in at most* $t(|x_n|)$ *steps.*
- *The family* **Π** *is* sound *with respect to (X,h); for each* $x \in X \cup \overline{X}$ *if there exists an accepting computation of the system* Π_x *then* $x \in X$*.*
- *The family* **Π** *is* complete*, with respect to (X, h); that is, for each* $x \in X$ *every computation of the system* Π_x *is accepting.*

We define the set of languages decided by a semi-uniform family of membrane systems in polynomial time to be

$$(\mathsf{H})\text{–}\mathsf{PMC}^*_{\mathcal{R}} = \bigcup_{k\in\mathbb{N}_0} (\mathsf{H})\text{–}\mathsf{MC}^*_{\mathcal{R}}(n^k).$$

Observation: 2.13. *Every uniform membrane system family can be represented as a semi-uniform family. Thus* (E,F)–$\mathsf{PMC}_{\mathcal{R}} \subseteq (\mathsf{H})$–$\mathsf{PMC}^*_{\mathcal{R}}$*.*

In this thesis we exclusively use active membrane systems without charges ($\mathcal{AM}^0$) so we let $\mathcal{R} = \mathcal{AM}^0$ and refer to (H)–$\mathsf{PMC}^*_{\mathcal{AM}^0}$.

Definition: 2.14 (P-semi-uniform active membrane systems)**.** *When* $\mathsf{H} = \mathsf{FP}$ *then the set of problems solved by semi-uniform families of active membrane systems is denoted* (P)–$\mathsf{PMC}^*_{\mathcal{AM}^0}$ *(normally written as* $\mathsf{PMC}^*_{\mathcal{AM}^0}$ *in the literature).*

Definition: 2.15 (P-uniform active membrane systems)**.** *When* $\mathsf{E},\mathsf{F} = \mathsf{FP}$ *then the set of problems solved by uniform families of active membrane systems is denoted* (P,P)–$\mathsf{PMC}_{\mathcal{AM}^0}$ *(normally written as* $\mathsf{PMC}_{\mathcal{AM}^0}$ *in the literature)*

Definition: 2.16 (L-semi-uniform active membrane systems)**.** *When* $\mathsf{H} = \mathsf{FL}$ *then the set of problems solved by semi-uniform families of active membrane systems is denoted* (L)–$\mathsf{PMC}_{\mathcal{AM}^0}$*.*

Definition: 2.17 (L-uniform active membrane systems)**.** *When* $\mathsf{E},\mathsf{F} = \mathsf{FL}$ *then the set of problems solved by uniform families of active membrane systems is denoted* (L,L)–$\mathsf{PMC}_{\mathcal{AM}^0}$*.*

Definition: 2.18 (AC^0-semi-uniform active membrane systems)**.** *When* $\mathsf{H} = \mathsf{FAC}^0$ *then the set of problems solved by semi-uniform families of active membrane systems is denoted* (AC^0)–$\mathsf{PMC}_{\mathcal{AM}^0}$*.*

Definition: 2.19 (AC^0-uniform active membrane systems)**.** *When* $\mathsf{E},\mathsf{F} = \mathsf{FAC}^0$ *then the set of problems solved by uniform families of active membrane systems is denoted* $(\mathsf{AC}^0,\mathsf{AC}^0)$–$\mathsf{PMC}_{\mathcal{AM}^0}$*.*

2.4.1 Uniformity definitions from other works

The definitions in Section 2.4 differ slightly in terms of notation from the canonical uniformity definitions for membrane systems [54] but describe the same concepts. However in this thesis we frequently change uniformity conditions and aim to be more in the tradition of uniform families of circuits.

We quote for contrast the definition for uniform families from [54] (note that $w \in I_X$). A decision problem for a language $X = (I_X, \theta_X)$ where $I_X = \Sigma^*$ and θ_X is the characteristic function for the language X.

Definition: (Encoding of a family of membrane systems with input [54]). *Let* $X = (I_X, \theta_X)$ *be a decision problem, and* $\mathbf{\Pi} = \{\Pi(n) : n \in \mathbb{N}\}$ *a family of recogniser P systems with input membrane. A* polynomial encoding *of* X *in* $\mathbf{\Pi}$ *is a pair* (cod, s) *of polynomial time computable functions over* I_X *such that for each instance* $w \in I_X$, $s(w)$ *is a natural number (obtained by means of a reasonable encoding scheme) and* $cod(w)$ *is an input multiset of the system* $\Pi(s(w))$.

The function s preforms a similar role as function f in Definition 2.11 in that they both define the members of the family. The function f constructs a membrane system instance from a number in unary, while the function s takes a problem instance and maps it to a number representing the appropriate membrane system. The function s has full access to the input word, the requirement that s is "reasonable" prevents it from computing a one to one mapping from input word to membrane system. The domain of function f is restricted to the input word length in unary, this ensures exponentially fewer membrane systems in a family than input words.

Our encoding function e in Definition 2.11 is a renaming of the *cod* function.

2.5 Unique labels

Here we describe a normal form for active membrane systems where every membrane in the membrane system has a unique label.

Normal Form: 2.20. *Any* $\mathcal{AM}^0$ *system* Π, *with* $m = |V_\mu|$ *membranes and* $l = |H|$ *labels that halts in* t *steps can be simulated by a* $\mathcal{AM}^0$ *system,* Π', *that has* m *membranes,* m *labels and halts in* t *steps.*

Proof. Given Π we create Π' by the following method. For each membrane $i \in V_\mu$ we add to the set R' a copy of every rule in R that mentions $h = L(i)$ but replace h with i in the rule. We then let L' to be the identity function and let $H' = V_\mu$.

Since there is now a copy of each rule for each applicable membrane the system Π' accepts iff the Π does. □

2.6 Dependency Graphs

The *dependency graph* (introduced by Gutiérrez-Naranjo et al. [30]) is an indispensable tool for characterising the computational complexity of membrane systems without dissolution. This technique is reminiscent of configuration graphs for Turing Machines. Similarly to a configuration graph, a dependency graph helps visualise a computation. However, it differs in its approach by representing a membrane system configuration as a subset of vertices rather than as a single vertex in configuration space.

Looking at membrane systems without dissolution as directed graphs allows us to employ the existing, mature corpse of techniques and complexity results for graph problems. As we see in Chapters 3 and 4, this greatly simplifies the process of proving upper-bounds and lower-bounds for such systems.

Each vertex in a dependency graph represents an object-membrane pair. An edge (a, b) exists in the dependency graph of Π if there is a rule in Π such that the left

hand side of the rule has an object-membrane pair matching a and the right hand side has an object-membrane pair matching b. Since membrane dissolution (type (d)) rules are not allowed, the parent/child relationships in the membrane structure tree of Π does not change during a computation. Thus when creating the edges for communication rules (types (b) and (c)) we can find the parent and child membranes for these rules at the beginning of the simulation and these choices remain the same for the entire computation (for example, to represent the rule $x[\ ]_h \to [x]_h$, that communicates an object x into a membrane of label h, it is only necessary to calculate the parent of h *one time* in the construction of the dependency graph). Note that in this section, for the sake of clarity, we overload the notation of multisets with a single object such that x represents both $x \in O$ and also the multiset $[x]$.

Definition: 2.21. *Let* Π *be a recogniser active membrane system without polarizations and without dissolution rules (*$\mathcal{AM}^0_{-d}$*). Let* R *be the set of rules associated with* Π*. The* dependency graph associated with Π *is the directed graph* $\mathcal{G}_\Pi = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \texttt{yes}, \texttt{no})$ *defined as follows:*

$\mathcal{V} = O \times H, \mathcal{E} = \{((x,h),(y,h')) :$
$((\mathrm{a}, x, h, \widetilde{m}) \in R \ \wedge y \in \mathrm{support}(m) \wedge h = h') \vee$
$((\mathrm{b}, x, h', y) \in R \ \wedge h = L(\mathrm{parent}(\mu, i)) \wedge h' = L(i)) \vee$
$((\mathrm{c}, x, h, y) \in R \ \wedge h' = L(\mathrm{parent}(\mu, i)) \wedge h = L(i)) \vee$
$((\mathrm{e}, x, h, \{y, z\}) \in R \wedge h = h' \vee$
$((\mathrm{e_s}, x, h, \{y, y\}) \in R \wedge h = h' \vee$
$((\mathrm{e_w}, x, h, \{y, z\}) \in R \wedge h = h'$

We also mention the special vertices $\texttt{yes} = (\texttt{yes}, 0)$ *representing the object* `yes` *in the output membrane and* $\texttt{no} = (\texttt{no}, 0)$ *representing the object* `no` *in the output membrane. Let* $\mathcal{I} = \{(x,i) \in \mathcal{V} \mid i \in V_\mu$ *where* $x \in M(i)\}$*. If a specific input set,* I*, is specified we add a further restriction:* $x \in I$*. The vertex* $\texttt{i} \in \mathcal{I}$*.*

Note that rules of type (f) do not contribute an edge to the graph since no objects are involved.

Lemma: 2.22. *Given an encoding of a membrane system* $\langle\Pi\rangle$*, its dependency graph* $\mathcal{G}_\Pi$ *is constructable in* FAC^0*.*

Proof. We provide a constant time CRAM algorithm that when given an encoding of a membrane system constructs an encoding of the dependency graph for that system. We assume that every membrane in the system has a unique label (via Normal Form 2.20).

The CRAM algorithm maps the set of rules R to the adjacency matrix $A_{\mathcal{G},\, n \times n}$ where $n = |H| \cdot |O|$. We assume that the rules are arranged in a matrix-like structure with $|R|$ columns and n^k rows where n^k is the biggest multiset $(\widetilde{m})$ in a rule of type (a) in R. The rules are formatted $\langle(\tau, x, h, \widetilde{m})\rangle$ so the first column stores the type of the rule, the second the triggering object x, the third the membrane h, and the remaining n^k each store a element in the multiset m. If the rule is not of type (a), then it has y in the fourth column or the elements $\{y, z\}$ in the fourth and fifth columns (all other columns are blank "␣"). This rules matrix is read by an equally sized matrix of processors, each processor reading an element of the rules matrix. In the first step,

a column of $|R|$ processors identifies the type of rule, the remaining processors behave differently depending on type.

If the rule is of type (a), it is formatted $(\mathrm{a}, x, h, \widetilde{m})$. One processor reads the object x and the label h, and the other n^k processors each read a single object y in $\widetilde{m}$. The values x and h are stored in shared registers. If one of the n^k processors reads a blank ␣ spacing character, it writes nothing out. The other processors who read an $y \in m$ then read the values x and h and write out a 1 to element $a_{\mathcal{G},\,(x,h),(y,h)}$ in the adjacency matrix. If two or more processors try to write to the same output register, only one (it does not matter which) succeeds.

If the rule is of type (b), it is formatted $\langle(\mathrm{b}, x, h', y)\rangle$: to write out the edge for this rule the CRAM needs to refer to the membrane structure μ of the encoded as adjacency matrix A_μ in membrane system $\langle\Pi\rangle$. Three processors load x, h' and y, (note that $n^k - 1$ of the processors read a blank symbol "␣" and do nothing). However to write out the edge we need to know the parent membrane of h', this value is written to a shared register. There there is a processor for each column p in A_μ, each reads the variable h' from the shared register, if element $a_{\mu,\,p,h'} = 1$ is then that processor writes p to a shared register (only one processor will do this since μ is a tree). The processor for the rule then reads the value p and writes a 1 in position $a_{\mathcal{G},\,(x,p),(y,h')}$ to the output register.

If the rules is of type (c), the rule format is $\langle(\mathrm{c}, x, h, y)\rangle$. Again we need to make reference to the membrane structure μ to find the parent of h. The process is similar to that for type (b) rules except the processor writes out a 1 to $a_{\mathcal{G},\,(x,h),(y,p)}$.

If the rule is of type (e), ($\mathrm{e_s}$), or ($\mathrm{e_w}$) then it is in the format $\langle(\tau, x, h, \{y, z\})\rangle$ (where $\tau \in \{\mathrm{e}, \mathrm{e_s}, \mathrm{e_w}\}$). In this case the CRAM writes 1 to $a_{\mathcal{G},\,(x,h),(y,h)}$, it writes out a second 1 at $a_{\mathcal{G},\,(x,h),(y,h)}$ if the rule is of type ($\mathrm{e_s}$) or ($\mathrm{e_w}$).

Rules of type (f) are ignored by the algorithm.

Thus a dependency graph is constructable by a CRAM (hence in FAC^0) given an encoded membrane system $\langle\Pi\rangle$. □

Dependency graphs are a traumatic simplification of membrane systems and it can be difficult to believe that they can be used to decide if an $\mathcal{AM}^0_{-d}$ system accepts. We now show a proof of correctness that is designed to make the reader feel more comfortable with dependency graphs by describing a series of small steps to lead us from a membrane system to a dependency graph. For a more laconic proof see the papers by Gutiérrez-Naranjo et al. [30, 29].

Lemma: 2.23. *Given a recogniser $\mathcal{AM}^0_{-d}$ system Π (in unique labels normal form) and its dependency graph $\mathcal{G}$, there exists a path from an object in $\mathcal{I}$ to* `yes`*, and no paths from an object in $\mathcal{I}$ to* `no` *iff Π halts in an accepting configuration.*

Proof. Given the membrane system Π we first convert it to the unique label normal form (Normal Form 2.20).

We now simulate a computation of a $\mathcal{AM}^0_{-d}$ system using as our data structure, a graph $G = (V, E)$ and a set of counters $c : V \to \mathbb{N}_0$ associated to each vertex. For each element of $O \times H$ there is a vertex in V. We define the special vertices `yes` $= (\mathtt{yes}, 0)$, `no` $= (\mathtt{no}, 0)$ and the set $\mathcal{I}$ of objects in membranes in the initial configuration. For

each vertex (o, h) in G there is an associated counter such that $c((o, h)) = Q_{M(h)}(o)$, it is it store the multiplicity of object type o in membrane h (note that due to the unique labels normal form, $H = V_\mu$ and L is the identity). The graph G combined with c represent the initial configuration of the system.

Then consider each membrane (h) in depth first order (w.r.t the membrane structure μ), we apply the rules in the maximal multiset for this membrane configuration in the following order:

- For each rule $(\mathrm{a}, o, h, \widetilde{m}) \in R$, subtract 1 from $c((o, h))$ and then for each $u \in \widetilde{m}$ add an edge to V from (o, h) to (u, h) and add 1 to $c((u, h))$.
- For each rule (b, o, h, u), add an edge from $(o, \mathrm{parent}(\mu, h))$ to (u, h), then subtract 1 from $c((o, \mathrm{parent}(\mu, h)))$ and add 1 to $c((u, h))$.
- For each rule (c, o, h, u), add an edge from (o, h) to $(u, \mathrm{parent}(\mu, h))$, then subtract 1 from $c((o, h))$ and add 1 to $c((u, \mathrm{parent}(\mu, h)))$.
- For each rule $(\mathrm{e}, o, h, \{u, v\})$, add an edge from (o, h) to (u, h) and then from (o, h) to (v, h), subtract 1 from $c((o, h))$, double all counters $c((x, h))$ for all $x \in O$, then add 1 to the counter $c((u, h))$ and add 1 to the counter $c((v, h))$. Add an edge from (o, h) to (u, h) and to (v, h) increasing their counters by 1.
- For each rule $(\mathrm{e_s}, o, h, \{u, u\})$, add an edge from (o, h) to (u, h), subtract 1 from $c((o, h))$, double all counters $c((x, h))$ for all $x \in O$, then add 2 to the counter $c((u, h))$.
- For each rule $(\mathrm{e_w}, o, h, \{u, v\})$, add an edge from (o, h) to (u, h) and then from (o, h) to (v, h), subtract 1 from $c((o, h))$, double the counters, $c((x, g))$ for all objects $x \in O$ in membranes $g \in \mathrm{subtree}(\mu, h)$. Add 1 to the counter $c((u, h))$ and to $c((v, h))$.
- For each rule $(\mathrm{f}, \{l, m\}, h, \{\})$, double the counters of $c((x, g))$ for all objects $x \in O$ and all membranes $g \in \mathrm{subtree}(\mu, h) \backslash (\mathrm{subtree}(\mu, m) \cup \mathrm{subtree}(\mu, l))$.

The graph now represents the configuration of the system Π after a single time-step. Now iterate the process until there are no more applicable rules, at this point the graph G and its associated counters c represent Π in a halting configuration.

If the membrane system has an accepting (rejecting) computation then there is a path from an object in $\mathcal{I}$ to vertex `yes` *(respectively* `no`*) in the graph G.* If there is a rule that evolves o to u and changes membrane from h to g then the above algorithm adds an edge to G from vertex (o, h) to (u, g). So if o in membrane h eventually evolves to u in membrane g then there is a path from vertex (o, h) to (u, g). Thus if the membrane system halts in an accepting (or rejecting) computation then there is a path in G from an object o in one of the initial object multisets ($(o, h) \in \mathcal{I}$), to `yes` (or `no`).

If there is a path in G from $\mathcal{I}$ to `yes` *(respectively* `no`*) then the membrane system has an accepting (rejecting) computation.* The algorithm described only adds an edge from one vertex (o, h) to another (u, g) to G if there is a rule that evolves o to u and

changes membrane from h to g. This means there is a path from one vertex (o, h) to another (u, g) only if o in membrane h eventually evolves to u in membrane g. If there is a path in G from an object o in one of the initial object multisets ($(o, h) \in \mathcal{I}$) to `yes` (or `no`), then the membrane system halts in an accepting (or rejecting) computation.

This property still holds when we add to G all rules in R that are not represented as edges in the graph. This does not add any new paths from $\mathcal{I}$ to `yes` or `no` since the new edges are from rules that are not reachable from any object in an initial membrane multiset.

We discard the counters c having established that existence of paths through the graph G to `yes` and `no` are equivalent to the membrane system having an accepting or rejecting computation. The graph G is now identical to the graph $\mathcal{G}$ for the membrane system Π. □

Chapter 3

Semi-uniform characterisations of NL and L without dissolution

In this chapter we present the first of our complexity results for membrane systems using uniformity conditions below P. First we re-examine an existing result using tighter uniformity conditions and then show a new way to adjust the computing power of membrane systems. Specifically we show that semi-uniform families of $\mathcal{AM}^0_{-d}$ systems,

- when using the (standard) recogniser definition, characterise NL (improving the previous P upper-bound [30]).
- when using a generalisation of the recogniser definition (that makes them easier to program), also characterise NL.
- when using a restriction of the recogniser definition, characterise L.

The first two results (which appear in Sections 3.1 and 3.2) hold for all semi-uniformity conditions computable in non-deterministic logspace while the third (found in Section 3.3) holds for those computable in deterministic logspace. However, if a semi-uniformity condition constructs a membrane system using more resources than are needed to evaluate the resulting membrane system, then the power of the semi-uniformity condition defines the problems that the family can solve. This effect is visualised in Figure 3.1.

In this chapter we introduce a new technique to prove our characterisations which takes advantage of the close similarity between semi-uniformity conditions and reductions. While it is known that a $\mathcal{AM}^0_{-d}$ membrane system can be represented as a dependency graph (see Section 2.6), we prove in Lemma 3.2 that each dependency graph can be converted in FAC^0 to an $\mathcal{AM}^0_{-d}$ membrane system of a very restricted form. We then define a reachability problem for dependency graphs such that there is a path between two vertices iff the corresponding membrane systems accept. By

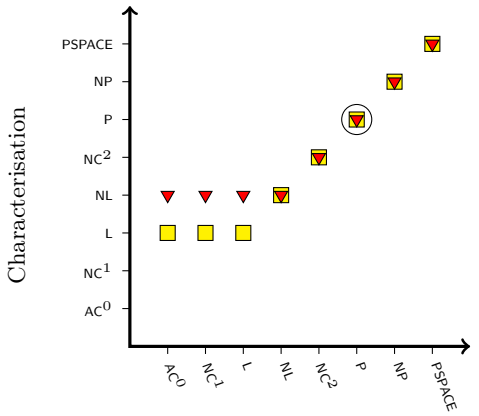

Figure 3.1: Characterisations by semi-uniform families of (general) recogniser $\mathcal{AM}^0_{-d}$ systems are denoted by ▼, those by semi-uniform families of restricted recogniser $\mathcal{AM}^0_{-d}$ systems are denoted by □. The P-semi-uniform P characterisation [29] is denoted ○.

giving a reduction from X, a complete problem for some class, to a dependency graph reachability problem and then converting the reachability problem to a membrane system we are providing a semi-uniform family to solve X.

We now observe that $\mathcal{AM}^0_{-d}$ systems can be significantly simplified.

Normal Form: 3.1. *An $\mathcal{AM}^0_{-d}$ system Π, is in* environmental normal form *if it (i) has no membranes other than the environment, (ii) and uses only rules of type* (a)*.*

Lemma: 3.2. *Given a dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \texttt{yes}, \texttt{no})$ we can construct in* FAC^0 *a recogniser active membrane system $\Pi_{\mathcal{G}}$ in environmental normal form that accepts iff there is a path from $\texttt{i} \in \mathcal{I}$ to $\texttt{yes}$ in $\mathcal{G}$.*

Proof. We let the vertices of the dependency graph be the objects of the membrane system. We then convert the edges of the dependency graph into object evolution rules.

The membrane system $\Pi_{\mathcal{G}}$: (i) uses a single membrane with label 0 and (ii) each edge in $\mathcal{G}$ becomes a rule of type (a).
The system $\Pi_{\mathcal{G}} = (O_{\mathcal{G}}, \mu_{\mathcal{G}}, M_{\mathcal{G}}, H_{\mathcal{G}}, L_{\mathcal{G}}, R_{\mathcal{G}})$ where

$O_{\mathcal{G}} = \mathcal{V}$,
$\mu_{\mathcal{G}} = (\{0\}, \emptyset)$,
$M_{\mathcal{G}} = \{(0, \{\texttt{i}, 1\}) \mid \texttt{i} \in \mathcal{I}\}$,
$H_{\mathcal{G}} = \{0\}$,
$L_{\mathcal{G}} = \{(0, 0)\}$,
$R_{\mathcal{G}} = \{(\mathrm{a}, v, 0, S(v)) \mid v \in \mathcal{V}\}$ where $S(v) = [s \in \mathcal{V} \mid (v, s) \in \mathcal{E}]$ (note that the multiset $S(v)$ is a set).

The mapping from $\mathcal{G}$ to $R_{\mathcal{G}}$ is surjective, there is no rule in $\Pi_{\mathcal{G}}$ that does not have a corresponding edge in $\mathcal{G}$. Thus it is the case that path$(\texttt{i}, \texttt{yes})$ in $\mathcal{G} \Leftrightarrow \texttt{i}$ eventually

evolves yes in membrane 0 of $\Pi_{\mathcal{G}}$ and that path(i, no) in $\mathcal{G} \Leftrightarrow i$ eventually evolves no in membrane 0 of $\Pi_{\mathcal{G}}$.

Apart from the function $S(x)$, this construction is more or less a direct mapping and is thus computable by a constant time CRAM. To compute the function $S(x)$ in constant time, a CRAM writes "(a, x, 0, ", to its output registers, this is the first part of the type (a) rule. There are $|\mathcal{V}|$ processors, each designated to read a single element of row x in the adjacency matrix A of $\mathcal{G}$. Each processor for row x writes out ", y" to their output register if the element $a_{x,y} = 1$ or the blank symbol "␣" if $a_{x,y} = 0$. A processor writes ")" after the $|\mathcal{G}_v|$ objects and blank symbols to end the rule. Thus the construction is in FAC^0. □

3.1 Recogniser membrane systems and NL

In this section we prove that semi-uniform families of recogniser $\mathcal{AM}^0_{-d}$ systems characterise NL. This is the "standard" definition of recogniser membrane systems which we now restate.

Definition: (Restatement of Definition 2.9)**.** *A* recognizer membrane system, Π, *is a membrane system such that:*

1. *all computations halt;*
2. yes, no $\in O$*;*
3. *the object* yes *or object* no *(but not both) appear in the multiset of the membrane with label* 0 *(the environment),*
4. *and this happens only in the halting configuration.*

If the yes object arrives in the membrane with label 0 the computation is accepting. If the no object arrives in the membrane with label 0 the computation is rejecting.

Points 3 and 4 from Definition 2.9 allow us to define the following subsets of the objects O in a $\mathcal{AM}^0_{-d}$ system. $O_{\texttt{yes}} = \{o \mid o \in O$ and o eventually evolves yes in the membrane with label $0\}$, $O_{\texttt{no}} = \{o \mid o \in O$ and o eventually evolves no in the membrane with label $0\}$, and $O_{\text{other}} = O \backslash (O_{\texttt{yes}} \cup O_{\texttt{no}})$.

Lemma: 3.3. *In recogniser* $\mathcal{AM}^0_{-d}$ *system* Π, $O_{\texttt{yes}} \cap O_{\texttt{no}} = \emptyset$.

Proof. Assume that object $o \in O_{\texttt{yes}} \cap O_{\texttt{no}}$, this implies that both a yes and a no object are produced by the system which contradicts point 3 of Definition 2.9. □

We prove that a dependency graph with the following properties can be converted to a standard recogniser $\mathcal{AM}^0_{-d}$ system as in Definition 2.9. A dependency graph satisfying these conditions is illustrated in Figure 3.2.

Lemma: 3.4. *A dependency graph* $\mathcal{G}$ *yields a* standard recogniser $\mathcal{AM}^0_{-d}$ *membrane system* $\Pi_{\mathcal{G}}$ *when converted by the method described in Lemma 3.2 if* $\mathcal{G}$ *has the following properties:*

1. *The graph $\mathcal{G}$ is acyclic.*
2. *The vertices* `yes` *and* `no` *are in $\mathcal{V}$.*
3. *The set $\mathcal{V}$ must contain the subsets $\mathcal{V}_{yes} = \{v \in \mathcal{V} \mid \text{path}(v, \texttt{yes})\}$, $\mathcal{V}_{no} = \{v \in \mathcal{V} \mid \text{path}(v, \texttt{no})\}$ such that $\mathcal{V}_{yes} \cap \mathcal{V}_{no} = \emptyset$ and $\texttt{i} \in \mathcal{V}_{yes} \cup \mathcal{V}_{no}$.*
4. *The path from vertex* `i` *to* `yes` *(or* `i` *to* `no`*) is the longest path in $\mathcal{G}$ starting at* `i`*.*

Proof. We show how each of the properties mentioned above ensures that a membrane system $\Pi_{\mathcal{G}}$ constructed from $\mathcal{G}$ using the technique in Lemma 3.2 is a recogniser membrane system as in Definition 2.9.

1. If the graph is acyclic then $\Pi_{\mathcal{G}}$ must halt, this satisfies point 1 from Definition 2.9.
2. Vertices `yes` and `no` correspond to the objects `yes` and `no` in membrane 0, this satisfies point 2 of Definition 2.9.
3. The subsets $\mathcal{V}_{yes}$ and $\mathcal{V}_{no}$ imply that the objects in $\Pi_{\mathcal{G}}$ can be divided into sets $O_{\texttt{yes}}$ and $O_{\texttt{no}}$ so that object `yes` or `no` but not both arrive in the membrane with label 0 in $\Pi_{\mathcal{G}}$. This satisfies point 3 of Definition 2.9.
4. If the path to vertex `yes` or vertex `no` is the longest in the graph then the corresponding object `yes` or object `no` will arrive in membrane 0 in the final step of the computation of $\Pi_{\mathcal{G}}$ which satisfies point 4 of Definition 2.9.

□

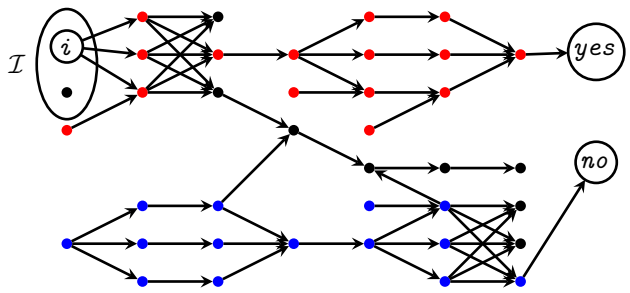

Figure 3.2: An example dependency graph $\mathcal{G}$ for some unspecified standard recogniser membrane system (Definition 2.9). Note that it satisfies all points of Lemma 3.4. For example, there are no directed paths from the vertices in $\mathcal{V}_{yes}$ (●) to the vertices of $\mathcal{V}_{no}$ (●).

We define the problem STDREC which is a reachability problem on the set of dependency graphs that have all the properties mentioned in Lemma 3.4.

Problem: 3.5 (STDREC)**.**
Instance: *A dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \{i\}, \texttt{yes}, \texttt{no})$ where $\texttt{i}, \texttt{yes}, \texttt{no} \in \mathcal{V}$, where $\mathcal{G}$ has all the properties mentioned in Lemma 3.4.*
Question: *Is there a directed path from* `i` *to* `yes`*?*

We state the problem STCON, the canonical NL-complete problem [35]. This problem is also commonly known as PATH, REACHABILITY, and GAP.

Problem: 3.6 (STCON)**.**
Instance: *A directed (possibly cyclic) graph* $G = (V, E, s, t)$ *where* $s, t \in V$*.*
Question: *Is there a directed path in* G *from* s *to* t*?*

The problem STCON is NL-complete for both cyclic and acyclic graphs.

Theorem: 3.7. STDREC *is* NL*-complete under* AC^0 *reductions.*

Before proving Theorem 3.7 we explain an important complication. If we wish to reduce G, an STCON instance, directly to $\mathcal{G}$, an instance of STDREC, it is important to ensure that path($\boldsymbol{i}$, $\boldsymbol{no}$) holds in $\mathcal{G}$ iff path(s, t) *does not hold* in G (which is needed to satisfy the requirements of Lemma 3.4). Our reduction simultaneously uses both an STCON instance and its complement problem coSTCON and then use both graphs to construct an instance of STDREC. To achieve this, one potential solution would be to take an instance of STCON and explicitly construct a complementary coSTCON instance via the construction in [31, 69]. Instead, we give a simpler proof via an NL-complete problem which already has the property we require. We use this third problem to prove Theorem 3.7.

Problem: 3.8 (coSTCON)**.**
Instance: *A directed (possibly acyclic) graph* $G' = (V', E', s', t')$ *where* $s', t' \in V'$ *.*
Question: *Is there no directed path in* G' *from* s' *to* t'*?*

Problem: 3.9 (STCON–coSTCON)**.**
Instance: *A directed (possibly acyclic) graph* $G = (V, V', E, E', s, s', t, t')$ *with two disjoint components* (V, E) *and* (V', E') *and where* $s, t \in V$ *and* $s', t' \in V'$ *where there is a path from* s' *to* t' *iff there is no path from* s *to* t*.*
Question: *Is there a path in* G *from* s *to* t*?*

Lemma: 3.10. *Both acyclic and cyclic* STCON–coSTCON *are* AC^0 *complete for* NL*.*

Proof. The problem STCON is AC^0 complete for NL and so all problems in NL are AC^0 reducible to STCON. Likewise, coSTCON is AC^0 complete for coNL and so all problems in coNL are AC^0 reducible to coSTCON. Immerman and Szelepcsényi showed that NL = coNL [31, 69], this immediately implies that that STCON is AC^0 reducible to coSTCON. So given an instance of STCON we can reduce it to coSTCON and combine the two problem instances into a third NL-complete problem called STCON–coSTCON.

An instance of (acyclic) STCON–coSTCON is trivially in NL. □

We are now ready to show acyclic STCON–coSTCON $\leq_{\mathsf{AC}^0}$ STDREC.

Proof. Given an instance $(V, V', E, E', s, s', t, t')$ of acyclic STCON–coSTCON, we construct a dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{i}, \boldsymbol{yes}, \boldsymbol{no})$. To ensure $\mathcal{G}$ has Property 4 in Lemma 3.4, that $\boldsymbol{yes}$ and $\boldsymbol{no}$ are at the end of the longest paths in the graph, we add

a directed path of $|V| + |V'| + 1$ edges leading from t to $\mathtt{yes}$ and from t' to $\mathtt{no}$.

$$\mathcal{V}_{\mathrm{y}} = \{v_1, \ldots, v_{|V|+|V'|}\}$$
$$\mathcal{V}_{\mathrm{n}} = \{v'_1, \ldots, v'_{|V|+|V'|}\}$$
$$\mathcal{E}_{\mathrm{y}} = \{(t, v_1)\} \cup \{(v_i, v_{i+1}) \mid i \in \{1, \ldots, |V| + |V'| - 1\}\} \cup \{(v_{|V|+|V'|}, \mathtt{yes})\}$$
$$\mathcal{E}_{\mathrm{n}} = \{(t', v'_1)\} \cup \{(v'_i, v'_{i+1}) \mid i \in \{1, \ldots, |V| + |V'| - 1\}\} \cup \{(v'_{|V|+|V'|}, \mathtt{no})\}$$

So the final dependency graph $\mathcal{G}$ has $\mathcal{V} = V \cup V' \cup \{\mathtt{i}, \mathtt{yes}, \mathtt{no}\} \cup \mathcal{V}_{\mathrm{y}} \cup \mathcal{V}_{\mathrm{n}}$ and $\mathcal{E} = E \cup E' \cup \mathcal{E}_{\mathrm{y}} \cup \mathcal{E}_{\mathrm{n}} \cup \{(\mathtt{i}, s), (\mathtt{i}, s')\}$.

To ensure $\mathcal{G}$ has Property 1 of Lemma 3.4 we consider only acyclic instances of STCON–coSTCON (note the paths $\mathcal{E}_{\mathrm{y}}$ and $\mathcal{E}_{\mathrm{n}}$ are acyclic). We see that $\mathcal{G}$ has Properties 3 and 2 from Lemma 3.4 because $\mathcal{V}_{\mathtt{yes}} \cap \mathcal{V}_{\mathtt{no}} = \emptyset$ and $\mathtt{i} \in \mathcal{V}_{\mathtt{yes}} \cup \mathcal{V}_{\mathtt{no}}$ since (V, E) and (V', E') are disjoint and only one of path$(s, \mathtt{yes})$ and path$(s', \mathtt{no})$ holds, thus $\mathtt{i}$ (which is connected to s and s') can be in $\mathcal{V}_{\mathtt{yes}}$ or $\mathcal{V}_{\mathtt{no}}$ but never both.

Since our construction does not modify the input graphs and we connect $\mathtt{i}$ to s and s' it is clear

- that path$(\mathtt{i}, \mathtt{yes})$ in $\mathcal{G}$ iff path(s, t) holds in graph G, and
- that path$(\mathtt{i}, \mathtt{no})$ in $\mathcal{G}$ iff path(s, t) does not hold in graph G.

This reduction is straightforward to compute in constant time with a CRAM, the extra edges and vertices added to the graph are output by two sets of $|V| + |V'| + 1$ processors that each write out an edge based on their processor number. The rest of the reduction is straightforward and is computable in constant time by a CRAM and thus is computable in FAC^0.

We show STDREC $\leq_{\mathsf{AC}^0}$ STCON. Given an instance $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathtt{i}, \mathtt{yes}, \mathtt{no})$ of STDREC, we construct $G = (V, E, s, t)$ such that $V = \mathcal{V}$ and $E = \mathcal{E}$ and let $s = \mathtt{i}$ and $t = \mathtt{yes}$. Clearly there is a path from s to t in G iff there is a path from $\mathtt{i}$ to $\mathtt{yes}$ in the dependency graph $\mathcal{G}$. This reduction is computable in constant time by a CRAM, and thus in FAC^0. □

Theorem: 3.11. (C)–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}} = \mathsf{NL}$ *using the* standard *recogniser conditions from Definition 2.9 for all* $\mathsf{C} \in \{\mathsf{AC}^0, \mathsf{NC}^1, \mathsf{L}, \mathsf{NL}\}$.

Proof. Given an acyclic instance x of STCON–coSTCON an NL-complete problem, we reduce it in FAC^0 (as in Theorem 3.7) to an instance of STDREC which satisfies all points in Lemma 3.4. The instance $\mathcal{G}$ of STDREC is convertible in FAC^0 (using the technique in Lemma 3.2) to a recogniser membrane system $\Pi_{\mathcal{G}}$ which accepts iff $x \in$ STCON–coSTCON and rejects otherwise. These two FAC^0 algorithms, applied in sequence, describe a semi-uniform family of recogniser $\mathcal{AM}^0_{-d}$ systems that solves an NL-complete problem, hence $\mathsf{NL} \subseteq (\mathsf{AC}^0)$–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}}$. If we increase the power of the reductions to classes mentioned in the statement, there is no increase in the set of problems solvable, that is STCON–coSTCON $\leq_{\mathsf{NL}}$ STDREC.

Now we show that a semi-uniform family of recogniser $\mathcal{AM}^0_{-d}$ systems can recognise no more than NL. We have seen that any $\mathcal{AM}^0_{-d}$ recogniser membrane system Π can be converted to an instance of STDREC by an FAC^0 algorithm via Lemma 2.22. We

then showed in Theorem 3.7 that any instance of STDREC can be reduced in FAC^0 to an instance of STCON. This implies that a non-deterministic logspace Turing machine can decide if a recogniser $\mathcal{AM}^0_{-d}$ accepts by first converting it to an instance of STCON, hence (AC^0)–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}} \subseteq \mathsf{NL}$. If the semi-uniformity condition is computable in one of the classes mentioned in the statement, there is no increase in the complexity of the problem since the reductions are all contained in FNL. □

3.2 General recogniser systems and NL

We introduce a generalisation of the standard definition of recogniser membrane systems. With this more general definition it is possible for a membrane system computation to have both `yes` and `no` objects in the environment. However, the first of these objects to arrive in the environment determines if the computation is accepting or rejecting. (Note that it is forbidden for both `yes` and `no` objects to arrive for the first time in the environment in the same time-step.) We now define *general recogniser membrane systems* and then show that the problems solved by semi-uniform families of general recogniser $\mathcal{AM}^0_{-d}$ systems is exactly the class NL. This characterisation shows that general recogniser membrane systems (for semi-uniform families of $\mathcal{AM}^0_{-d}$ systems) have equal power to the standard recogniser systems discussed in Section 3.1.

Definition: 3.12. *A* general recogniser membrane system Π *is a membrane system such that:*

1. $\mathtt{yes}, \mathtt{no} \in O$*;*
2. *the object* `yes` *and the object* `no` *may not arrive in the membrane with label* 0 *(the environment) for the first time, in the same time-step.*

If the `yes` object arrives in the membrane with label 0 before object `no`, the computation is accepting. If the `no` object arrives in the membrane with label 0 before object `yes`, the computation is rejecting. Note that computations may be infinite.

We prove that a dependency graph with the following properties can be converted to a general recogniser $\mathcal{AM}^0_{-d}$ system as in Definition 3.12. A dependency graph satisfying these conditions is illustrated in Figure 3.3.

Lemma: 3.13. *A dependency graph* $\mathcal{G}$ *yields a* general recogniser $\mathcal{AM}^0_{-d}$ *membrane system* $\Pi_{\mathcal{G}}$ *when converted by the method described in Lemma 3.2 if* $\mathcal{G}$ *has the following properties:*

1. *The vertices* `yes` *and* `no` *are in* $\mathcal{V}$*.*
2. *The length of the shortest path from* `i` *to* `yes` *is not equal to the length of the shortest path from* `i` *to* `no`*.*

Proof. We show how each of the properties mentioned above ensures that a membrane system $\Pi_{\mathcal{G}}$ (constructed from $\mathcal{G}$ using the technique in Lemma 3.2) is a general recogniser membrane system as in Definition 3.12.

1. Vertices *yes* and *no* correspond to objects yes and no in membrane 0, this satisfies point 1 of Definition 3.12.

2. If the length of the shortest path from *i* to *yes* is not equal to the length of the shortest path from *i* to *no* then in a membrane system represented by this graph, object yes or no arrive in membrane with label 0, but not at the same time, satisfying point 2 of Definition 3.12.

□

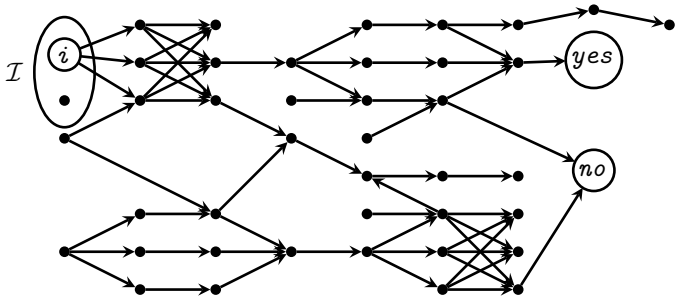

Figure 3.3: An example dependency graph $\mathcal{G}$ for some unspecified *general recogniser membrane system* (Definition 3.12). Note that it satisfies all points of Lemma 3.13. This example represents a rejecting computation since the minimum directed path from *i* to no is of length 6, while the minimum directed path from an element of $\mathcal{I}$ to yes is of length 7.

We define the problem GENREC which is a reachability problem on the set of dependency graphs that have all the properties mentioned in Lemma 3.13.

Problem: 3.14 (GENREC)**.**
Instance: *A dependency graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \{i\}, \texttt{yes}, \texttt{no})$ *where* $i, \texttt{yes}, \texttt{no} \in \mathcal{V}$*, where* $\mathcal{G}$ *has all the properties mentioned in Lemma 3.13.*
Question: *Is the length of the shortest directed path from* *i* *to* *yes* *less than the length of the shortest directed path from* *i* *to* *no*?*

Theorem: 3.15. GENREC *is* NL*-complete under* AC^0 *reductions.*

Proof. First we show $\mathsf{STCON} \leq_{\mathsf{AC}^0} \mathsf{GENREC}$. Given an instance $G = (V, E, s, t)$ of STCON, we construct a dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, i, \texttt{yes}, \texttt{no})$. Let $i = s$. The set of vertices is $\mathcal{V} = V \cup \{\texttt{yes}, \texttt{no}\} \cup \{v_1, \ldots, v_{|V|}\}$. We construct the set of edges such that $\mathcal{E} = E \cup \{(t, \texttt{yes}), (s, v_1), (v_{|V|}, \texttt{no})\} \cup \{(v_i, v_{i+1}) \mid i \in \{1, \ldots, |V| - 1\}\}$. Clearly there is a path from *i* to *yes* iff there is a path from s to t in $\mathcal{G}$. By adding a path of length $|V| + 1$ edges from *i* to *no* we are providing a path longer than any (acyclic) path to *yes* (satisfying point 2 of Observation 3.13). This reduction is computable by a constant time CRAM, the extra edges and vertices added to the graph are output by $|V| + 1$ processors that each write out an edge based on their processor number. Thus this reduction is computable in FAC^0.

We now show that GENREC $\in$ NL, by providing an algorithm that is computable by a Turing machine storing a single binary counter x.

let $x = 0$
for x from 0 to $|\mathcal{V}|$
  if there exists a path from *i* to *yes* of length x **then**
    accept
  else if there exists a path from *i* to *no* of length x **then**
    reject
  end if
end for

Deciding if there is a path of length x between two vertices is computed with the standard STCON algorithm [35] with an added line that rejects if length not equal to x. This Turing machine uses a non-deterministic algorithm and a single binary counter to decide GENREC and so the problem is in NL. □

Theorem: 3.16. (C)–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}} = \mathsf{NL}$ *using the* general *recogniser conditions from Definition 3.12 for all* $\mathsf{C} \in \{\mathsf{AC}^0, \mathsf{NC}^1, \mathsf{L}, \mathsf{NL}\}$.

Proof. Given an instance x of STCON, an NL-complete problem, we reduce it in FAC^0 (as in Theorem 3.15) to an instance of GENREC which satisfies all points in Lemma 3.13. The instance $\mathcal{G}$ of GENREC is convertible in FAC^0 (using the technique in Lemma 3.2) to a general recogniser membrane system $\Pi_{\mathcal{G}}$ which accepts iff $x \in$ STCON and rejects otherwise. These two FAC^0 computable algorithms, applied in sequence, describe a semi-uniform family of general recogniser $\mathcal{AM}^0_{-d}$ systems that solves an NL-complete problem, hence $\mathsf{NL} \subseteq (\mathsf{AC}^0)$–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}}$. If we increase the power of the reductions to classes mentioned in the statement, there is no increase in the set of problems solvable, that is STCON $\leq_{\mathsf{NL}}$ GENREC.

Now we show that a semi-uniform family of general recogniser $\mathcal{AM}^0_{-d}$ systems can recognise no more than NL. We have seen that any $\mathcal{AM}^0_{-d}$ general recogniser membrane system Π can be converted to an instance of GENREC by an FAC^0 algorithm via Lemma 2.22. We then showed via Theorem 3.15 that any instance of GENREC is decided by a logspace non-deterministic Turing machine, hence (AC^0)–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}} \subseteq \mathsf{NL}$. If the semi-uniformity condition is computable in one of the classes mentioned in the statement, there is no increase in complexity of the problem since the reductions are all contained in NL. □

3.3 Restricted recogniser systems and L

We now consider a restriction on the standard definition of recogniser membrane systems. In Definition 2.9 it is forbidden for an object that eventually evolves a `yes` to also yield a `no` (and vice versa). Now we further restrict the system so that all objects in the system must eventually evolve either a `yes` or `no` object. Notice that this restriction forbids objects that do not contribute to the final answer (accept or reject) and forbids rules of the form $[a \to \lambda]$ where λ is the empty word. We now

define this acceptance condition and then go on to show that the class of problems solved by semi-uniform families of such restricted recogniser $\mathcal{AM}^0_{-d}$ systems is exactly the class L.

Definition: 3.17. *A* restricted recogniser membrane system, Π, *is a membrane system such that:*

1. *all computations halt;*
2. $\texttt{yes}, \texttt{no} \in O$*;*
3. *the object* yes *or object* no *(but not both) arrive in the membrane with label* 0 *(the environment);*
4. *and this happens only in the halting configuration;*
5. *each* $o \in O$ *must, via a sequence of zero or more developmental rules, eventually evolve either* yes *or* no *in the membrane with label* 0*, but not both.*

If the yes object arrives in the membrane with label 0 the computation is accepting. If the no object arrives in the membrane with label 0 the computation is rejecting.

Points 3 and 4 from Definition 3.17 allow us to define the following subsets of the objects O in a $\mathcal{AM}^0_{-d}$ system. $O_{\texttt{yes}} = \{o \mid o \in O$ and o eventually evolves yes in the membrane with label $0\}$, $O_{\texttt{no}} = \{o \mid o \in O$ and o eventually evolves no in the membrane with label $0\}$. Since this is a restriction of the standard definition, we inherit Lemma 3.3 which states that $O_{\texttt{yes}} \cap O_{\texttt{no}} = \emptyset$. We also have the following consequence of the restriction.

Lemma: 3.18. *In restricted recogniser* $\mathcal{AM}^0_{-d}$ *system* Π, $O \backslash (O_{\texttt{yes}} \cup O_{\texttt{no}}) = \emptyset$.

Proof. Assume that object $o \in O$ is such that $o \notin O_{\texttt{yes}} \cup O_{\texttt{no}}$, this implies that o does not eventually evolve a yes nor a no object in the membrane with label 0. This contradicts point 5 of Definition 3.17. □

We prove that a dependency graph with the following properties can be converted to a restricted recogniser $\mathcal{AM}^0_{-d}$ system as in Definition 3.17. A dependency graph satisfying these conditions is illustrated in Figure 3.4.

Lemma: 3.19. *A dependency graph* $\mathcal{G}$ *yields a* restricted recogniser $\mathcal{AM}^0_{-d}$ *membrane system* $\Pi_{\mathcal{G}}$ *when converted by the method described in Lemma 3.2 if* $\mathcal{G}$ *has the following properties:*

1. *The graph* $\mathcal{G}$ *is acyclic.*
2. *The vertices* $\texttt{yes}$ *and* $\texttt{no}$ *are in* $\mathcal{V}$*.*
3. *The set* $\mathcal{V}$ *must contain the subsets* $\mathcal{V}_{yes} = \{v \in \mathcal{V} \mid \text{path}(v, \texttt{yes})\}$, $\mathcal{V}_{no} = \{v \in \mathcal{V} \mid \text{path}(v, \texttt{no})\}$ *such that* $\mathcal{V}_{yes} \cap \mathcal{V}_{no} = \emptyset$ *(see Lemma 3.3) and* $\texttt{i} \in \mathcal{V}_{yes}$ *or* $\texttt{i} \in \mathcal{V}_{no}$*.*
4. *The path from vertex* $\texttt{i}$ *to* $\texttt{yes}$ *or* $\texttt{no}$ *is the longest in* $\mathcal{G}$ *starting at* $\texttt{i}$*.*

5. $\mathcal{V} \backslash \mathcal{G}_{yes} \cup \mathcal{V}_{no} = \emptyset$ *(see Lemma 3.18).*

Proof. We show how each of the properties mentioned above ensures that a membrane system $\Pi_{\mathcal{G}}$, constructed from $\mathcal{G}$ using the technique in Lemma 3.2, is a restricted recogniser membrane system as in Definition 3.17.

1. If the graph is acyclic then $\Pi_{\mathcal{G}}$ must halt, this satisfies point 1 from Definition 3.17.

2. Vertices yes and no correspond to the objects `yes` and `no` in membrane 0, this satisfies point 2 of Definition 3.17.

3. The subsets $\mathcal{V}_{yes}$ and $\mathcal{V}_{no}$ imply that the objects in $\Pi_{\mathcal{G}}$ can be divided into sets $O_{\texttt{yes}}$ and $O_{\texttt{no}}$ so that every object in $\Pi_{\mathcal{G}}$ must eventually evolve `yes` or `no` but not both. This satisfies point 3 of Definition 3.17.

4. If the path to vertex yes or vertex no is the longest in the graph then the corresponding object `yes` or object `no` will arrive in membrane 0 in the final step of the computation of $\Pi_{\mathcal{G}}$ which satisfies point 4 of Definition 3.17.

5. This ensures that every object in the system evolves to either a `yes` or a `no` object satisfying point 5 of Definition 3.17.

□

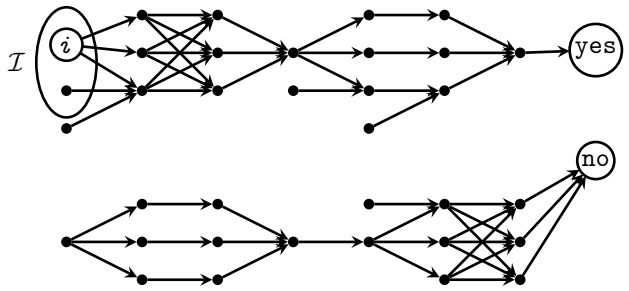

Figure 3.4: An example dependency graph $\mathcal{G}$ for some unspecified restricted recogniser membrane system (Definition 3.17). Note that the graph consists of exactly two disjoint components.

We define the problem RSTREC which is a reachability problem on the set of dependency graphs that have all the properties mentioned in Lemma 3.19.

Problem: 3.20 (RSTREC)**.**
Instance: *A dependency graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \{i\}, yes, no)$ *where* $i, yes, no \in \mathcal{V}$*, where* $\mathcal{G}$ *has all the properties mentioned in Lemma 3.19.*
Question: *Is there a directed path from* i *to* yes*?*

We state the L-complete problem Directed Forest Accessibility (DFA) [19].

Problem: 3.21 (Directed Forest Accessibility (DFA) [19])**.**
Instance: *An acyclic directed graph* $G = (V, E, s, t)$ *where* $s, t \in V$ *and each vertex has out-degree of* 0 *or* 1*.*
Question: *Is there a directed path from* s *to* t*?*

Theorem: 3.22. RSTREC *is* L*-complete under* AC^0 *reductions.*

Proof. First we show $\mathsf{DFA} \leq_{\mathsf{AC}^0} \mathsf{RSTREC}$.

Given an instance $G = (V, E, s, t)$ of DFA, we construct a dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{i}, \boldsymbol{yes}, \boldsymbol{no})$. We let $\boldsymbol{i} = s$ and the set of vertices $\mathcal{V} = V \cup \{\boldsymbol{yes}, \boldsymbol{no}\}$. The edges of the dependency graph are $\mathcal{E} = E \backslash \{(t, v) \mid v \in V\} \cup \{(t, \boldsymbol{yes})\} \cup \{(v, \boldsymbol{no}) \mid v \in V \text{ and } v \text{ has out-degree } 0\}$.

To ensure that the vertex $\boldsymbol{yes}$ is at the end of every path that leads to t, $\mathcal{E}$ has edge from t to $\boldsymbol{yes}$ and does not include edges from E that leave t. $\mathcal{G}$ has Property 1 from Lemma 3.19 since G (as a forest) is acyclic, our reduction ensures $\mathcal{G}$ is acyclic also. To ensure $\mathcal{G}$ has property 3 of Lemma 3.19 we add extra edges to $\mathcal{E}$ connecting those vertices in $\mathcal{G}$ that cannot reach t (vertices with out-degree 0) to $\boldsymbol{no}$. Since all vertices in G with out-degree 0 (except t) now point to $\boldsymbol{no}$ there cannot exist a vertex that is not on a path to $\boldsymbol{yes}$ or $\boldsymbol{no}$ satisfying point 5 in Lemma 3.19. This, combined with the edge from t to $\boldsymbol{yes}$ gives $\mathcal{G}$ Property 4 of Lemma 3.19 by ensuring that only the vertices $\boldsymbol{yes}$ and $\boldsymbol{no}$ have out-degree 0. Clearly there is a path from $\boldsymbol{i}$ to $\boldsymbol{yes}$ in $\mathcal{G}$ iff there is a path from s to t in graph G.

We now explain how a CRAM writes an edge for every vertex with out-degree 0 in constant time. There is a processor for each element $a_{x,y}$ of the adjacency matrix of the graph G and a shared register $r_x = 0$ for each $x \in V$. In the first step these processors read their matrix element, if $a_{x,y} = 1$ they write "1" to register r_x. In the next time-step a processor for each $x \in V$ reads the value of r_x. If $r_x = 0$ then there was no edge leaving x in G and the processor writes an edge from $(x, \boldsymbol{no})$ in the set $\mathcal{E}$ to its output register. The rest of the reduction is a straightforward mapping and easily shown to be computable by a CRAM in constant time. Thus this reduction can be computed in FAC^0.

Now we show RSTREC is contained in L by outlining a deterministic logspace Turing machine algorithm that decides RSTREC. The input to the algorithm is an instance $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{i}, \boldsymbol{yes}, \boldsymbol{no})$ of RSTREC.

```
let x = i
for each (a, b) in E
  if a = x then
    if b = yes then accept
    else if b = no then reject
    else x = b
  end if
end for
```

Starting with the input vertex $\boldsymbol{i}$, the algorithm follows a path through the graph, storing only its current position in a variable. If it reaches the vertex $\boldsymbol{no}$ the al-

gorithm rejects, if it reaches ***yes*** the algorithm accepts. The algorithm correctly decides RSTREC since no vertex in $\mathcal{G}$ that leads to ***yes*** can also be on a path to ***no*** (non-***yes*** non-***no*** sink vertices are forbidden). Thus, it does not matter which particular edge is followed leaving a vertex, all lead to the correct answer. Since only one variable of length $|\mathcal{V}|$ is stored at any time, the algorithm uses $O(\log n)$ space (where $n = |\mathcal{V}|$ is the input length). Thus RSTREC $\in$ L. □

Theorem: 3.23. (C)–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}} = \mathsf{L}$ *using the* restricted *recogniser conditions from Definition 3.17 for all* $\mathsf{C} \in \{\mathsf{AC}^0, \mathsf{NC}^1, \mathsf{L}\}$.

Proof. Given an instance x of DFA, an L-complete problem, we reduce it in FAC^0 (as in Theorem 3.22) to an instance of RSTREC which satisfies all points in Lemma 3.19. The instance $\mathcal{G}$ of RSTREC is convertible in FAC^0 (using the technique in Lemma 3.2) to a restricted recogniser membrane system $\Pi_{\mathcal{G}}$ which accepts iff $x \in$ DFA and rejects otherwise. These two algorithms in FAC^0, applied in sequence, describe a semi-uniform family of restricted recogniser $\mathcal{AM}^0_{-d}$ systems that solves an NL-complete problem, hence $\mathsf{NL} \subseteq (\mathsf{AC}^0)$–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}}$. If we increase the power of the reductions to classes mentioned in the statement, there is no increase in the set of problems solvable, that is DFA $\leq_{\mathsf{L}}$ RSTREC.

Now we show that a semi-uniform family of restricted recogniser $\mathcal{AM}^0_{-d}$ systems can recognise no more than L. We have seen that any $\mathcal{AM}^0_{-d}$ restricted recogniser membrane system Π can be converted to an instance of RSTREC by an algorithm in FAC^0 algorithm via Lemma 2.22. We then showed via Theorem 3.22 that any instance of RSTREC decided by a logspace deterministic Turing machine, hence (AC^0)–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}} \subseteq \mathsf{L}$. If the semi-uniformity condition is computable in one of the classes mentioned in the statement, there is no increase in complexity of the problem since the reductions are all contained in L □

3.4 Discussion

In this chapter we have shown three characterisations that were made possible by using uniformity conditions below P. The first (in Section 3.1) improves the existing [30] P upper-bound for semi-uniform families of recogniser $\mathcal{AM}^0_{-d}$ systems to an NL characterisation.

The result in [30] claims a P characterisation for semi-uniform families of $\mathcal{AM}^0_{-d}$ systems, however these families "solve" P-complete problems by using the semi-uniformity condition to solve it for them. Thus, this P lower-bound is dependant on the semi-uniformity condition being P-hard. We claim that our NL characterisation is a more truthful one for semi-uniform $\mathcal{AM}^0_{-d}$ systems since it is robust under uniformity conditions computable in a range of classes in FNL (see Figure 3.1). Thus we see in this chapter that choosing an appropriate uniformity condition is crucial to analysing the true power of a membrane system.

It is curious that a system that can generate an exponential number of objects and membranes is limited to solving problems in non-deterministic logspace. It becomes more obvious why this is the case when we recall that such systems can be represented

using a single membrane with only a polynomial increase in the number of objects (see Normal Form 3.1). Moreover, the technique of using dependency graphs to predict $\mathcal{AM}^0_{-d}$ systems implies object multiplicities are not necessary for $\mathcal{AM}^0_{-d}$ systems to recognise a language. Thus, we can replace multisets with sets in the active membrane system definition and the class (AC^0)–$\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}}$ remains unchanged.

In Section 3.2 we introduced general recogniser membrane systems, these systems are simpler to program since the restrictions on a system (i) to produce only the output object `yes` or the output object `no`, and (ii) only in the final step of the computation, are relaxed. Despite this, we have shown that semi-uniform families of general $\mathcal{AM}^0_{-d}$ systems also characterise NL. Furthermore, our results prove the existence of a "complier" which, via reductions, translates a system that uses the general definition into a system that uses the standard definition. This is significant since we can more easily program a general recogniser system, then, convert it to a standard recogniser for which it is often easier to prove certain properties such as correctness.

This chapter also showed that it is possible to change the complexity of a membrane system by adjusting the definition of a valid computation and not just by varying the type of rules allowed. We were inspired by how the complexity of STCON drops from NL to L when the graph has a restricted shape. We added a restriction to the recogniser framework that forces the dependency graphs of $\mathcal{AM}^0_{-d}$ systems to have the same property. The class of problems solvable by semi-uniform families of $\mathcal{AM}^0_{-d}$ systems using this restricted definition correspondingly dropped to L. This technique could prove very useful to when trying to characterise other complexity classes such as P and PSPACE.

Finally, the techniques we have employed in this chapter reveal that semi-uniformity is closely related to the notion of reductions. Thus, similarly to how a reduction can turn a complicated problem into a simpler one, a semi-uniformity condition has the potential to disguise the true power of a membrane system. For example, a P-semi-uniform family of standard $\mathcal{AM}^0_{-d}$ systems can solve P complete problems, this is clear when we express it in terms of reductions: $\mathsf{AGAP} \leq_{\mathsf{P}} \mathsf{STDREC}$.

Chapter 4

Uniformity is strictly weaker than semi-uniformity

For many naturally inspired models of computation it often easier to design hard coded devices than a general problem solver. Such models map a specific *instance* of the problem to a computing device, we call this mapping *semi-uniformity*. In contrast, the well-established framework of circuit uniformity maps each input length $n \in \mathbb{N}$ to a circuit $c_n \in C$. This raises the question of whether the notions of uniformity and semi-uniformity are equivalent.

It has been shown in a number of models that whether one chooses to use uniformity or semi-uniformity does not affect the power of the model. However, in this chapter we show that these notions are not equivalent. We prove that choosing one notion over another gives characterisations of completely different complexity classes, including known distinct classes.

Why is this result surprising? Every class of problems solved by a uniform family of devices is contained in the analogous semi-uniform class, since one is a restriction of the other. However, in all membrane system models studied to date, the classes of problems solved by semi-uniform and uniform families turned out to be equal [7, 44, 66]. Specifically, if we want to solve some problem, by specifying a family of membrane systems (or some other model), it is often much easier to first use the more general notion of semi-uniformity, and then subsequently try to find a uniform solution. In almost all cases where a semi-uniform family was given for some problem [5, 45, 55, 66], at a later point a uniform version of the same result was published [4, 7, 55]. Here we prove that this improvement is not always possible.

In this chapter we show that uniformity is not equal to semi-uniformity, resolving Open Problem C in [56]. We show that the class of problems decidable by AC^0-*uniform* families of active membrane systems without charges or dissolution rules is a strict subset of the problems solvable by AC^0-*semi-uniform* families of systems of the same type. Besides their respective use of uniformity and semi-uniformity, both models are identical. Specifically, we show in Section 4.1 that AC^0-uniform $\mathcal{AM}^0_{-d}$ systems characterise AC^0. Combined with Theorem 3.11, which showed that AC^0-semi-

uniform $\mathcal{AM}^0_{-d}$ systems characterise NL, and the fact that $\mathsf{AC}^0 \subsetneq \mathsf{NL}$ [24], this gives the following result:

Theorem: 4.1. $\mathsf{AC}^0 = (\mathsf{AC}^0, \mathsf{AC}^0)\text{–}\mathsf{PMC}_{\mathcal{AM}^0_{-d}} \subsetneq (\mathsf{AC}^0)\text{–}\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}} = \mathsf{NL}$

This result is illustrated by the leftmost pair of triangles In Figure 4.1. In rest of this chapter, we prove the left hand side equality of Theorem 4.1, that is, that $(\mathsf{AC}^0, \mathsf{AC}^0)$-uniform recogniser $\mathcal{AM}^0_{-d}$ systems characterise AC^0. In fact we can also state a more general result for a number of complexity classes below NL, for brevity we keep the list short.

Theorem: 4.2. *Let* $\mathsf{C} \in \{\mathsf{AC}^0, \mathsf{NC}^1, \mathsf{L}\}$ *and assuming* $\mathsf{NC}^1 \subsetneq \mathsf{L} \subsetneq \mathsf{NL}$ *then* $\mathsf{C} = (\mathsf{C}, \mathsf{C})\text{–}\mathsf{PMC}_{\mathcal{AM}^0_{-d}} \subsetneq (\mathsf{C})\text{–}\mathsf{PMC}^*_{\mathcal{AM}^0_{-d}} = \mathsf{NL}$

This shows that such uniform membrane systems are essentially powerless compared to their uniformity conditions, and as far down as AC^0, they are as weak and as strong as their uniformity condition. In Figure 4.1, Theorem 4.2 is illustrated by the triangles to the left of (and including) the uniformity condition L.

The proof works by showing that in a uniform family of such membrane systems, even though predicting an arbitrary membrane system may be NL-complete, in fact there is an equivalent, but simpler, membrane system that can evaluated in AC^0. This, along with some other tools, is used to show that if the power of the uniformity notion is AC^0 or more, then the power of the entire family of systems is determined by the power of the encoding function.

This result proves something general about families of finite devices that is independent of particular formalisms and can be applied to other computational models besides membrane systems. To demonstrate this, in Section 4.2 we relate the notion of semi-uniformity to circuit complexity and obtain an analogous result with a shorter proof. Besides membrane systems and circuits, some other models that use notions of uniformity and semi-uniformity include families of neural networks, molecular and DNA computers, tile assembly systems, branching programs and cellular automata [9, 16, 49, 64, 65]. Our results could conceivably be applied to these models.

4.1 Uniform families without dissolution

The following theorem is key to the proof of Theorems 4.1 and 4.2. Roughly speaking, Theorem 4.3 states that in uniform membrane systems of the type we consider, the uniformity condition dominates the computational power of the system. By letting $\mathsf{E} = \mathsf{F} = \mathsf{AC}^0$, the statement of Theorem 4.3 gives us the left hand side equality in Theorem 4.1. By letting $\mathsf{E} = \mathsf{F} \in \{\mathsf{AC}^0, \mathsf{NC}^1, \mathsf{L}\}$ we get the left hand side of Theorem 4.2. The remaining classes quoted in the theorem serve to illustrate Figure 4.1.

Theorem: 4.3. *Let* $\mathsf{E}, \mathsf{F} \in \{\mathsf{AC}^0, \mathsf{NC}^1, \mathsf{L}, \mathsf{NL}, \mathsf{NC}^2, \mathsf{P}, \mathsf{NP}, \mathsf{PSPACE}\}$ *and let* $\mathsf{F} \subseteq \mathsf{E}$. *Then* $(\mathsf{E}, \mathsf{F})\text{–}\mathsf{PMC}_{\mathcal{AM}^0_{-d}} = \mathsf{E}$.

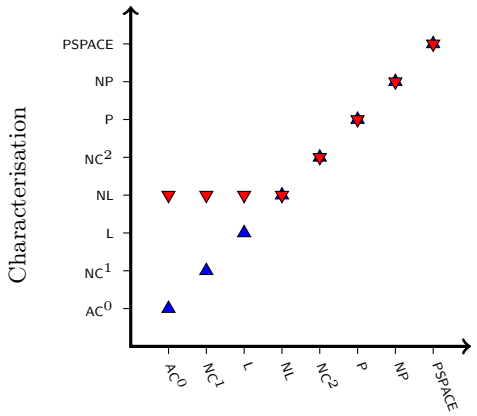

Figure 4.1: Complexity classes that are characterised by the membrane systems studied in this chapter. Characterisations by uniform systems are denoted by ▲, and semi-uniform by ▼. For example, Theorem 4.1 is illustrated by the fact that AC^0-uniform systems characterise AC^0, and that AC^0-semi-uniform systems characterise NL.

The proof works by showing that for each uniform membrane family $\mathbf{\Pi}$ (of polynomial sized recogniser $\mathcal{AM}^0_{-d}$ systems), that decides problem X, there exists another family $\mathbf{\Pi}'$ that decides X but where every system in $\mathbf{\Pi}'$ is so greatly simplified that $\mathbf{\Pi}'$ and can be evaluated by a uniform family of AC^0 circuits.

First we recall from Lemma 3.3 that the objects in a $\mathcal{AM}^0_{-d}$ system can be divided into two sets $O_{\mathtt{yes}}$ and $O_{\mathtt{no}}$ such that $O_{\mathtt{yes}} \cap O_{\mathtt{no}} = \emptyset$. The objects in $O_{\mathtt{yes}}$ all eventually evolve (or are) the object `yes`, and all those in $O_{\mathtt{no}}$ eventually evolve (or are) the object `no`.

Lemma: 4.4. *For Π_n in a uniform family of recogniser $\mathcal{AM}^0_{-d}$ systems, a size-two input alphabet $I = \{\mathtt{a}, \mathtt{b}\} \subsetneq O_{\mathtt{yes}} \cup O_{\mathtt{no}}$ is both necessary (in the worst case) and sufficient, in the sense that this restriction does not alter the computing power of the system Π_n.*

Proof. It can be seen that it is *necessary* that the input alphabet I contains at least 1 object from $O_{\mathtt{yes}}$ and 1 from $O_{\mathtt{no}}$ as follows. In a uniform family of recogniser (see Definition 2.9) membrane systems each membrane system $f(1^n) = \Pi_n$ must in the worst case decide all inputs $e(x)$ where $x \in \Sigma^n$ (see Definition 2.11). Thus each Π_n has the potential to evolve either of the `yes` and `no` objects*. In Lemma 3.3 we saw that $O_{\mathtt{yes}} \cap O_{\mathtt{no}} = \emptyset$. Thus the input set I of each member of a uniform family, if a Π_n needs to accept and reject, must have at least one object in $O_{\mathtt{yes}} \cap I$ *and* at least one object in $O_{\mathtt{no}} \cap I$.

It can be seen that it is *sufficient* for set I to contain a single object from each set $O_{\mathtt{yes}}, O_{\mathtt{no}}$ as follows. Given a membrane system Π_n, a member of a uniform family

*Of course one can think of degenerate cases such as a family that accepts all and only words of length 5.

recognising X by functions (e, f) with $|I| > 2$, we describe another system Π'_n in a uniform family by functions (e', f') that also recognises X. The system Π'_n is identical to Π_n except that: the set of objects is $O' = O \cup \{\mathtt{a}, \mathtt{b}\}$, the set of input objects (and the range of e') is $I' = \{\mathtt{a}, \mathtt{b}\}$, and there are two extra type (a) evolution rules, $[\,\mathtt{a} \to I \cap O_{\mathtt{yes}}\,], [\,\mathtt{b} \to I \cap O_{\mathtt{no}}\,]$. In the first step of its computation Π'_n uses exactly one of the input objects ($\mathtt{a}$ or $\mathtt{b}$ created by $e'(x)$) to generate $e(x)$ (as well as some "extra" objects, $(I \cap O_{\mathtt{yes}}) \backslash e(x)$). That is Π'_n evolves those objects from I that $e(x)$ produces for Π_n as and some extra, these extra objects cannot change the outcome of the computation because they all are elements of $O_{\mathtt{yes}}$ or all elements of $O_{\mathtt{no}}$. Thus uniform families of recogniser $\mathcal{AM}^0_{-d}$ systems need at most an input alphabet $I = \{\mathtt{a}, \mathtt{b}\}$ with one of $\mathtt{a}$, $\mathtt{b}$ from set $O_{\mathtt{yes}}$ and the other from set $O_{\mathtt{no}}$. □

Lemma 4.4 permits us to consider only those systems that have two input objects $I = \{\mathtt{a}, \mathtt{b}\}$. Thus we restrict attention to the case where the input encoding function is of the form $e : X \to \{\mathtt{a}, \mathtt{b}\}$. We say that e is a characteristic function with range $\{\mathtt{a}, \mathtt{b}\}$.

Lemma: 4.5. *Let* $\mathbf{\Pi}$ *be a uniform family (by functions* e *and* f*) of confluent recogniser* $\mathcal{AM}^0_{-d}$ *systems (with* $I = \{\mathtt{a}, \mathtt{b}\}$ *via Lemma 4.4) which recognises instances of* X*. There exists a family* $\mathbf{\Pi}_{\mathrm{m}}$ *that also recognises instances of* X *but uses a uniformity function* f_{m} *whose range is a set with only two membrane systems, both of which can be evaluated in* AC^0*.*

Proof. Consider the membrane system $f(n) = \Pi_n \in \mathbf{\Pi}$. The essential property of this system is that one object in its input set $I = \{\mathtt{a}, \mathtt{b}\}$ eventually evolves to $\mathtt{yes}$ while the other eventually evolves to $\mathtt{no}$ in the output membrane (the accepting/rejecting state of the membrane system). That is $\mathtt{a} \in O_{\mathtt{yes}} \Rightarrow \mathtt{b} \in O_{\mathtt{no}}$ and $\mathtt{a} \in O_{\mathtt{no}} \Rightarrow \mathtt{b} \in O_{\mathtt{yes}}$.

However, this essential property is captured by two extremely simple membrane systems with only 4 objects and a single membrane labelled 0. Both systems use the following membrane system whose input membrane i = 0:

$$(\{\mathtt{a}, \mathtt{b}, \mathtt{yes}, \mathtt{no}\}, \{(0, \emptyset)\}, \{(0, \emptyset)\}, \{0\}, \{(0, 0)\}, R).$$

Let Π_{P} be the system with the rules $R = \{[\,\mathtt{a} \to \mathtt{yes}\,]_0, [\,\mathtt{b} \to \mathtt{no}\,]_0\}$.

Let Π_{N} be the system with the rules $R = \{[\,\mathtt{a} \to \mathtt{no}\,]_0, [\,\mathtt{b} \to \mathtt{yes}\,]_0\}$.

Therefore if there is a family $\mathbf{\Pi}$, uniform by the pair (e, f), that solves X, where f outputs valid but arbitrary membrane systems then there is another family $\mathbf{\Pi}_{\mathrm{m}}$ uniform by (e, f_{m}) that also solves X and the range of f_{m} is $\{\Pi_{\mathrm{P}}, \Pi_{\mathrm{N}}\}$. Both possible members of $\mathbf{\Pi}_{\mathrm{m}}$ can be trivially evaluated in AC^0. □

Let (E, F)–$\mathsf{PMC}_{\mathcal{AM}^0_{-d}}$ be the set of problems solved by uniform families of recogniser $\mathcal{AM}^0_{-d}$ systems whose functions e, f are respectively computable in E, F. Next we prove Theorem 4.3, which we restate:

Theorem: (4.3)**.** *Let* $\mathsf{E}, \mathsf{F} \in \{\mathsf{AC}^0, \mathsf{NC}^1, \mathsf{L}, \mathsf{NL}, \mathsf{NC}^2, \mathsf{P}, \mathsf{NP}, \mathsf{PSPACE}\}$ *and let* $\mathsf{F} \subseteq \mathsf{E}$*. Then* (E, F)–$\mathsf{PMC}_{\mathcal{AM}^0_{-d}} = \mathsf{E}$.

Proof. First, we prove the upper-bound (E,F)–$\mathsf{PMC}_{\mathcal{AM}^{0}_{-d}} \subseteq \mathsf{E}$ ($\mathsf{F} \subseteq \mathsf{E}$, and the classes E, F are as given in the statement). As we have seen in Lemma 4.5, each problem in the class (E,F)–$\mathsf{PMC}_{\mathcal{AM}^{0}_{-d}}$ is solved by a family composed only of membrane systems Π_{P} and Π_{N}. To simulate this family on input x we first compute $f_{\mathrm{m}}(1^{|x|})$ and then evaluate the resulting membrane system (either Π_{P} or Π_{N}) with input $e(x)$. The weakest E that we consider is $\mathsf{E} = \mathsf{AC}^0$ and we only consider the case where $\mathsf{F} \subseteq \mathsf{E}$, thus the functions e, f are both in FAC^0. Similarly, when E is any of the classes mentioned in the statement then e and f are in the function equivalent of that class. As observed in Lemma 4.5, the membrane systems Π_{P} and Π_{N} can be simulated in AC^0.

The lower-bound $\mathsf{E} \subseteq (\mathsf{E},\mathsf{F})$–$\mathsf{PMC}_{\mathcal{AM}^{0}_{-d}}$ is easy to show. We use the fact, shown above, that e is a characteristic function with access to the input word. Thus the following simple family computes any problem from E: function $e(x) = \{\mathtt{a}\}$ if $x \in X$ and $e(x) = \{\mathtt{b}\}$ if $x \notin X$, and f_{m} is the constant function $f_{\mathrm{m}} = \Pi_{\mathrm{P}}$. □

4.2 Uniform and semi-uniform circuit complexity

The notion of uniformity was first introduced, by Borodin [12], for Boolean circuits. We briefly compare and contrast the complexity of uniform and semi-uniform families of Boolean circuits. First, we consider circuits with AND, OR, and NOT gates. Uniform families of these circuits are well-known to characterise P, we show that the same is true for semi-uniform families. If we forbid the circuits to use AND and NOT gates; the resulting *semi-uniform* families characterise NL. However, the *uniform* families of such circuits are much weaker and solve at most problems in AC^0. So in this context we see a large difference between the power of uniformity and semi-uniformity.

Any circuit (with AND, OR and NOT gates) of polynomial size and depth can be evaluated in polynomial time, this is the canonical P-complete problem known as CVP.

Problem: 4.6 (Circuit Value Problem (CVP) [39])**.**
Instance: *A Boolean circuit α, inputs $x_1, \ldots, x_n$, and a designated gate y.*
Question: *Is the value of gate y in α true on input $x_1, \ldots, x_n$?*

We now give a more detailed definition of uniform circuit family than the summary presented in Section 1.2.3.

Definition: 4.7 (Uniform families of circuits [27])**.** *A* uniform circuit family $\{\alpha_n\}$ *is an infinite collection of circuits with a single output gate such that there is a function f (computable within some resource bound) such that $f(1^n) = \alpha_n$. We say a uniform family of circuits $\{\alpha_n\}$ decides a language X if for each $x \in \{0,1\}^n$, circuit α_n evaluates to 1 if $x \in X$ and 0 if $x \notin X$.*

We now introduce a definition of semi-uniform families of Boolean circuits that is inspired by Definition 2.12.

Definition: 4.8 (Semi-uniform families of circuits)**.** *A* semi-uniform circuit family $\{\alpha_x\}$ *is an infinite collection of Boolean circuits with a single output gate such that there is a function h (computable within some resource bound) such that $h(x) = \alpha_x$*

and $x \in \{1,0\}^*$. We say a semi-uniform family of circuits $\{\alpha_x\}$ decides a language X if for each x, circuit α_x evaluates to 1 if $x \in X$ and 0 if $x \notin X$.

The complexity of uniform circuits has been well explored, we recall a well-known [27, 33] fact.

Theorem: 4.9. AC^0*-uniform circuits (with AND, OR and NOT gates) solve exactly the problems in* P*.*

Using a known reduction we show the power of semi-uniform families.

Theorem: 4.10. AC^0*-semi-uniform circuits (with AND, OR and NOT gates) solve exactly the problems in* P*.*

Proof. There exists an FAC^0 reduction, r from the P-complete problem AGAP to CVP such that $r(x) \in \mathsf{CVP}$ iff $x \in \mathsf{AGAP}$ [27, 33]. To show P-hardness, we use this reduction to define a semi-uniform family of circuits which recognise instances of AGAP by simply letting the function $h = r$ in Definition 4.8. As observed, the problem of evaluating a circuit (with AND, OR, and NOT gates) is in polynomial time. □

We have seen that both uniform and semi-uniform Boolean circuits with AND, OR and NOT gates characterise P. To show a difference in their computational power (analogous to what we have seen for membrane systems in Section 4.1) we restrict the Boolean circuits by prohibiting AND and NOT gates, we refer to these as "OR circuits". The problem of evaluating OR circuits is defined as follows:

Problem: 4.11 (OR Circuit Value Problem (ORCVP))**.**
Instance: *A Boolean circuit α (using only disjunctive logic, i.e. no AND nor NOT gates), inputs $x_1, \ldots, x_n$, and a designated gate y.*
Question: *Is the value of gate y in α true on input $x_1, \ldots, x_n$?*

We quickly observe that $\mathsf{ORCVP} \in \mathsf{NL}$: a path from the gate y to an input gate, that has been assigned 1, is non-deterministically chosen, if the path is valid then accept, else reject. We show NL-hardness for ORCVP by a reduction from STCON (Problem 3.6).

Lemma: 4.12. $\mathsf{STCON} \leq_{\mathsf{AC}^0} \mathsf{ORCVP}$

Proof. Given an instance of $G = (V, E, s, t)$ and instance of STCON, we construct a circuit α. For each vertex $v \in V$ in the graph there is an OR gate v in the circuit α. Each directed edge (v_i, v_j) in the graph becomes a wire (v_i, v_j) in the circuit α. Add a constant gate x_1 with value 1 and a wire (x_1, s) linking the 1 to the "s" gate in the circuit. Then add an output gate y and a wire (t, y) linking the "t" gate to the output gate of the circuit. We add a constant gate x_0 with value 0, for each (non-input) gate v in α with no incoming wires, add a wire connecting x_0 them to v.

If there is a path from s to t in the graph then the single constant gate x_1 with value 1 causes each successive OR gate to have value 1, including the gate y. If there is no path from s to t, then the 1 input never propagates to the gate y and the circuit answers 0. Thus the value of the gate y is true iff there is a directed path from s to t in the graph. This reduction is straightforward and is easily computable in FAC^0. □

Now we are ready to discuss the power of uniform and semi-uniform families of OR circuits.

Theorem: 4.13. AC^0*-semi-uniform families of OR circuits characterise* NL*.*

Proof. The reduction provided in Lemma 4.12 is a function r such that $r(x) \in \mathsf{ORCVP}$ iff $x \in \mathsf{STCON}$. We use this reduction to define a semi-uniform family of circuits which recognise instances of STCON by simply letting the function $h = r$ in Definition 4.8. Each circuit in the family outputs 1 iff $x \in \mathsf{STCON}$ and outputs 0 iff $x \notin \mathsf{STCON}$. Thus AC^0-semi-uniform families of OR circuits can solve NL-complete problems.

Each member of the semi-uniform family is an instance of ORCVP and thus can be evaluated in non-deterministic logspace. □

If the function h is more difficult to compute than the resulting circuit is to evaluate then the function h defines the complexity of the family. For example, if we consider a h that is computed in PSPACE then we can reduce a PSPACE-complete problem such as QSAT to ORCVP giving an "artificially" huge amount of power to the family.

Now we compare this with a characterisation for uniform families of OR circuits.

Theorem: 4.14. AC^0*-uniform families of OR gate circuits are upper-bounded by* AC^0*.*

Proof. Each member of a uniform family of circuits to decide a language has a single output gate y. To simulate such a circuit we must compute the uniformity function $f(1^{|x|})$ and then evaluate resulting circuit on input x. However a single OR gate, with inputs from $x_1, \ldots, x_\ell$ computes the same function as an arbitrary OR circuit where y is reachable from each of the inputs $x_1, \ldots, x_\ell$ (and no other input or constant gates). Thus given a family of arbitrary OR circuits there exists a family where each circuit has a single OR gate that recognises the same class of languages. A circuit family of single OR gate circuits is trivially computable in AC^0, so if the uniformity function f (see Definition 4.7) is computable in FAC^0 then the problem decided by the family is in AC^0. □

If a uniform family of circuits requires more computing resources to construct a circuit than are needed to evaluate that circuit it is unknown how this affects the set of problems solved by the family. For example, we do not know if the problems solved by P-uniform constant depth circuits with unbounded fan-in can solve more than FO-uniform AC^0 (see [8]).

Combining Theorem 4.13 and Theorem 4.14 we have:

Corollary: 4.15. *The set of problem solved by* AC^0*-uniform families of OR only circuits* $\subseteq \mathsf{AC}^0 \subsetneq \mathsf{NL} =$ *the set of problems solved by* AC^0*-semi-uniform families of OR only circuits.*

4.3 Discussion

We have shown that uniformity and semi-uniformity are actually distinct notions that can lead to large differences in computing power. This bucks a trend where, for many

models, both notions were found to coincide in the sense that they characterise the same complexity classes.

The strict difference in the power of uniform and semi-uniform families of OR circuits in Corollary 4.15 echoes our main result for membrane systems given in Theorem 4.1. However, we also note that for active membranes with dissolution ($\mathcal{AM}^0_{+d}$, see Chapter 5) and for standard Boolean circuits, AC^0 uniform, and semi-uniform, families characterise the same class, namely P. These facts are summarised in Table 4.3.

	AC^0-uniform	AC^0-semi-uniform
Circuits	P	P
$\mathcal{AM}^0_{+d}$	P	P
OR circuits	$\subseteq \mathsf{AC}^0$	NL
$\mathcal{AM}^0_{-d}$	AC^0	NL

Table 4.1: The computational power of 4 models with AC^0 uniform and semi-uniform families.

The results in this chapter concern the general concepts of uniformity and semi-uniformity and so can also be applied to other computational models besides membrane systems and circuits. Some other models that use notions of uniformity and semi-uniformity include families of: neural networks, molecular and DNA computers, tile assembly systems, cellular automata, and branching programs [9, 16, 49, 64, 65].

Chapter 5

Dissolution rules and characterising P

Previous complexity results for membrane systems had a free P lower-bound because they used P-(semi-)uniform families [28, 30]. However a P lower-bound provided by the family constructor does not tell us much about the computing power of the model itself. For example: in Chapter 3 we lowered the power of some models from P down to NL by restricting the semi-uniform families to be constructable in FAC^0 rather than FP; and more surprisingly, the same restriction to uniform families in Chapter 4 lowered the power of some models from P down to AC^0! In Section 5.1 of this chapter we give an AC^0-uniform family of $\mathcal{AM}^0$ systems which solves a P-complete problem. This is the first P lower-bound for active membrane systems where the problem is solved by the members of the family and not by the uniformity or input encoding function. This P lower-bound is also interesting as it uses only evolution and dissolution rules.

In Section 5.2 we attack the open problem known as the "P-conjecture".

Conjecture: 5.1 (P-conjecture [56])**.** *The class of all decision problems solvable in polynomial time by active membranes without charges using evolution, communication, dissolution and division rules for elementary membranes is equal to the class* P*.*

We prove that the conjecture holds when we restrict the systems to use type ($\mathrm{e_s}$) rules instead of type (e). Specifically this restriction insists that the two membranes that result from a weak elementary division rule must be identical. This more closely models mitosis, the biological process of cell division, and we refer to it using the biological term "symmetric division" [2]. When the two resulting daughter cells are different, as is the case for type (e) rules (weak elementary division), we use the biological term "asymmetric division". Stem cells use asymmetric division in nature as a way to achieve cell differentiation. So, as well as providing a partial result for the P-conjecture, this variation is biologically motivated.

The lower-bound and upper-bound in this chapter combine to give a characterisation of P for uniform and semi-uniform families of $\mathcal{AM}^0_{+d,-a}$ systems (dissolution and symmetric non-elementary division rules). The results are visualised in Figure 5.1.

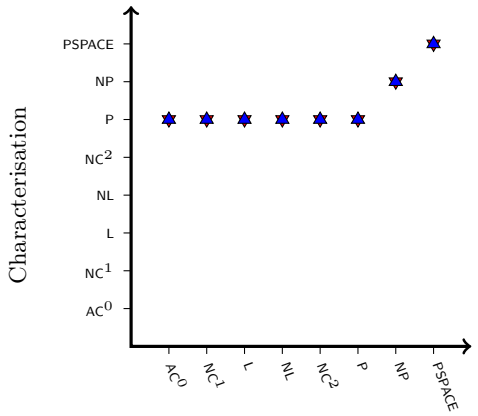

Figure 5.1: Characterisations by semi-uniform families of recogniser $\mathcal{AM}^0_{+d,-a}$ systems are denoted by ▼, uniform families of recogniser $\mathcal{AM}^0_{+d,-a}$ systems are denoted by ▲. In this case they are equal in power and the symbol appears as ★.

5.1 P lower-bound for uniform families with dissolving rules

In this section we prove that the set of problems solved by AC^0-uniform families of polynomial time active membrane systems with dissolution rules contains P.

Theorem: 5.2. $\mathsf{P} \subseteq (\mathsf{AC}^0, \mathsf{AC}^0)\text{–}\mathsf{PMC}_{\mathcal{AM}^0_{+d}}$

This is shown by solving AGAP [27], the P-complete analogue of STCON (also known as GAP). We actually consider a slight restriction of AGAP where the vertices are ordered topologically. This problem is NC^1 complete for P via a reduction from the topologically sorted monotone circuit value problem [27].

Problem: 5.3 (Topologically Sorted Alternating GAP (TAGAP))**.**
Instance: *A directed acyclic graph $G = (V, E, A, s, t)$ where $s, t \in V$, and V is the ordered list of vertices such that if $(p, c) \in E$ then p is before c in V. Let $A \subseteq V$ be the set of universal vertices.*
Question: *Is* apath(s, t) *true? The function* apath(x, y) *holds iff*

- *$x = y$ or*
- *x is existential (that is $x \in V \backslash A$) and there exists $z \in V$ with $(x, z) \in E$ and* apath(z, y) *holds, or*
- *x is universal (that is $x \in A$) and for all $z \in V$ with $(x, z) \in E$,* apath(z, y) *holds.*

We define a uniform family of membrane systems $\Pi_n \in \mathbf{\Pi}$ to decide if $x \in$ TAGAP. The system Π_n decides all instances of x where the graph encoded in x has n vertices, the number of vertices $|V|$ is trivially obtainable from $1^{|x|}$. Without loss of generality

we assume that the vertices of the graph are numbered from 1 to n, and that $s = 1$ and $t = n$. The algorithm used by each membrane in the family operates in two distinct phases. The first phase records paths through the graph, the second phase uses these paths to evaluate apath(s, t). The output of $e(x)$ is a set of objects and is placed in the input membrane "i" of the membrane system, $e(x) \subseteq M(\mathrm{i})$. In this family the instance $x \in \mathsf{TAGAP} \cup \mathsf{coTAGAP}$ to be input to Π_n is encoded by the function e as follows:

$$\begin{aligned} e(x) = &\{s, \delta, d_4\} \\ &\cup \{\cancel{u v} \mid (u, v) \notin E\} \\ &\cup \{v_\forall \mid v \in A\} \\ &\cup \{v_\exists \mid v \in V \backslash A\}. \end{aligned}$$

Lemma: 5.4. *For this uniform family deciding* TAGAP*, the function $e(x)$ is computable in* FAC^0.

Proof. We detail how a CRAM generates the objects $\cancel{u v}$ representing all edges *not* in E.The problem instance x is assumed to encode the edges of the graph in a binary adjacency matrix M of size $|V|^2$. If the element at $M_{u,v}$ is 1 there is an edge $(u, v) \in E$, if 0 there is no edge. The CRAM uses a gird of $|V|^2$ processors to write out the list of non-edges in 2 steps. Each processor reads a single designated element of $M_{u,v}$. If the element $M_{u,v}$ is 1 then the CRAM writes out a blank "␣" to its output register; if $M_{u,v}$ is 0 then the processor writes out "$\cancel{u v}$" to its output register. The other objects generated by $e(x)$ are straightforward to produce in constant time. □

When the function $f : \{1\}^* \to \mathbf{\Pi}$ is given $|x|$ in unary it constructs (in constant parallel time) a membrane system Π_n that recognises all instances TAGAP of length $|x|$*. The membrane system $\Pi_n = f(1^{|x|})$ is as follows:

- The membrane to label relation, L, is the identity relation (unique labels).
- The membrane structure, $\mu = (V_\mu, E_{\mu,\text{first}} \cup E_{\mu,\text{second}})$, is a sequence of concentric membranes. The sets $E_{\mu,\text{first}}$ and $E_{\mu,\text{second}}$ are defined separately in Sections 5.1.1 and 5.1.2.
- The set of rules $R = R_{\text{first}} \cup R_{\text{second}}$, the sets R_{first} and R_{second} are defined in Sections 5.1.1 and 5.1.2.
- The sets of objects, O, and labels, H, are all those mentioned in μ and R.

To ease the explanation of the computation of such a membrane system we break the construction into two sections, in Section 5.1.1 we describe the first phase of the algorithm and in Section 5.1.2 the second phase of the algorithm.

*The encoding of TAGAP ensures that all instances of $|x|$ have n vertices.

5.1.1 First phase: following all paths in the graph

The first phase of the membrane system follows (in parallel) all paths through the graph starting from s. The membrane structure of the first phase of the membrane algorithm is a sequence of concentric membranes representing all possible edges in a graph with n vertices. Each membrane has a unique label and they are arranged such that there is a "reset" membrane interleaved between the membranes representing edges.

$$\begin{aligned} E_{\mu,\text{first}} = \bigcup_{x=1}^{n-1} \Big\{ & \{(\langle x, y+1\rangle, r_{\langle x,y\rangle,\langle x,y+1\rangle}) \mid y \in \{2,\ldots,n-2\} \wedge x < y\} \\ & \cup \{(r_{\langle x,y\rangle,\langle x,y+1\rangle}, \langle x, y\rangle) \mid y \in \{2,\ldots,n-2\} \wedge x < y\} \\ & \cup \{(r_{\langle x,n\rangle,\langle x+1,x+2\rangle}, \langle x, n\rangle)\} \\ & \cup \{(\langle x+1, x+2\rangle, r_{\langle x,n\rangle,\langle x+1,x+2\rangle})\}\Big\} \cup \{(\langle 1,2\rangle, \mathrm{i})\} \end{aligned}$$

A CRAM using $n \times n$ processors can write out the structure $E_{\mu,\text{first}}$ in constant time. The number of each processor represents all different combinations of x and y. If $x < y$ and $2 \leq y \leq n-2$ then the processor writes out two edges to its output registers representing $(\langle x, y+1\rangle, r_{\langle x,y\rangle,\langle x,y+1\rangle})$ and $(r_{\langle x,y\rangle,\langle x,y+1\rangle}, \langle x, y\rangle)$. The other edges are straightforward to generate.

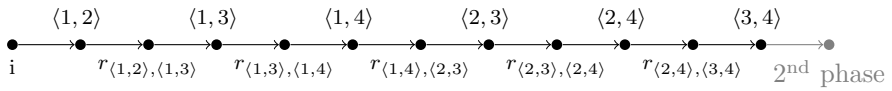

Figure 5.2: The section of the membrane structure used by the first phase of a membrane system Π_4 from our uniform family to recognise TAGAP instances. In this case the number of vertices is $n = 4$. The direction of the arrows ($\rightarrow$) indicates the movement of the objects during the computation as they dissolve their way up the membrane structure. If the arrow directions are reversed we see the parent-child relationships of the vertices. The parent of the vertex $\langle 3, 4\rangle$ is the first membrane of the second phase of the computation.

We now define R_{first}, the set of membrane rules for the first phase of the algorithm. Each set of rules from (5.1) to (5.7) is a subset of R_{first} and are specified over the range of numbers from 1 to n so a CRAM can output R_{first} the whole set of rules in constant time.

The first phase of the membrane system algorithm produces objects which represent every (and only) edge(s) in E that are traversable from s, as well as the distance(s) of each vertex from s on all paths through the graph. The vertices of each of these edge objects is marked with their distance from s on some path. These edges are created when an object u, marked with some length, is in a membrane labeled "$\langle u, v\rangle$", the object u is preserved by an evolution rule and two new objects are created, one object representing an edge from u to v, and another representing vertex v. The membrane

is then dissolved and the objects move to a reset membrane where they are prepared for another edge test. The input set includes an object for each possible edge not in E, these objects are used to control the algorithm and prevent objects being generated for edges not in the graph. They do this by dissolving the membranes representing an edge not in the input graph before any objects are affected by its rules.

We start the computation in the input membrane "i", the most deeply nested membrane in the structure E_μ (see Figure 5.2). Of the original input multiset $M(\mathrm{i}) = e(x)$ the first phase uses the objects $\{s, \delta\} \cup \{\cancel{uv} \mid (u, v) \notin E\}$.

In the first step of the computation the membrane "i" is dissolved and the object $s'\mathrm{L}_0$ is created. This represents that the algorithm has started following a path that is currently of length 0, and is currently at vertex s (the start vertex).

$$[\, s \,]_{\mathrm{i}} \;\rightarrow\; s'\mathrm{L}_0 \tag{5.1}$$

The objects are now in the first of a number of nested membranes which alternate from "reset" membranes ($r_{\langle w,x\rangle,\langle y,z\rangle}$) to membranes representing possible edges $\langle w, x\rangle$ in the graph. Now the computation of the first phase really begins, the steps below repeat for each nested pair of membranes $\langle w, x\rangle, r_{\langle w,x\rangle,\langle y,z\rangle}$ in the nested structure. The following are all sets of rules defined where $u \in \{0, \ldots, n-1\}$, $v \in \{1, \ldots, n\}$, and $i \in \{0, \ldots, n-1\}$.

First time-step: In the first time-step, any object representing a vertex in the graph of type u (for any distance from s) looses its prime and becomes ready to evolve in the following time-step.

$$[\, u'\mathrm{L}_i \;\rightarrow\; u\mathrm{L}_i \,]_{\langle u,v\rangle} \tag{5.2}$$

In the same time-step, in parallel, if there is an object of type $\cancel{uv}$ in a membrane with label $\langle u, v\rangle$ it dissolves the membrane, this prevents the vertex objects ($u\mathrm{L}_i$) from "following" this non-existent edge on a path through the graph.

$$[\cancel{uv}]_{\langle u,v\rangle} \;\rightarrow\; \lambda \tag{5.3}$$

Second time-step: If Rule (5.3) was applied then the edge (u, v) does not exist in the input graph. The membrane representing the edge was dissolved and the objects are now in the reset membrane. Here the primes removed in the first time-step are reapplied to all objects representing vertices. All reset membranes have identical rules, thus we have $u, v, w, x \in \{0, \ldots, n\}$.

$$[\, u\mathrm{L}_i \;\rightarrow\; u'\mathrm{L}_i \,]_{r_{\langle u,v\rangle,\langle w,x\rangle}} \tag{5.4}$$

The object δ dissolves the reset membrane and the objects are enter a new edge membrane and these steps repeat until all membranes of the first phase are dissolved.

$$[\, \delta \,]_{r_{\langle u,v\rangle,\langle w,x\rangle}} \;\rightarrow\; \delta \tag{5.5}$$

Second time-step: If Rule (5.3) was not applied, then the edge (u, v) is in the input graph. The primes have been removed from the objects representing vertices allowing

them to be used in this time-step by an evolution rule to creates an object representing and edge from u to v.

$$[\, u\mathrm{L}_i \;\rightarrow\; u\mathrm{L}_i\text{-}v\mathrm{L}_{i+1}{}',\, u\mathrm{L}_i,\, v\mathrm{L}_{i+1},\, d\,]_{\langle u,v\rangle} \tag{5.6}$$

Rule (5.6) is only applicable in the case where the vertex v is reachable from u in the graph (that is if Rule (5.3) was not applied). The object $u\mathrm{L}_i$-$v\mathrm{L}_{i+1}$ records that the edge was followed on a path from s and the distances these vertices are on the path, the object $v\mathrm{L}_{i+1}$ is kept for the case where there are several edges leaving the vertex u. If the vertices are not numbered in topological order, Rule (5.6) may not follow a path correctly through the graph.

Third time-step: Now that the edge object has been generated we use the d object to dissolve the membrane and move the contents into the reset membrane.

$$[\, d\,]_{\langle u,v\rangle} \;\rightarrow\; \lambda \tag{5.7}$$

Fourth time-step: The primes are returned to the vertex objects via Rule (5.4) while the δ object dissolves the reset membrane via Rule (5.5). The objects enter a new edge membrane and these steps repeat until all membranes of first phase are dissolved.

We began this phase with an object representing the start vertex s, this ensures all paths generated are rooted at s. This s object is tagged with L_0, thus, only objects with the correct path lengths L_i are generated. The vertices of the input problem are ordered topologically, ensuring that the algorithm will not encounter an edge (u, v) until the vertex u has been encountered.

These points guarantee that at the end of the first phase of the algorithm, in the system there is (i) at least one object $u\mathrm{L}_i$-$v\mathrm{L}_{i+1}{}'$ representing each edge (u, v) in the input graph reachable from s, and (ii) each edge is marked with i and $i + 1$, the two component vertices' distances from s. Note that if there are multiple paths involving the same vertices, objects representing the same edges with different lengths are generated.

5.1.2 Second phase: checking for an alternating path

The second phase of the algorithm evaluates if there is an alternating path from the start vertex s to the terminal vertex t in the input graph, that is, the function $\mathrm{apath}(s, t)$ evaluates to true. In Problem 5.3, the function apath is defined recursively, so to evaluate $\mathrm{apath}(u, t)$ it is necessary to evaluate $\mathrm{apath}(v, t)$ for all v where $(u, v) \in E$. Accordingly, the second phase of the algorithm evaluates the vertices at the non s end of each path first, then works backwards along the paths until they finally converge at the start vertex s. The algorithm considers each vertex at each distance $\leq n - 1$ (i.e. up to the maximum possible path length). By evaluating $\mathrm{apath}(u, t)$ for a vertex u at distance i from vertex s on some path, it provides a piece of information used to evaluate a vertex on the same path at distance $i - 1$. By evaluating every vertex at distance i, the algorithm ensures that it has all necessary information to evaluate all

vertices at distance $i-1$. When the algorithm reaches $i=0$ it evaluates the start vertex s and outputs the answer to this instance of the problem.

The membrane structure used by the second phase of the algorithm is another sequence of concentric membranes (see Figure 5.3) defined as follows:

$$\begin{aligned}
E_{\text{start}} &= \{(n\mathrm{L}_{n-2}, n\mathrm{L}_{n-1}), (n\mathrm{L}_{n-1}, \langle n-1, n\rangle)\} \\
E_{\text{apath1}} &= \{(u_{\exists}\mathrm{L}_i, u_{\forall}\mathrm{L}_i) \mid 1 < u < n, 0 < i < n-1\} \\
E_{\text{apath2}} &= \{(e_u\mathrm{L}_i, u_{\exists}\mathrm{L}_i) \mid 1 < u < n, 0 < i < n-1\} \\
E_{\text{vlink}} &= \{(u-1_{\forall}\mathrm{L}_i, e_u\mathrm{L}_i) \mid 2 < u < n, 0 < i < n-1\} \\
E_{\text{nlink}} &= \{(n-1_{\forall}\mathrm{L}_i, n\mathrm{L}_i) \mid 0 < i < n-1\} \\
E_{\text{elink}} &= \{(n\mathrm{L}_{i-1}, e_2\mathrm{L}_i) \mid 1 < i < n-1\} \\
E_{\text{head}} &= \{(1_{\forall}\mathrm{L}_0, e_2\mathrm{L}_1), (1_{\exists}\mathrm{L}_0, 1_{\forall}\mathrm{L}_0), (0, 1_{\exists}\mathrm{L}_0)\} \\
E_{\mu,\text{second}} &= E_{\text{start}} \cup E_{\text{apath1}} \cup E_{\text{apath2}} \cup E_{\text{nlink}} \cup E_{\text{elink}} \cup E_{\text{head}}
\end{aligned}$$

This structure is easily constructed by a constant time CRAM: using $|V|^2$ processors the CRAM prepares all $n \times n$ values of u and i and then writes out the edges $E_{\mu,\text{first}}$, representing the membrane structure of the second phase, as specified above. The membrane $\langle n-1, n\rangle$ is from the first phase of the algorithm and is contained in the most deeply nested membrane of the second phase, $n\mathrm{L}_{n-1}$.

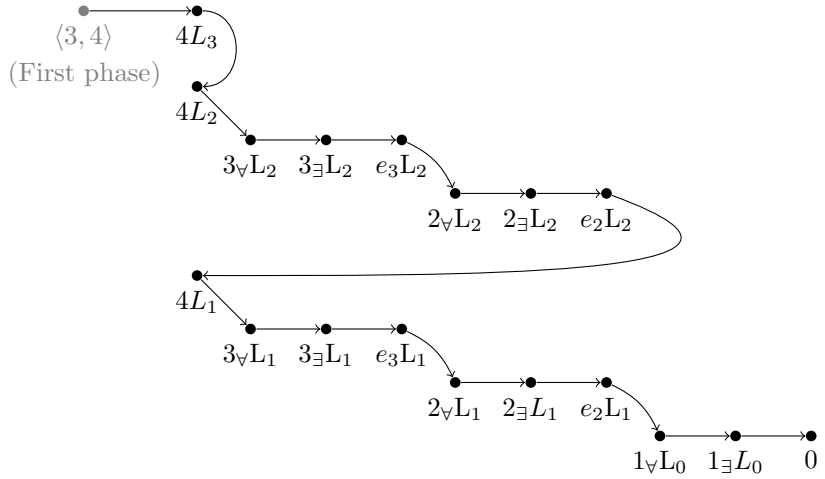

Figure 5.3: The membrane structure of the second phase of our uniform family to solve TAGAP, in this case the number of vertices $n = |V| = 4$. The direction of the arrows ($\rightarrow$) indicates the movement of the objects through the structure by dissolving membranes. To see the parent child relationship, reverse the direction of the arrows. The child of first vertex is the most outer membrane used by the first phase of the computation, in this case $\langle 3, 4\rangle$.

The second phase of the algorithm evaluates $\text{apath}(s, t)$ using the information generated in the first phase. It commences in a membrane labeled "$n\mathrm{L}_{n-1}$" (recall that the vertex t is assumed to be numbered n and the longest path in an acyclic graph

is $n-1$), which is the most deeply nested membrane at this point in a computation since all membranes from the first phase have been dissolved (see Figure 5.3). The objects in membrane $n\mathrm{L}_{n-1}$ are: the objects produced by the first phase, $\{u\mathrm{L}_i\text{-}v\mathrm{L}_{i+1}{}' \mid (u,v) \in E, i \in \{1,\ldots,n-1\}\}$ where the index i represents the distance that each vertex is from s on a path through the graph; and some objects from the input encoder $e(x)$ that are unused from the first phase $\{v_\forall \mid v \in A\} \cup \{v_\exists \mid v \in V \backslash A\}$. There is also a counter object d_4 which is used to dissolve membranes at the appropriate time.

First we explain the rule applied when an object representing an edge that reaches t is encountered (that is (u,t)). In this uniform solution, the vertex t is the vertex labeled n in the graph.

$$\left[\ u\mathrm{L}_{i-1}\text{-}n\mathrm{L}_i{}' \rightarrow \mathrm{Y}u\mathrm{L}_{i-1}\ \right]_{n\mathrm{L}_i} \text{ where } u,i \in \{1,\ldots,n-1\} \tag{5.8}$$

This is the where the first "yes" ($\mathrm{Y}v\mathrm{L}_{i-1}$) objects appear. These objects are used by Rules (5.15) to (5.44) to generate further "yes" or "no" ($\mathrm{N}u\mathrm{L}_{i-1}$) objects which represent the positive or negative evaluation of apath(u,t) where $v \in V$ and $(u,v) \in E$.

The rules to evaluate a universal vertex v use the fact that apath(v,t) does not hold if there exists $(v,w) \in E$ such that apath(w,t) does not hold. The rules to evaluate an existential vertex v are based on the same principle except that apath(v,t) holds if there exists $(v,w) \in E$ such that apath(w,t) holds.

The evaluation of apath(v,t) assumes that all vertices that are on paths from v to t (excluding v) have been evaluated by the rules of the second phase of the algorithm. These rules use three membranes to evaluate each object representing an edge marked with vertex distances from s, the steps of this process are visible in Figure 5.4.

In the following sub-sections we define the set of membrane rules R_{second} for the second phase of the algorithm. Each of the Rules (5.13) to (5.44) are subsets of R_{second}. Let $q \in \{\forall, \exists\}$.

The Rules (5.9) to (5.13) are common for evaluating both universal and existential vertices and are defined where $u \in \{1,\ldots,n-2\}, v \in \{1,\ldots,n\}$ and $i \in \{0,\ldots,n-1\}$.

First time-step: Figure 5.4, parts (e), (f), (g), and (h). The objects representing edges and apath evaluations that are related to vertex v at distance i have their primes removed, ready to be used in the next time-step.

$$[\, u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}' \rightarrow u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i \,]_{v_q\mathrm{L}_i} \tag{5.9}$$

$$[\, \mathrm{N}v\mathrm{L}_i{}' \rightarrow \mathrm{N}v\mathrm{L}_i \,]_{v_q\mathrm{L}_i} \tag{5.10}$$

$$[\, \mathrm{Y}v\mathrm{L}_i{}' \rightarrow \mathrm{Y}v\mathrm{L}_i \,]_{v_q\mathrm{L}_i} \tag{5.11}$$

The counter object is decremented:

$$[\ d_4 \rightarrow d_3\ ]_{v_\forall\mathrm{L}_i} \tag{5.12}$$

If the vertex v is existential, then the universal membrane is dissolved:

$$[\, v_\exists \,]_{v_q\mathrm{L}_i} \rightarrow v_\exists \tag{5.13}$$

In Section 5.1.2.1 we explain the rules for evaluating universal vertices, that is where Rule (5.13) *is not* applied. If Rule (5.13) *is* applied, then v is existential and the rules for this case are detailed in Section 5.1.2.2.

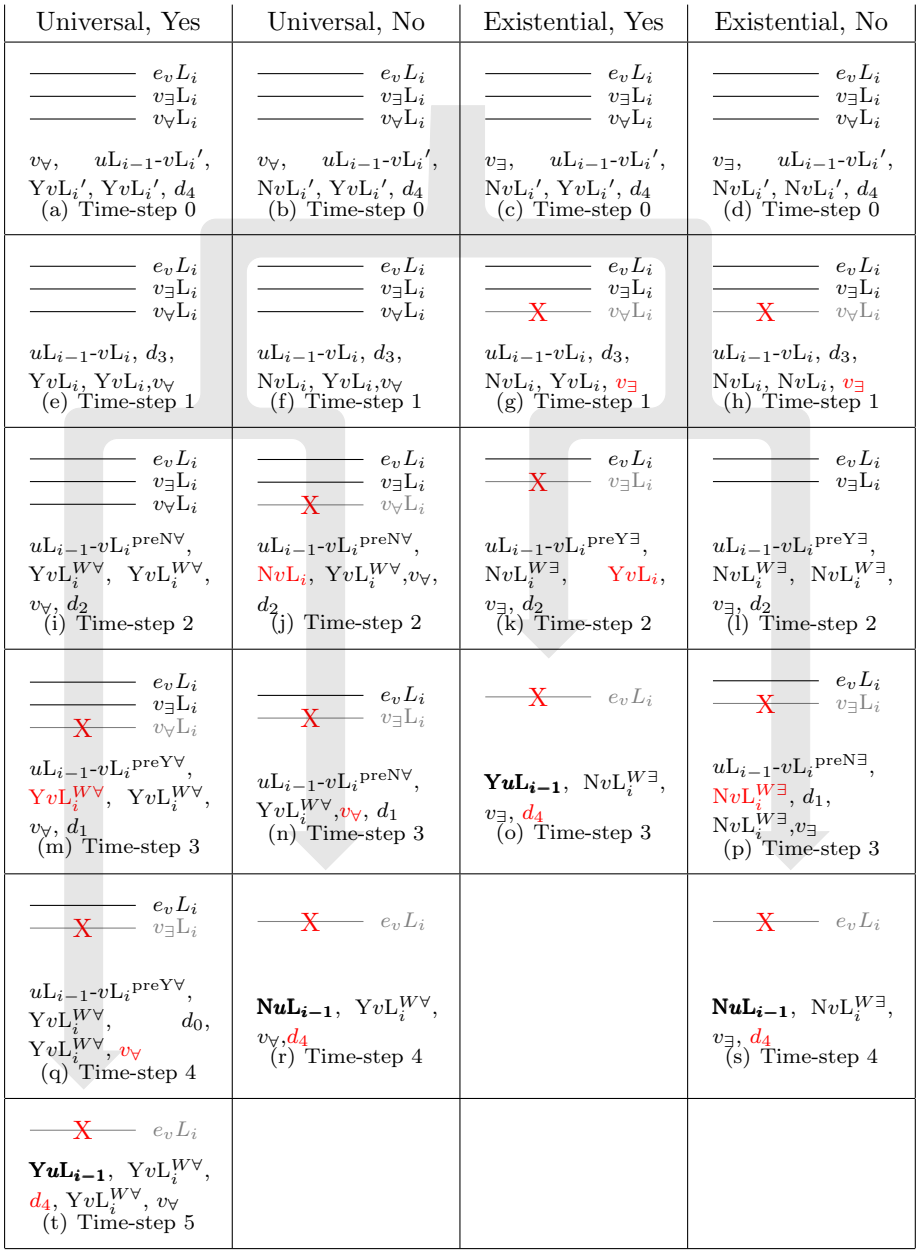

Figure 5.4: The steps needed by the second phase of the algorithm to evaluate the function apath(v, t) on a vertex v in a graph. The initial objects decide the final outcome, the branches of the grey arrow indicate where the different computation paths diverge.

5.1.2.1 Rules for universal vertices

In this section the rules are defined over $u \in \{1, \ldots, n-2\}, v \in \{2, \ldots, n-1\}$ and $i \in \{0, \ldots, n-1\}$, unless stated otherwise.

Second time-step: Figures 5.4(i), 5.4(j). The objects remain in membrane $v_\forall \mathrm{L}_i$ since Rule (5.13) was applied. In a universal vertex v, apath(v, t) returns false if there is a single negative apath(w, t) evaluation for any child vertex, w, of v. Accordingly, the algorithm awaits a $\mathrm{N}v\mathrm{L}_i$ object to dissolve the membrane:

$$[\ \mathrm{N}v\mathrm{L}_i\ ]_{v_\forall \mathrm{L}_i} \rightarrow \lambda, \tag{5.14}$$

and prepares the edge object to be become a $\mathrm{N}u\mathrm{L}_{i-1}$ object.

$$\left[\ u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i \rightarrow u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preN}\forall}\ \right]_{v_\forall \mathrm{L}_i} \tag{5.15}$$

Any $\mathrm{Y}v\mathrm{L}_i$ objects must wait for the next time-step, they are tagged with "$W\forall$".

$$\left[\ \mathrm{Y}v\mathrm{L}_i \rightarrow \mathrm{Y}v\mathrm{L}_i{}^{W\forall}\ \right]_{v_\forall \mathrm{L}_i} \text{ where } v \in \{1, \ldots, n-1\} \tag{5.16}$$

The "$\forall$" tag on the objects generated by Rules (5.15) and (5.16) prevents these objects reacting while they pass through the parent $u_\exists \mathrm{L}_i$ membrane while on their way to the evaluating membrane, $e_u \mathrm{L}_i$. The counter object is decremented:

$$[\ d_3 \rightarrow d_2\ ]_{v_q \mathrm{L}_i} \text{ where } v \in \{1, \ldots, n-1\} \tag{5.17}$$

We continue with the case where a no object ($\mathrm{N}v\mathrm{L}_i$) which dissolved the membrane $v_\forall \mathrm{L}_i$ via Rule (5.14). This implies that the universal vertex v does not have an alternating path to t (that is apath(v, t) does not hold). The description of the computation where Rule (5.14) *was not applied* recommences at Rule (5.22) and Figure 5.4(m).

Third time-step: Figure 5.4(n). In the previous step the membrane $v_\forall \mathrm{L}_i$ was dissolved by $\mathrm{N}v\mathrm{L}_i$ via Rule (5.14) and the objects are now in the existential membrane $v_\exists \mathrm{L}_i$. However, vertex v is universal so the membrane $v_\exists \mathrm{L}_i$ is immediately dissolved by the $v_\forall$ object.

$$[\, v_\forall\,]_{v\mathrm{L}_i \exists} \rightarrow v_\forall \tag{5.18}$$

The objects previously tagged with $\forall$ by Rules (5.15) and (5.16) are not affected by Rules (5.27) to (5.34) as they pass through the existential membrane. The counter decrements:

$$[\ d_2 \rightarrow d_1\ ]_{v_q \mathrm{L}_i} \text{ where } v \in \{1, \ldots, n\} \tag{5.19}$$

Fourth time-step: Figure 5.4(r). The objects are in evaluating membrane $e_v \mathrm{L}_i$

where the prepared $u\mathrm{L}_{i-1}$-$v\mathrm{L}_i{}^{\mathrm{preN}\forall}$ object evolves to be object $\mathrm{N}u\mathrm{L}_{i-1}$. This implies that $\mathrm{apath}(v,t)$ is false.

$$\left[\ u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preN}\forall}\ \rightarrow\ \mathrm{N}u\mathrm{L}_i{}'\ \right]_{e_v\mathrm{L}_i} \tag{5.20}$$

At the same time, the counter dissolves the evaluating membrane and resets. It will be used again, along with the other objects, in the next series of membranes.

$$[\ d_1\ ]_{e_v\mathrm{L}_i}\ \rightarrow\ d_4 \tag{5.21}$$

Now we go back a step to explain what happens if there was no $\mathrm{N}v\mathrm{L}_i$ object and Rule (5.14) was not applied.

Back track, third time-step: Figure 5.4(m). Here we consider the case that Rule (5.14) was not applied because there was no $\mathrm{N}v\mathrm{L}_i$ object to dissolve the universal membrane $v_\forall\mathrm{L}_i$. This implies that either v has no children or that all of its children are on a path to t. The algorithm prepares for the case that one of the child vertices does have an alternating path to t by preparing the edge object to produce a yes object in the following time-step.

$$\left[\ u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preN}\forall}\ \rightarrow\ u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preY}\forall}\right]_{v_\forall\mathrm{L}_i} \tag{5.22}$$

The counter object is decremented via Rule (5.19). If there are any $\mathrm{Y}v\mathrm{L}_i^{W\forall}$ objects present, they can now dissolve the membrane,

$$\left[\ \mathrm{Y}v\mathrm{L}_i^{W\forall}\right]_{v_\forall\mathrm{L}_i}\rightarrow\lambda. \tag{5.23}$$

Fourth time-step: Figure 5.4(q). If Rule (5.23) was applied there was a child of v with an alternating path to t, the universal membrane has been dissolved and the objects are now in the existential membrane $v\exists\mathrm{L}_i$. However, vertex v is universal so the membrane is immediately dissolved by the $v_\forall$ object via Rule (5.18). The objects previously tagged with $\forall$ by Rules (5.15) and (5.16) are not affected by Rules (5.27) to (5.34) as they pass through the existential membrane. The counter decrements:

$$[\ d_1\ \rightarrow\ d_0\ ]_{v_q\mathrm{L}_i}\,. \tag{5.24}$$

Fifth time-step: Figure 5.4(t). In the evaluating membrane $e_v\mathrm{L}_i$, the prepared object $u\mathrm{L}_{i-1}$-$v\mathrm{L}_i{}^{\mathrm{preY}\forall}$ becomes a yes object for u ($\mathrm{Y}u\mathrm{L}_{i-1}{}'$), this represents that $\mathrm{apath}(v,t)$ evaluates to true.

$$\left[\ u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preY}\forall}\ \rightarrow\ \mathrm{Y}u\mathrm{L}_{i-1}{}'\ \right]_{e_v\mathrm{L}_i} \tag{5.25}$$

At the same time, the counter dissolves the evaluating membrane and resets. It will be used again, along with the other objects, in the next series of membranes.

$$[\ d_0\ ]_{e_v\mathrm{L}_i}\rightarrow d_4 \tag{5.26}$$

5.1.2.2 Evaluating existential vertices

To evaluate existential vertices (those in $V \backslash A$) the algorithm and rules work in a similar manner to those described in Section 5.1.2. The main principle here is that $\text{apath}(v,t)$ is true if $\text{apath}(w,t)$ is true for any w where w is a child vertex of v. The rules here are defined over $u \in \{1,\ldots,n-2\}, v \in \{2,\ldots,n-1\}$ and $i \in \{0,\ldots,n-1\}$, unless stated otherwise.

First time-step, Figure 5.4(g) and 5.4(h). If Rule (5.13) is applied in the first time-step, then vertex v is existential. The objects that will be used to evaluate this vertex are de-primed via Rules (5.9), (5.10), and (5.11). The counter is decremented via Rule (5.12).

Second time-step , Figure 5.4(k). Any $\text{N}v\text{L}_i$ objects are tagged with a $W\exists$.

$$\left[\ \text{N}v\text{L}_i \ \rightarrow\ \text{N}v\text{L}_i{}^{W\forall}\ \right]_{v_\exists \text{L}_i} \text{ where } v \in \{1,\ldots,n\} \tag{5.27}$$

Any $u\text{L}_{i-1}$-$v\text{L}_i$ objects prepare to become yes objects.

$$\left[\ u\text{L}_{i-1}\text{-}v\text{L}_i \ \rightarrow\ u\text{L}_{i-1}\text{-}v\text{L}_i{}^{\text{preY}\exists}\ \right]_{v_\forall \text{L}_i}. \tag{5.28}$$

If there are any $\text{Y}v\text{L}_i$ objects then a child of v has a alternating path to t, dissolve the membrane.

$$[\ \text{Y}v\text{L}_i\ ]_{v_\exists \text{L}_i} \rightarrow \lambda, \tag{5.29}$$

The counter decrements via Rule (5.17).

Third time-step, Figure 5.4(o). The prepared edge object becomes a yes object implying that $\text{apath}(v,t)$ holds.

$$\left[\ u\text{L}_{i-1}\text{-}v\text{L}_i{}^{\text{preY}\exists}\ \rightarrow\ \text{Y}u\text{L}_{i-1}{}'\ \right]_{e_v \text{L}_i} \tag{5.30}$$

At the same time, the counter dissolves the evaluating membrane and resets. It will be used again, along with the other objects, in the next series of membranes.

$$[\ d_2\ ]_{e_v \text{L}_i} \rightarrow d_4 \tag{5.31}$$

Now we go back a time-step and explain the rules for the case where none of the children of v had an alternating path to t.

Back track to the third time-step, Figure 5.4(p). If none of the children of v had an alternating path to t then there was no $\text{Y}v\text{L}_i$ object to dissolve the existential membrane $v_\exists \text{L}_i$. The algorithm now prepares the edge object to become a no object $\text{N}ui-1$.

$$\left[\ u\text{L}_{i-1}\text{-}v\text{L}_i{}^{\text{preY}\exists}\ \rightarrow\ u\text{L}_{i-1}\text{-}v\text{L}_i{}^{\text{preN}\exists}\right]_{v_\exists \text{L}_i} \tag{5.32}$$

Any no objects that were waiting ($\mathrm{N}v\mathrm{L}_i^{W\exists}$) may now dissolve the membrane.

$$\left[\ \mathrm{N}v\mathrm{L}_i^{W\exists}\right]_{v\exists\mathrm{L}_i} \to \lambda \tag{5.33}$$

The counter decrements via Rule (5.19).

Fourth time-step, Figure 5.4(s). In the evaluation membrane, the prepared object $u\mathrm{L}_{i-1}$-$v\mathrm{L}_i{}^{\mathrm{preN}\exists}$ becomes an $\mathrm{N}u\mathrm{L}_{i-1}{}'$ object, this implies that apath(v,t) evaluates to false.

$$\left[\ u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preN}\exists} \to \mathrm{N}u\mathrm{L}_{i-1}{}'\ \right]_{e_v\mathrm{L}_i} \tag{5.34}$$

The counter dissolves the evaluation membrane via Rule (5.26) and the objects are released to the parent membrane for use in evaluating future vertices. The rules here are defined over $u \in \{1,\ldots,n-2\}, v \in \{2,\ldots,n-1\}$ and $i \in \{0,\ldots,n-1\}$, unless stated otherwise.

5.1.2.3 Dead ends and unused vertices

If there is an edge (u,v) in the graph and vertex v has no child vertices (and $v \neq t$), then apath(v,t) does not hold. There can be neither $\mathrm{Y}v\mathrm{L}_i$ nor $\mathrm{N}v\mathrm{L}_i$ objects in the membrane $v_\forall\mathrm{L}_i$ so the algorithm uses the edge object $u\mathrm{L}_{i-1}$-$v\mathrm{L}_i{}'$ to produce the $\mathrm{N}u\mathrm{L}_{i-1}$ object.
Vertex v is a dead end. As usual, Rule (5.13) dissolves membrane $v\mathrm{L}_i$ if v is an existential membrane. If v is a universal vertex then the edge object evolves via Rules (5.9), (5.15), (5.22):

$$u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}' \to u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i \to u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preN}\forall} \to u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preY}\forall}$$

While if v is an exponential vertex, then the edge object evolves in membrane $v_\exists\mathrm{L}_i$ via Rules (5.9), (5.28), (5.32):

$$u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}' \to u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i \to u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preY}\exists} \to u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preN}\exists}$$

During these three time-steps, the counter decrements $d_4 \to d_3 \to d_2 \to d_1$ via Rules (5.12), (5.17), and (5.19).

In the fourth time-step, the objects are still in the universal $v_\forall\mathrm{L}i$, or existential $v_\exists\mathrm{L}i$, membrane due to the absence of $\mathrm{Y}u\mathrm{L}_i$ or $\mathrm{N}u\mathrm{L}_i$ objects. The edge object either dissolves the $v_q\mathrm{L}i$ membrane and creates object $\mathrm{N}u\mathrm{L}_{i-1}{}'$:

$$\left[\ u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preY}\forall}\ \right]_{v_\forall\mathrm{L}_i} \to \mathrm{N}u\mathrm{L}_{i-1}{}', \tag{5.35}$$

or

$$\left[\ u\mathrm{L}_{i-1}\text{-}v\mathrm{L}_i{}^{\mathrm{preN}\exists}\ \right]_{v_\exists\mathrm{L}_i} \to \mathrm{N}u\mathrm{L}_{i-1}{}'. \tag{5.36}$$

The counter decrements to d_0 via Rule (5.24).

In the fifth time-step, d_0 dissolves the existential membrane (if it is not already dissolved). Note that if the object $v_\forall$ is present, then Rule (5.18) could also be

chosen to dissolve the membrane. This is the only instance of non-determinism in the algorithm, however the results of both rules are identical.

$$[\ d_0\ ]_{v_q \mathrm{L}_i} \rightarrow d_0 \tag{5.37}$$

The evaluation membrane is dissolved via Rule (5.26) releasing the NuL$_{i-1}$ object (among all the others) into the next sequence of membranes.

Unused vertex If there is no edge (u, v) in the graph, or where v is not a distance i from s on a path to t then the sequence of membranes are not needed at all. In this case when the counter object reaches zero it dissolves all the three membranes and moves the objects into the next sequence of the computation. As usual if Rule (5.13) dissolves membrane $v\mathrm{L}_i$ if v is an existential membrane. The counter object decrements, $d_4 \rightarrow d_3 \rightarrow d_2 \rightarrow d_1 \rightarrow d_0$, (via Rules (5.12), (5.17), (5.19), (5.24)). At when the object reaches d_0 it dissolves the universal then existential membrane (or just the existential membrane) via Rules (5.37). Then Rule (5.26) dissolves the evaluating membrane with d_0 (which resets) and the objects move into the next sequence of membranes.

5.1.2.4 Evaluating the start vertex

The algorithm begins by evaluating all vertices that are a distance of $i = |V| - 1$ from s on some path to t. Each vertex v at distance i is correctly evaluated and the information (objects) needed to evaluate all vertices at distance $i - 1$ is produced.

We now describe the rules for the case when $i = 1$, i.e. to evaluate the start vertex s. At this stage the rules evolve the output objects `yes` or `no` and the computation halts (as opposed to the YuL$_i$ and NuL$_i$ objects we have already considered).

At this point in the algorithm all paths have been evaluated from their end vertices back to vertex s and length $i = 0$. Recall that in an instance of TAGAP the s vertex is numbered 1. If there is a single Y1L$_0$ object, then the algorithm outputs a `yes` into the environment. This implies that the input instance was an element of TAGAP.

First of all the Y10′ and N1L$_0$′ objects have their primes removed by the Rules (5.9), (5.10), and (5.11). If the vertex s is existential the $1_\forall$L0 membrane is dissolved via Rule (5.13).

The start vertex s is existential. In the second time-step the objects are in the $1_\exists$L$_0$ membrane. Here the no objects wait for the next step via Rule (5.27). If there are any yes objects, they dissolve the membrane and become a `yes` object. There are no rules applicable in the environment so the system halts in an accepting configuration.

$$[\ \mathrm{Y1L}_0\ ]_{1_\exists \mathrm{L}_0} \rightarrow \texttt{yes} \tag{5.38}$$

In the third time-step, if there were no Y10 objects in the membrane, then any available no objects dissolve the membrane and become a `no` object. There are no rules applicable in the environment so the system halts in a rejecting configuration.

$$\left[\ \mathrm{N1L}_0^{W\exists}\ \right]_{1_\exists \mathrm{L}_0} \rightarrow \texttt{no} \tag{5.39}$$

The start vertex s is universal. In the second time-step, if the vertex s is universal, then Rule (5.13) does not dissolve membrane $n_{\forall}\mathrm{L}_0$. Here any yes objects are made wait for the next step via Rule (5.16). If there are any no objects, we dissolve the membrane and produce a `no` object.

$$[\ \mathrm{N1L}_0\ ]_{1_{\forall}\mathrm{L}_0} \rightarrow \texttt{no} \tag{5.40}$$

In the next time-step the `no` object dissolves the parent $1_{\exists}\mathrm{L}_0$ and is released to the environment. There are no rules applicable in the environment so the system halts in a rejecting configuration.

$$[\ \texttt{no}\ ]_{1_{\exists}\mathrm{L}_0} \rightarrow \texttt{no} \tag{5.41}$$

If there were not any no objects to trigger Rule (5.40) then any yes objects present will dissolve the membrane and produce a `yes` object.

$$\left[\ \mathrm{Y1L}_0^{W\forall}\ \right]_{1_{\forall}\mathrm{L}_0} \rightarrow \texttt{yes} \tag{5.42}$$

In the next time-step the `yes` object dissolves the parent $1_{\exists}\mathrm{L}_0$ and is released to the environment. There are no rules applicable in the environment so the system halts in an accepting configuration.

$$[\ \texttt{yes}\ ]_{1_{\exists}\mathrm{L}_0} \rightarrow \texttt{yes} \tag{5.43}$$

If there were no paths leaving s in the graph, The counter decrements during the above steps via Rules (5.12), (5.17), (5.19) from $d_4 \rightarrow d_3 \rightarrow d_2 \rightarrow d_1$. In the fourth time-step, if the object d_1 is in membrane $1_{\forall}\mathrm{L}_0$ or $1_{\exists}\mathrm{L}_0$ then there must be no edges leaving s so $\mathrm{apath}(s,t)$ does not hold.

$$[\ d_1\ ]_{1_q\mathrm{L}_0} \rightarrow \texttt{no} \tag{5.44}$$

If the $1_{\exists}\mathrm{L}_0$ membrane exists then `no` dissolves it via Rule (5.41) and the `no` is released to the environment. There are no rules applicable in the environment so the system halts in a rejecting configuration.

5.2 P upper-bound for systems with symmetric division

In this section we prove a P upper-bound on families of a type of active membrane systems without charges, $\mathcal{AM}^0_{+d,-a}$. In a $\mathcal{AM}^0_{+d,-a}$ system only the following types of rules are permitted: evolution (a), communication-in (b), communication-out (c), dissolution (d), and symmetric elementary division ($\mathrm{e_s}$). Specifically we show that any problem solvable by a semi-uniform family (constructable in polynomial time) of $\mathcal{AM}^0_{+d,-a}$ systems is also solvable on a polynomial time Turing machine.

Theorem: 5.5. (H)–$\mathsf{PMC}^*_{\mathcal{AM}^0_{+d,-a}} \subseteq \mathsf{P}$ *where* $\mathsf{H} \in \mathsf{FP}$

Via Observation 2.13, this result also holds for uniform families.

Corollary: 5.6. (E,F)–$\mathsf{PMC}_{\mathcal{AM}^0_{+d,-a}} \subseteq \mathsf{P}$ *where* $\mathsf{E},\mathsf{F} \in \mathsf{FP}$

The proof of Theorem 5.5 is given by a high-level Turing machine algorithm that simulates the computation steps of Π, an arbitrary $\mathcal{AM}^0_{+d,-a}$ system. If Π_x is a member of a polynomially semi-uniform family of $\mathcal{AM}^0_{+d,-a}$ systems $\mathbf{\Pi}$ that recognises the problem X, our simulation if Π_x operates using space and time polynomial in n, where n is the size of a problem instance $x \in X$.

We begin with an important definition followed by a high-level view of the algorithm. In Sections 5.2.1 and 5.2.2 the algorithm details are examined.

Definition: 5.7 (Equivalence class of membranes)**.** *An equivalence class of membranes (denoted κ) is a set of membranes such that, in relation to a configuration $\mathcal{C}$ of a membrane system, each membrane in the set has: the same parent p, the same label h, an identical multiset of objects m, and exactly the same set of child membranes C. More formally $\exists p, h, m, C$ such that*

$$\kappa = \Big\{ i \in V_\mu \mid (p,i) \in E_\mu \wedge L(i) = h \wedge M(i) = m \wedge \{c \in V_\mu \mid (i,c) \in E_\mu\} = C \Big\}$$

Remark 5.8. *With this definition, no equivalence class κ of $|\kappa| > 1$ can contain a membrane with children.*

At a high level the data structure (see Section 5.2.1) used by the simulation algorithm stores the membrane configuration at each time-step of a valid computation. To conserve space complexity our algorithm does not explicitly store all membranes in a system, instead it compresses the information into equivalence classes. Each stored equivalence class `k` contains: the number ($|\kappa|$ in binary) of membranes in the class, a reference to each of the distinct object types in those membranes, and the number (in binary) of copies of that object type. By storing the quantities of objects and membranes in binary the algorithm needs only polynomial space to store the potentially exponential numbers of both generated by type (a) and ($\mathrm{e_s}$) rules.

While it is possible for a computation of an $\mathcal{AM}^0_{+d,-a}$ system to need an exponential number of equivalence classes (and thus exponential space in our simulating machine), our analysis guarantees that there is another, equally valid, computation for the same system that uses at most a polynomial number of equivalence classes. Since the system is confluent (see Section 2.3), this "cheaper" computation path is just as valid to follow as any alternative path. Our algorithm finds this path in deterministic polynomial time by applying the observation that if a rule is applicable for a single membrane in an equivalence class, then it is equally applicable to all members of that class. Via our algorithm, after a single time-step, the increase in the number of equivalence classes is never greater than $|H||O|$, the product of the number of membrane labels in the system and the number of object types in the system.

In Section 5.2.2 we prove that by using our algorithm:

- Type (a) rules do not increase the number of equivalence classes since the rule has the same effect on each membrane of a given equivalence class.
- Type (b) rules increase the number of equivalence classes by at most $|H||O|$ in a time-step.
- Type (c) rules do not increase the number of equivalence classes since the involved membranes in an equivalence class all eject the same type of object and remain equivalent (the receiving, parent equivalence class contains a single membrane).
- Type (d) rules do not increase the number of equivalence classes since the rule is applied to all membranes in the equivalence class. The contents and child membranes are transfered to the parent equivalence class (of size one).
- Type ($\mathrm{e_s}$) rules do not increase the number of equivalence classes, the algorithm simply doubles the number of membranes in the affected equivalence class.

5.2.1 Algorithm data structure

Our algorithm uses a number of binary registers that is a polynomial of the length n of the input. The length of each register is also bounded by a polynomial of n.

The data structure is used to store an entire configuration of the membrane system. Given an initial configuration $\mathcal{C}_0$ of an $\mathcal{AM}^0_{+d,-a}$ membrane system (guaranteed to be poly(n) in size) we store an equivalence class for each initial membrane in the system. The algorithm increases the number of equivalence classes as required throughout the computation.

An equivalence class, κ, is stored in the following manner by the algorithm, this encoding of an equivalence class is referred to as `k`.

- The register `label` stores the label of the equivalence class and is an element of the set H (see Definition 2.1). The size of register `label` is fixed for all configurations and is bounded by some polynomial of n.
- The register `parent` stores a reference to the equivalence class that contains this membrane (always a single membrane). This value is bounded by the polynomial depth of the membrane structure (note the depth of the membrane structure may not increase during a computation).
- The `children` register references all equivalence classes that are immediate children of this membrane. Its size is bounded by some polynomial of n via Theorem 5.9.
- The register `copies` stores the number of membranes in the equivalence class. In the worst case, the number that is stored in `copies` doubles at each time-step (due to type ($\mathrm{e_s}$) rules). Since we store this number in binary the register length remains some polynomial of n.
- The register `used` is a Boolean and records if the membranes of this equivalence class have been used by a rule in this time-step.

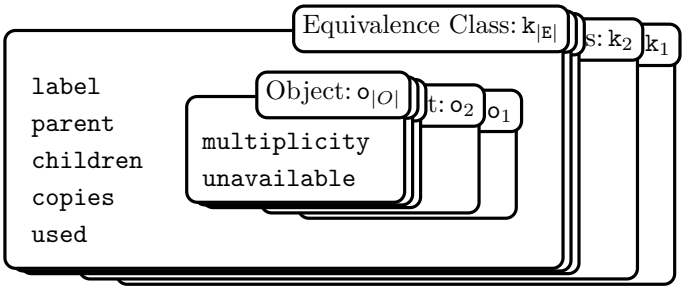

Figure 5.5: A representation of the data-structure described in Section 5.2.1. There is a number of equivalence classes each of which has an object register for each object type in O.

Each membrane in an equivalence class contains an identical multiset of objects. This multiset is stored in `k` in $|O|$ different registers. For each object type $o \in O$, the algorithm stores:

- The register `o` represents the type of the object (that is which element of O). Throughout the computation, the size of the set O is fixed so this register does not grow beyond its initial size.
- The `multiplicity` register is the number of copies of the object `o` in a membrane of this equivalence class. Type (a) rules may cause an exponential growth in the number of objects, however, `o` is stored using binary so the register length remains some polynomial in n.
- The register `unavailable` stores the number of `o` objects that have been used by rules this time-step. It is always the case that `unavailable` $\leq$ `multiplicity` for each object type.

5.2.2 Simulation algorithm

Before proceeding to the algorithm we remind the reader that we are simulating a family of *confluent* (see Section 2.3) recogniser active membrane systems. So given a configuration of a recogniser membrane system, after t time-steps we could be in any one of a large number of possible configurations due to non-determinism in the choice of rules and objects. Confluence means that all computations are guaranteed to all halt with the same result (either all accepting or all rejecting) so when designing an algorithm to simulate such a system we are free to choose which computation is most convenient for us.

Theorem 5.9 asserts that given an $\mathcal{AM}^0_{+d,-a}$ system (sized polynomial in n) there exists a computation such that each configuration in the computation can be represented fully using a polynomial number of equivalence classes. This is shown by proving that there is a computation path where the application of each rule type (a)

to (e_s), in a single time-step, leads to at most an additive polynomial increase in the number of equivalence classes.

In this section we prove Theorem 5.9 with an algorithm that when given $\mathcal{C}$, a configuration of an $\mathcal{AM}^0_{+d,-a}$ system, accepts if the system halts in an accepting configuration and rejects if the system halts in a rejecting configuration. This proof only considers systems constructed in time polynomial in n, hence the size of all sets (including O and H) in Π are limited to be elements of poly(n). We let K_x be a set of equivalence classes that contains every membrane in a configuration $\mathcal{C}_x$ of a $\mathcal{AM}^0_{+d,-a}$ membrane system.

Theorem: 5.9. *Given Π_x, an $\mathcal{AM}^0_{+d,-a}$ system that solves some instance x of a problem, where Π_x is of size polynomial in $n = |x|$, and where Π_x has $|K_0| = |V_\mu|$ equivalence classes, $|O|$ distinct object types, and $|H|$ labels. Then there is a computation starting with the initial configuration $\mathcal{C}_0$ of Π_x such that at time $t \in$ poly(n) the number of equivalence classes is $|K_t| = \mathcal{O}(|K_0| + t|H||O|)$ which is a polynomial of n.*

Proof. Base case: There can be at most a polynomial number of membranes in the initial configuration thus $|K_0| \in$ poly(n). Each of Lemmas 5.10 to 5.14 gives an upper-bound on the increase in the number of equivalence classes after one time-step for rule types (a) to (e_s), respectively. Lemma 5.11 has an additive increase of $|H||O|$ and the other four lemmas have an increase of 0. Thus at time 1 there is a computation path where the number of equivalence classes is $|K_1| \leq |K_0| + |H||O|$.

Inductive step: Assume that $|K_i|$, the number of equivalence classes at time i, is polynomial in n. Then, via Lemmas 5.10 to 5.14, there exists a computation path where $|K_{i+1}| \leq |K_0| + i|H||O|$.

After t time-steps we have $|K_t| = \mathcal{O}(|K_0| + t|H||O|)$, which is polynomial in n, if t is. □

We now give the algorithm that simulates the computation of any membrane system of the class $\mathcal{AM}^0_{+d,-a}$ in time polynomial to the input length n. The algorithm operates on any valid initial configuration and successively simulates the developmental rules of the membrane system. It takes as input a configuration of a $\mathcal{AM}^0_{+d,-a}$ system Π of size polynomial in n, which is then encoded into the registers of the simulating device in polynomial time as a tree of equivalence classes. The algorithm then iterates over the equivalence classes. At each iteration all available rules are applied, this simulates a single time-step of the membrane system's computation. The outer while loop terminates when there are no more rules applicable, this indicates that the computation has halted.

In line 1 the algorithm makes a depth first ordering of μ which it then loops over $\mathcal{O}(|V_\mu|)$ times. To construct a depth first ordering takes $\mathcal{O}(|V_\mu| + |E_\mu|)$ steps [70] and ensures that the rules are applied to the children of a membrane before its parents. To store an equivalence class we count the number of objects in each multiset, this is dominated by the size of the biggest multiset $\mathcal{O}(\max_m(M))$ (where $\max_m(M) = \max(\{|M(i)| \mid \forall i \in V_\mu\})$. We store the references to the child membranes of each equivalence class, in the worst case every other membrane is a child of this one, $\mathcal{O}(|V_\mu|)$.

Algorithm 1 Simulate membrane system($\mathcal{C}_0$ of Π)

```
1: for each membrane i in the depth first ordering of μ store an k in K^DF
2:    for each object o store Q_{M(i)}(o) in the multiplicity of o
3:    set the children of k to be references to i's child membranes.
4: end for
5: while there are rules to apply
6:    for each k in K^DF
7:       for each object type with multiplicity − unavailable > 0 in k
8:          for each r in the set of rules for this label & object
9:             add r to A if it is of type (a)
10:            add r to B if it is of type (b)
11:            add r to C if it is of type (c)
12:            add r to D if it is of type (d)
13:            add r to E if it is of type (e_s)
14:         end for
15:      end for
16:      apply rules in A to k
17:      apply rules in B to k
18:      for each r in C, D, E
19:         try to apply r to k                    //remember at most 1 will be applied
20:      end for
21:   end for
22: end while
23: if object yes is in the environment then accept
24: else if object no is in the environment then reject
25: else reject
```

This gives the loop in line 1 a running time of $\mathcal{O}((|V_\mu|+|E_\mu|)+|V_\mu|(\max_m(M)+|V_\mu|))$. The stored list of depth first ordered equivalence classes is denoted $\mathtt{K}^{\mathrm{DF}}$.

The initial configuration is now stored in the registers, the algorithm loops t times where t is the number of time-steps the membrane system runs until halting. For each time-step the algorithm iterates through the depth first ordering of equivalence classes, $\mathtt{K}^{\mathrm{DF}}$ (line 6) $\mathcal{O}(|K_t|)$. Note that the membrane structure cannot be affected by rules such that the depth first ordering on the list of equivalence classes is disrupted. Line 7 iterates over all the objects ($\mathcal{O}(|O|)$) in this equivalence class and checks if there are any available objects of that type. If there are, line 8 iterates through the set of rules ($\mathcal{O}(|R|)$) looking for those triggered by object `o` and `label`. If it finds such a rule, it is added to a set of rules according to its type (we check if it can be applied later).

Testing if a rule is applicable to `k` is straightforward and consists of checking if the equivalence class has the correct label and contains an available object of the triggering type ($\mathcal{O}(|O|)$). Communication-in rules (type (b), line 10) are different, we must check the labels of the children of `k` (which takes $\mathcal{O}(|K_t|)$ time). So the total time needed to

build the list of applicable rules for an equivalence class is $\mathcal{O}(|O| \cdot |R|(|O| + |K_t|))$.

At line 16, Algorithm 2 is called to apply all rules of type (a), the time complexity mentioned below is $\mathcal{O}(|R| \max_m(R_M))$. Then the rules of type (b) are applied using Algorithm 3 with time complexity $\mathcal{O}(|H| \cdot |K_t| \cdot |R| \cdot |O|)$.

In line 18 the algorithm considers rules of type (c), (d), and (e_s), only one of which can be applied to an equivalence class in a time-step. Of the three Algorithms 4, 5, and 6 to simulate the application of these rules, Algorithm 5 (type (d)) is the worst case taking $\mathcal{O}(|O| + |K_t|)$ time.

Thus, the total running time of our simulation algorithm is:

$$\begin{aligned}\mathcal{O}\Big(&|V_\mu| + |E_\mu| + |V_\mu|(\max_m(M) + |V_\mu|) \\ &+ t \cdot |K_t|(|O| \cdot |R|(|O| + |K_t|) \\ &\quad + |R| \max_m(R_M) \\ &\quad + |H| \cdot |K_t| \cdot |R| \cdot |O| \\ &\quad + |O| + |K_t|)\Big)\end{aligned}$$

This algorithm takes a membrane system as input and iterates through its configurations until the membrane system halts. The algorithm tries to apply each rule from the set of all applicable rules, if a membrane has been used by another rule or if the object needed to trigger the rule has been used in a previous rule, the rule fails to be applied. Thus, the final list of rules that are actually applied in a computation step is maximal (see Definition 2.4) for this configuration since it is impossible to apply another rule to the system.

We now explain Algorithm 2 which applies rules of type (a). This algorithm starts

Algorithm 2 Apply a set of rules A of type (a) to equivalence class `k`

```
1: for each rule r = (a, o, h, m) in A
2:    let available be the multiplicity of o minus the o's unavailable
3:    set the multiplicity of o equal to the number of o's unavailable
4:    for each object u in m                    //the objects created by the rule r
5:       increase the multiplicity of u by the number of o's available
6:       increase the number of u's unavailable by o's available
7:    end for
8: end for
```

with a set of type (a) rules, A, and tries to apply them one by one, ($\mathcal{O}(|R|)$). For each rule, it calculates the number of objects available to be used by the rule. Since all unused objects of type o will be used by the application of this rule, the algorithm sets (in lines 2 and 3 the quantity of unused objects to be equal to the number unaffected by the rule. Next the algorithm loops (line 4) over the multiset of objects in the rule $\mathcal{O}(\max_m(R_M))$ where R_M is the set of multisets in rules of type (a). The quantity of each object u in the multiset of m is increased by the number of available o objects in the equivalence class. If an object appears x times in m, its counter is increased x times in line 6. The total time complexity for this function is $\mathcal{O}(|R| \max_m(R_M))$.

We now prove that the number of equivalence classes does not increase due to the application of Algorithm 2 and thus that the space complexity of the algorithm remains a polynomial of n.

Lemma: 5.10 (Rules of type (a))**.** *Given a configuration $\mathcal{C}_t$ of a $\mathcal{AM}^0_{+d,-a}$ system with $|K_t|$ equivalence classes. After a single time-step, where only rules of type (a) (object evolution) are applied via Algorithm 1, there exists a configuration $\mathcal{C}_{t+1}$ such that $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$ and $\mathcal{C}_{t+1}$ has $|K_t|$ equivalence classes.*

Proof. If a type (a) rule is applicable to an object in a membrane in an equivalence class, then the rule is also applicable in exactly the same way to all membranes in that class. Due to non-determinism in the choice of rules and objects, it could be the case that the membranes in the equivalence class evolve differently. However we apply the type (a) rules to objects using Algorithm 2 which chooses a valid computation path where all membranes in an equivalence class evolve identically in the time-step. Thus there is a computation step $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$ where there is no increase in the number of equivalence classes. □

Now we describe the most awkward part of this simulation, rules of type (b). Observe that type (b) rules have the potential to increase the number of equivalence classes in one time-step by sending distinct object types into membranes in the same equivalence class. For example, if objects of type o_1 are sent into some of the membranes in an equivalence class, and o_2 objects are sent into the remainder, then we increase the number of equivalence classes by 1. Algorithm 3 operates by applying rules of type (b) in set B to the children of an equivalence class (line 2, $\mathcal{O}(|K_t|)$) in order of their labels (line 1, $\mathcal{O}(|H|)$) and checks if the equivalence class has been used by other rules already in this time-step. In line 3 we loop through the rules ($\mathcal{O}(|R|)$) in the set B and try to apply those which communicate into membranes with the label l we are currently considering. If there is more than one rule that communicates the same object type into the same label, the first rule uses all the triggering objects or all the destination membranes so the other rules cannot be applied. We calculate the number of objects that this rule can communicate into a membrane (line 4). If the number of objects is greater than the number of membranes in the child membrane, no new equivalence classes are created. However if there are less objects than membranes in the class, some membranes in an equivalence class receive an object that others do not so we must split the equivalence class in two (line 9). The creation of the new equivalence class is straightforward. The total running time of Algorithm 3 is $\mathcal{O}(|H| \cdot |K_t| \cdot |R| \cdot |O|)$. However we must take care that the number of new classes does not grow too quickly, this is addressed in the following lemma.

The next proof seems quite involved, however it can be summarised as follows: if the objects to be communicated into a set of equivalence class of the same label are sorted lexicographically, then the number of equivalence classes at the end of the step can increase by at most the number of object types communicated in. As an analogy we imagine some x pieces of string of various lengths, it is clear that if we make y cuts to the strings (it is possible to miss the strings), there are at most $x + y$ pieces of string.

Algorithm 3 Apply a set of rules B of type (b) to equivalence class `k`

```
1: for each label l ∈ H in the system
2:   for each child with label l in the children of k that is not used
3:     for each rule r = (b, o, h, u) in B where l = h
4:       let available be the multiplicity − unavailable of o in k.
5:       if number of o's available ≥ number of copies of child then
6:         increase u's multiplicity in child by o's available
7:         increase number of u's unavailable in child by o's available
8:         decrement multiplicity of o in k by o's available
9:       else if number of o's available is < copies of child then
10:        create a new equivalence class and add after child in K^DF
11:        add new to the children of k and set new's parent to be k
12:        set new's label to be the same as child's
13:        for each o in child
14:          copy all the values of o from child to new
15:        end for
16:        increase the multiplicity of u in new by o's available
17:        increase the unavailable copies of u in new by o's available
18:        set the number of copies of new to be available
19:        set the number of copies of child to be copies − available
20:        set the new equivalence class used
21:        decrease the multiplicity of o by the number of o's available
22:      end if
23:    end for
24:  end for
25: end for
```

Lemma: 5.11 (Rules of type (b))**.** *Given a configuration $\mathcal{C}_t$ of a $\mathcal{AM}^0_{+d,-a}$ system Π with $|K_t|$ equivalence classes. Let $|H|$ be the number of membrane labels system Π. Let $|O|$ be the number of distinct object types in Π. After a single time-step, where only rules of type (b) (incoming objects) are applied, there exists a configuration $\mathcal{C}_{t+1}$ such that $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$ and $\mathcal{C}_{t+1}$ has $\leq |K_t| + |H||O|$ equivalence classes.*

Proof. First we consider the case where the algorithm is operating on an equivalence class whose child equivalence classes are all of size 1. In this case the type (b) communication rules are applied without any increase to the number of equivalence classes.

The remainder of the proof is concerned with the other case, where the parent membrane contains a non-zero number of equivalence classes of elementary membranes.

We now prove that for each membrane label, using Algorithm 3, that there is at most an increase of $|O|$ equivalence classes per time-step. Let $O' \subseteq O$ be the set object types being communicated into a set of equivalence classes of the same label, in a time-step.

Base case: We now show that when $i = 1$ (the size of the set of objects to

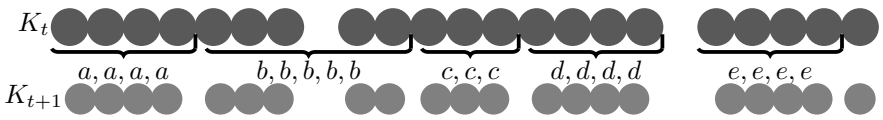

Figure 5.6: An illustration of the effects of Algorithm 3 and proof of Lemma 5.11. A set of equivalence classes, K_t, contains 3 classes of sizes 7, 9, and 5 (of label h). The parent membrane contains the objects $a, a, a, a, b, b, b, b, b, c, c, c, d, d, d, d, e, e, e, e$. The rules $a[\,]_h \to [\,a\,]_h$, $b[\,]_h \to [\,b\,]_h$, $c[\,]_h \to [\,c\,]_h$, $d[\,]_h \to [\,d\,]_h$, $e[\,]_h \to [\,e\,]_h$ are applied. The resulting set of equivalence classes K_{t+1} contains classes of size 4,3,2,3,4,4, and 1.

communicate $|O'|$), there is at most an increase of 1 to the number of equivalence classes (with membrane label $h \in H$). Clearly, if we communicate x objects into x membranes all in the same equivalence class, there is no increase in the number of classes. However, if we communicate $y < x$ objects of type o into x membranes (all in the same equivalence class), then $x - y$ membranes did not receive an object. This causes the algorithm to create a new equivalence class of size y, an increase of 1 equivalence class for 1 object type. Note that the remainder of the old class (of size $x - y$) still exists.

Inductive step: Assume $|K^h_{t+1}| = |K^h_t| + i$, that the number of new equivalence classes (of label h) in a time-step is i where i is the number of distinct object types communicated into the membrane.

We now give the case where $|O'| = i + 1$ object types are communicated. We observe the algorithm in mid computation, at the point where all available objects of types $o_1, \ldots, o_i \in O'$ have been communicated into the equivalence classes. These "used" equivalence classes and membranes can no longer receive objects in this time-step and so to analyse the communication of the objects of the final type o_{i+1} we need only focus on the remaining unused equivalence classes. Note that this situation is analogous to the base case; we are communicating in objects of one distinct object type into some equivalence classes. Hence we see that the increase in equivalence classes for $i + 1$ object types is equal to the increase for i types, plus 1. By iterating backwards through the possible sizes of $|O'|$ we reach the base case of $i = 1$ and see that the maximum increase in the number $|O'|$ of equivalence classes is equal to the number of object types being communicated.

Thus given some equivalence classes K^h_t (at time-step t, all with label h), the number of classes at time-step $t + 1$ is $|K^h_{t+1}| \leq |K^h_t| + |O|$. This same process occurs for each membrane label $h \in H$ for each time-step. Thus over the whole time of the system, the algorithm for type (b) rules produces no more than $|K_0| + t|O||H|$ equivalence classes. □

We now discuss rules of type (c). This algorithm has no major time complexity overhead, we show that it does not cause an increase in the number of equivalence classes.

Algorithm 4 Apply a rule $r = (\mathrm{c}, o, h, \{u\})$ of type (c) to equivalence class `k`

```
1: if the membranes of k have not been used then
2:    let available be o's multiplicity less the number of o's unavailable
3:    if the number of o's available is greater than 0 then
4:       decrement the number of multiplicity of o in k
5:       increase the multiplicity of u by available in the parent of k
6:       increase number of u's unavailable by o's available in parent of k
7:       set the membranes of k to be used
8:    end if
9: end if
```

Lemma: 5.12 (Rules of type (c))**.** *Given a configuration $\mathcal{C}_t$ of a $\mathcal{AM}^0_{+d,-a}$ system with $|K_t|$ equivalence classes. After a single time-step, where only rules of type (c) (communication out) are applied, there exists a configuration $\mathcal{C}_{t+1}$ such that $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$ and $\mathcal{C}_{t+1}$ has $|K_t|$ equivalence classes.*

Proof. If a type (c) rule is applicable to an object in a membrane in equivalence class κ_k, then the rule is also applicable in exactly the same way to all membranes in κ_k. Due to non-determinism in the choice of rules and objects it could be the case that different membranes in κ_k eject different symbols. However we choose a computation path such that all membranes in an equivalence class evolve identically (each membrane ejects the same symbol), and so no new equivalence classes are created from κ_k. The single parent of all the membranes in κ_k is in an equivalence class κ_j which (via Remark 5.8) contains exactly one membrane and so no new equivalence classes are created.

Thus there is a computation path $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$ where there is no increase in the number of equivalence classes. □

We saw that $\mathcal{AM}^0_{-d}$ systems (without dissolution rules) characterise NL in semi-uniform families (Section 3.1) and that the power of uniform families is dependant on the encoding function (when above AC^0) (see Section 4.1). But, in Sections 5.1 we saw that the addition of dissolution rules allows uniform and semi-uniform families of $\mathcal{AM}^0_{+d}$ to solve P-complete problems, hence one would expect dissolution to be a difficult rule to simulate. However, storing membranes in equivalence classes makes dissolution a relatively easy rule to simulate. Algorithm 5 is quite straightforward, line 6 moves all the objects to the parent membrane, its time complexity is $\mathcal{O}(|O|)$. To adjust the parent pointer of each child equivalence class in line 10 we need at most $\mathcal{O}(|V_\mu|)$ time. The total worst case running time is $\mathcal{O}(|O| + |V_\mu|)$. We now show that rules of type (d) do not increase the number of equivalence classes. In fact, such rules can decrease the number of equivalence classes.

Lemma: 5.13 (Rules of type (d))**.** *Given a configuration $\mathcal{C}_t$ of a $\mathcal{AM}^0_{+d,-a}$ system with $|K_t|$ equivalence classes. After a single time-step, where only rules of type (d) (membrane dissolution) are applied then for all $\mathcal{C}_{t+1}$, such that $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$, $\mathcal{C}_{t+1}$ has $\leq |K_t|$ equivalence classes.*

Algorithm 5 Apply a rule $r = (\mathrm{d}, o, h, \{u\})$ of type (d) to equivalence class `k`

if the equivalence class membranes have not been `used` **then**
  if o's `multiplicity` is greater than o's `unavailable` **then**
    decrement the `multiplicity` of o in `k`
    increment the `multiplicity` of u in `k`
    increment the `unavailable` copies of u in `k`
    for each `o` in `k`
      add `o`'s `multiplicity` $\times$ `k`'s `copies` to `o`'s `multiplicity` in `parent`
      add `o`'s `multiplicity` $\times$ `k`'s `copies` to `o`'s `unavailable` in `parent`
    end for
    for all `children` of `k`
      set the `parent` of the child membrane to be the `parent` of `k`
    end for
    add the `children` of `k` to the `children` of `parent`
    remove `k` from the set $\mathtt{K}^{\mathrm{DF}}$
  end if
end if

Proof. If there is at least one type (d) rule that is applicable to an object and a membrane in equivalence class κ_k, then that rule is applicable in exactly the same way to all membranes in κ_k. We chose the computation path where this is the case. Thus a whole equivalence class of membranes dissolve together, reducing the number of equivalence classes in total. The single parent of all the membranes in κ_k is in an equivalence class κ_j which, by Remark 5.8, contains exactly one membrane and so no new equivalence classes are created from the dissolution of all membranes in κ_j. Thus for all $\mathcal{C}_{t+1}$, where $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$, there is no increase in the number of equivalence classes. □

We now provide an algorithm for rules of type $(\mathrm{e_s})$.

Algorithm 6 Apply rule $r = (\mathrm{e_s}, o, h, \{u, u\})$ of type $(\mathrm{e_s})$ to equivalence class `k`

if the membranes of `k` have not been `used` **then**
  decrement the `multiplicity` of o in `k`
  increment the `multiplicity` of u in `k`
  increment the number of us `unavailable` in `k`
  double the number of `copies` of `k`
  set the membranes of `k` to be `used`
end if

This algorithm only involves incrementing, decrementing and doubling counters and so has a trivial time complexity. We now show that despite increasing the number of membranes, the number of equivalence classes remains unchanged.

Lemma: 5.14 (Rules of type $(\mathrm{e_s})$)**.** *Given a configuration $\mathcal{C}_t$ of a $\mathcal{AM}^0_{+d,-a}$ system with $|K_t|$ equivalence classes. After a single time-step, where only rules of type $(\mathrm{e_s})$*

(symmetric elementary membrane division) are applied, there exists a configuration $\mathcal{C}_{t+1}$ *such that* $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$ *and* $\mathcal{C}_{t+1}$ *has* $\leq |K_t|$ *equivalence classes.*

Proof. If a type (e_s) rule is applicable to an object and membrane in equivalence class κ_k, then the rule is also applicable in exactly the same way to all membranes in κ_k. We chose the computation path where all membranes in an equivalence class evolve identically in a time-step, each membrane in κ_k divides using the same rule. The number of membranes in κ_k doubles, but since each new membrane is identical, no new equivalence classes are created from κ_k.

Thus there is a computation path $\mathcal{C}_t \vdash \mathcal{C}_{t+1}$ where there is no increase in the number of equivalence classes. □

5.3 Discussion

In Chapters 3 and 4 when some classes of $\mathcal{AM}^0$ systems are restricted from being P-(semi-)uniform to being AC^0-(semi-)uniform they no longer characterise P. We also saw that uniform and semi-uniform families do not always have equal power. In this chapter however we have shown that for at least one variety of $\mathcal{AM}^0$, AC^0-uniform and semi-uniform families both characterise P.

Previous results relied on FP computable (semi-)uniformity conditions to prove P lower-bounds [30, 54], this is somewhat unsatisfying as it says nothing about the complexity of the system itself. We have given the first concrete P lower-bound for active membrane systems, an AC^0-uniform membrane family to solve a P-complete problem. In fact, our characterisation holds for all uniformity conditions contained in and including P (see Figure 5.1).

Each membrane in the uniform family uses only *evolution* and *dissolution* rules to solve the P-complete problem. Since only evolution rules are necessary to solve problems in NL (see Chapter 3) we immediately wonder what role dissolution plays in increasing the complexity. Can uniform families of membranes using only dissolution rules solve all of P alone or is the interaction with evolution rules necessary.

We also presented a P upper-bound for (semi-)uniform families of $\mathcal{AM}^0_{+d,-a}$ systems. These systems use a slight restriction of weak elementary division so we have given a partial result for the P-Conjecture. This result is also the first $\mathcal{AM}^0$ system with dissolution rules to have a P rather than PSPACE upper-bound.

The restriction we make to the system is to replace division rules of type (e) with symmetric division (e_s) (so called due to its resemblance to symmetric mitosis in biological cells). A system with symmetric weak elementary division (e_s) can easily generate an exponential number of membranes and objects in polynomial time. However, this result shows that the symmetric rule restricts the system so that it does not create exponentially many different membranes on all computation paths. This result can be interpreted in two ways, first that any future computers built using symmetric division and without charges will not be able to solve NP-complete problems or any problem outside of P. For those designing parallel simulators of membrane systems this result means that any system that uses both dissolution and evolution rules will (unless $\mathsf{NC} = \mathsf{P}$) be extremely difficult to parallelise efficiently.

Our upper-bound in Section 5.2 cannot be applied naively for asymmetric weak elementary division (e_w). For example the rules

$$[\,x_0\,] \to [\,0_0\,][\,1_0\,],$$
$$[\,x_1\,] \to [\,0_1\,][\,1_1\,],$$
$$[\,x_2\,] \to [\,0_2\,][\,1_2\,],$$

will when given x_1, x_2, x_3 as input, generate 2^3 membranes that (without some non trivial extension to our proof) can only be stored in 2^3 equivalence classes.

By extending the definition of equivalence class in a recursive manner so that the child membranes of two membranes x and y are equal sets of equivalence classes we may be able to extend the technique of this proof to systems with symmetric non-elementary division rules. $\mathcal{AM}^0$ systems with strong non-elementary rules are known to solve PSPACE [3] and those with weak non-elementary division [21] are known to solve NP $\cup$ coNP so if systems with symmetric versions of these rules characterised P it would be very interesting.

Chapter 6

Conclusions

Membrane computers are at the early stages of experimental implementation. Recent progress has been made in the laboratory [25], in custom silicon hardware [46], and in software running on massively parallel devices [13]. By studying the computational complexity of cellular operations we provide information for researchers as to which features are the most important for inclusion in their implementations of membrane systems.

Our results have particular relevance for those seeking to exploit the natural parallelism inherent in cellular systems for computation. Previously all* complexity results for membrane systems were shown exclusively using polynomial time uniformity. This was a reasonable choice since the focus of the research was to discover which variants could solve NP-complete problems and which could not. However, due to the way the input is encoded in membrane systems, polynomial time uniformity prevents membrane systems from characterising any class of problems contained in P. This does is not much of an obstacle until we wish to discuss the class P and below. When we wish (using a physical device) to solve problems in a parallel manner, being restricted to classes that contain P is not an ideal situation since P-complete problems are thought to be intrinsically sequential. That is, no significant speed-up is achieved (unless NC = P) when we polynomially increase the number of processors working on a P-complete problem [27].

But what about the Parallel Computation Thesis [26]? Parallel machines can decide problems in PSPACE exponentially faster than sequential machines. However, to do so they require an exponential number of processors. When it comes to implementing such a device, providing an exponential number of processors is a major set back. Initially many people dreamed of "growing" processors using the native ability of cells to duplicate themselves, however a brute force approach to such intractable problems with cellular computers will run into the same difficulty. For example, to use brute force to solve the Traveling Salesman problem (an NP-complete problem) with 125 cities using *Escherichia coli* bacteria as our implementation medium would require 2^{125} individual *E. coli* with a mass equal to that of the moon†. Time, however, is

*All but Obtułowicz [47].

†Assuming the mass of the Moon is 7.347×10^{22}kg and the mass of an *E. coli* is 1×10^{-15}kg

on our side: given perfect growing conditions, a single *E. Coli* can divide 125 times in just over 2 days. Then we just need to locate the single *E. coli* that contains the correct solution. Thus any naive approach to solving intractable problems with cellular computers will have the same problems as any other classical computing device.

So, for practical parallel implementations of membrane systems perhaps it is more rewarding to focus on problems in NC, where a modest *polynomial* amount of processors provides the desired speed-up. But, using P uniformity, it seems that all active membrane systems either characterise P or PSPACE. However, we had no success trying to solve P-complete problems with systems without dissolution rules where were said to characterise P [30]. We found the reason was that these systems were actually weaker than P and the uniformity function was artificially inflating the set of problems solvable by the system. It turned out that semi-uniform families of these systems actually characterised NL. For those building efficient parallel simulators, this result is quite useful since $\mathsf{NL} \subseteq \mathsf{NC}^2$, so simulations of these systems can be parallelized. This result is also useful for sequential simulators since the problems in NL are solvable using very little memory ($\mathcal{O}(\log^2 n)$) on a deterministic polynomial time Turing machine [63].

Our P lower-bound in Section 5.1 (the first genuine P lower-bound for active membranes systems) is important for those attempting to write fast parallel simulators for membrane systems [13]. We gave a uniform family to solve a P-complete problem which uses only evolution and dissolution rules, this implies (along with other known results [74, 3, 21]) that any system combining dissolution rules with some other rule type can solve P-compete problems and thus cannot be simulated with better than linear time overhead (if $\mathsf{NC} \subsetneq \mathsf{P}$).

We presented in Chapter 4 an interesting gap in the set of problems solved by uniform and semi-uniform families of $\mathcal{AM}^0_{-d}$ systems. For those seeking to implement membrane systems biologically we showed that uniform families of $\mathcal{AM}^0_{-d}$ systems are severely crippled and cannot be used to solve all problem instances of a certain size, in fact they are unable to solve any (interesting) problem independently of the device that constructed the system. However, the same system can solve NL complete problems if we design devices that solve only a single problem instance. This result proves something general about families of finite devices that is independent of particular formalisms and can be applied to other computational models besides membrane systems. Besides membrane systems and circuits, some other models that use notions of uniformity and semi-uniformity include families of neural networks, molecular and DNA computers, tile assembly systems, branching programs and cellular automata [9, 16, 49, 64, 65]. Our results could conceivably be applied to these models.

It is known that active membrane systems without division rules are upper-bounded by P [75]; in Section 5.2 we give the first P upper-bound for active membrane systems with division and dissolution rules. However, this is for a restricted form of division where the resulting membranes are identical; we refer to this as symmetric division. This is a partial result for the P-conjecture which we hope to fully resolve in the near future.

This thesis also argues that in general, membrane computing should use at most

logspace uniformity (has been standard in circuit complexity since its inception) when discussing systems that solve problems in P. Ideally, all systems would be FO-uniform or DLOGTIME-uniform (deterministic log time) [10, 33], and we would use FO projections instead of reductions. However, in this thesis we chose to use AC^0-uniformity and AC^0 reductions because this allows us to focus on the results rather than the reductions. FAC^0 is weak enough for our purposes since $\mathsf{AC}^0 \subsetneq \mathsf{NC}^1 \subseteq \mathsf{NL} \subseteq \mathsf{P}$[24] and it means that with very little effort we can show that some pre-existing lower-bounds still hold. (When giving explicit lower-bounds, most authors use simple mappings and so with only minor tweaks to their mappings we can show that their results hold under AC^0 reductions.)

We now show that an existing PSPACE lower-bound is unaffected by the change for polynomial time uniformity to AC^0 uniformity. We will then discuss some future work and open problems.

6.1 AC^0-uniformity and PSPACE

In Figure 1.1 there is a PSPACE characterisation for P-uniform families using rules of type (f). We now quickly sketch how the same characterisation also appears in Figure 1.2 with its lower-bound unaffected by the restriction to AC^0 uniformity.

Clearly stricter uniformity notions have no affect on the PSPACE upper-bound [67]. The original [7] lower-bound is a P-uniform family to solve QSAT (in conjunctive and "even number of variables" normal forms). The PSPACE-complete problem of QSAT is defined as follows:

Problem: 6.1 (Qualified Boolean Satisfiability (QSAT))**.**
Instance: *Given a boolean expression φ in conjunctive normal form $\varphi = C_1 \wedge C_2 \ldots \wedge C_m$ where $C_i = y_{i,1} \vee \ldots \vee y_{i,l_i}, 1 \leq i \leq m$ where $y_{i,k} \in \{x_j, \neg x_j \mid 1 \leq j \leq n\}, 1 \leq i \leq m, 1 \leq k \leq l_i$.*
Question: *Does that formula $\exists x_1 \forall x_2 \ldots \exists x_{n-1} \forall x_n \ \varphi(x_1, \ldots, x_n)$ evaluate to true?*

The original uniformity condition specified the membrane family using the number $\langle n, m \rangle$. This is a function computed from n, the number of variables in the instance, and m, the number of clauses in the instance. However, the function $\langle n, m \rangle$ is neither computable nor reversible in FAC^0. To generate the family without computing this function, we assume that the input instance of QSAT is in a normal form where $n = m$.

Normal Form: 6.2. *An instance of* QSAT *is in* NVENC normal form *if the number of variables in the instance is equal to the number of clauses.*

Given an instance of QSAT in conjunctive and even-variables normal forms, we can adjust the instance such that the number of variables (n) and clauses (m) are equal as follows.

If $n < m$ we add $m - n$ "junk" variables $\{x_{n+1}, \ldots, x_m\}$ to the instance, the variables are not listed in the clauses and so do not affect the first n bits of the satisfying solution. If the number of variables is now odd then we add another variable $x_m + 1$ and add the tautological clause $(x_1 \vee \neg x_1)$ to φ.

If $n > m$ we add $n - m$ copies of the tautological clause $(x_1 \vee \neg x_1)$ to φ. These clauses always evaluate to true and so do not affect the satisfying assignment to φ.

Our new function f to construct the family takes the length of the instance $1^{|x|}$, $x \in \mathsf{QSAT} \cup \mathsf{coQSAT}$ as input. It is trivial to calculate the number n from $1^{|x|}$, n is then used to generate a membrane system that is identical to the family member indicated by the number $\langle n, n \rangle$ as specified in [7]. The function f is computable in FAC^0: the most complicated aspect involves multiplication by constants (essentially addition) which is known [72] to be in FAC^0.

The input encoding e function maps the variables in the clauses to objects and is straightforward to compute in FAC^0.

Thus AC^0-uniform families of $\mathcal{AM}^0$ using strong elementary division can solve at most PSPACE-complete problems.

6.2 Conjectures and Problems

We collect here the future problems and directions that were mentioned throughout this thesis and some additional conjectures.

1. *What types of model have uniform/semi-uniform gaps?* So far we have only observed a uniform/semi-uniform gap in systems where the semi-uniform family characterises NL. Is this gap a feature of weaker models or can a similar result be shown for more powerful families? The gap may be a consequence of the lack of context sensitivity in some systems.

2. *Can we prove that object multiplicities for all $\mathcal{AM}$ systems are not essential for lower-bound results?* We have shown that it is possible to replace multisets of objects with sets of objects for $\mathcal{AM}^0_{-d}$ systems. Can we do the same for all $\mathcal{AM}$ systems? None of the membranes systems specified in this thesis (including that in Section 6.1) use multiplicities, providing evidence that this is the case. This result would make proving upper-bounds much easier on these systems.

3. *Study the complexity of families where each membrane system in a uniform family is easier to evaluate than it is to construct.* It is unclear how much the function f (when its domain is limited to $\{1\}^*$) can help the resulting systems "cheat" and solve harder problems. To examine this, the encoding function e must be extremely weak, for example be the identity function or computable in FAC^0. We can pose this as $(\mathsf{F}, \mathsf{AC}^0)$–$\mathsf{PMC}_{\mathcal{AM}^0_{-d}} \stackrel{?}{=} \mathsf{AC}^0$ where $\mathsf{F} \in \{\mathsf{P}, \mathsf{PSPACE}, \mathsf{EXP}\}$. This echoes similar open problems in circuit uniformity, for example P-uniform $\mathsf{AC}^0 \stackrel{?}{=}$ FO-uniform AC^0 [8].

4. *Can we characterise the NC hierarchy with active membrane systems?* We have shown that semi-uniform families of $\mathcal{AM}^0_{-d}$ systems characterise NL which is contained in the class NC^2. If active membrane systems are a good model of parallel computation it should be possible to characterise every level of the NC hierarchy. This characterisation my also extend into a characterisation of the polynomial hierarchy between P and PSPACE.

5. *Is there a link between monotone circuit complexity and active membranes?* Can we define a restriction on membrane systems such that they can compute only monotone boolean functions? If this is the case could we possibly prove lower-bounds for problems in terms of membrane systems [58]?

6. *Do (semi-)uniform $\mathcal{AM}^0$ systems using* only *dissolution rules characterise* P*?* We have shown that systems with only type (a) rules characterise NL, while those with (a) and (d) rules characterise P. What role do type (d) rules play in this jump in complexity? If they characterise NL there could be a uniform/semi-uniform gap for this model. If they characterise P, it would improve our lower-bound in Section 5.1.

7. *Resolve the P-conjecture.* We are currently attempting to resolve this conjecture and have another partial solution for systems with weak non-elementary division and dissolution [74] which is not included this thesis.

8. *Symmetric non-elementary division.* It would be interesting to see if families of systems with symmetric non-elementary division rules are weaker than their asymmetric equivalents (rules of type ($\mathrm{e_w}$) and (f)) which can solve problems in NP [21] and PSPACE [7]. Perhaps the proof we gave in Section 5.2 could be extended to solve such problems if we extended the definition of equivalence class so that identical membranes with identical subtrees of child membranes are part of the same equivalence class.

9. *Do uniform families of general recogniser $\mathcal{AM}^0_{-d}$ a uniform/semi-uniform gap?* Our proof for standard recogniser systems does not trivially extend to general recogniser systems. It would be extremely interesting if uniform families of such systems could solve NL complete problems, however it seems unlikely to be the case.

Bibliography

[1] Leonard Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, 1994.

[2] Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell*. Garland Science, New York, fourth edition, 2002.

[3] Artiom Alhazov and Mario de Jesús Pérez-Jiménez. Uniform solution to QSAT using polarizationless active membranes. In Miguel Angel Gutiérrez-Naranjo, Gheorghe Paun, Agustín Riscos-Núñez, and Francisco José Romero-Campero, editors, *Fourth Brainstorming Week on Membrane Computing, Sevilla, January 30 - February 3,2006. Volume I*, pages 29–40. Fénix Editora, 2006.

[4] Artiom Alhazov, Carlos Martín-Vide, and Linqiang Pan. Solving a PSPACE-complete problem by recognizing P Systems with restricted active membranes. *Fundamenta Informaticae*, 58(2):67–77, 2003.

[5] Artiom Alhazov and Linqiang Pan. Polarizationless P Systems with active membranes. *Grammars*, 7:141–159, 2004.

[6] Artiom Alhazov, Linqiang Pan, and Gheorghe Păun. Trading polarizations for labels in P systems with active membranes. *Acta Informatica*, 41(2-3):111–144, December 2004.

[7] Artiom Alhazov and Mario J. Pérez-Jiménez. Uniform solution to QSAT using polarizationless active membranes. In Jérôme Durand-Lose and Maurice Margenstern, editors, *Machines, Computations and Universality (MCU)*, volume 4664 of *LNCS*, pages 122–133, Orléans, France, September 2007. Springer.

[8] Eric Allender. Applications of time-bounded Kolmogorov complexity in complexity theory. In Osamu Watanabe, editor, *Kolmogorov Complexity and Computational Complexity*, chapter 1, pages 4–22. Springer, 1992.

[9] Martyn Amos. *Theoretical and Experimental DNA Computation*. Natural Computing Series. Springer, 2005.

[10] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41(3):274–306, 1990.

[11] Yaakov Benenson, Binyamin Gil, Uri Ben-Dor, Rivka Adar, and Ehud Shapiro. An autonomous molecular computer for logical control of gene expression. *Nature*, 429:423–429, 2004.

[12] Allan Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6(4):733–744, 1977.

[13] José M. Cecilia, Ginés D. Guerrero, Miguel Á. Martínez-del-Amor José M. García, Ignacio Pérez-Hurtado, and Mario J. Pérez-Jiménez. Simulation of p systems with active membranes on cuda. In *International Workshop on High Performance Computational Systems Biology (HiBi09)*, pages 61–70. Conference Publishing Services (IEEE Computer society), 2009.

[14] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[15] Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the International Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North-Holland Publishing Co., Amsterdam, 1964.

[16] Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. Technical Report CaltechPARADISE:2008.ETR090, Caltech Parallel and Distributed Systems Group, 2008.

[17] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[18] Stephen A. Cook. Deterministic cfl's are accepted simultaneously in polynomial time and log squared space. In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 338–345, New York, NY, USA, 1979. ACM.

[19] Stephen A. Cook and Pierre McKenzie. Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8(5):385–394, 1987.

[20] David Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London*, 400(1818):97–117, July 1985.

[21] Daniel Díaz-Pernil, Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, and Agustin Riscos-Núñez. A logarithmic bound for solving subset sum with P systems. In George Eleftherakis, Petros Kefalas, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *8th International Workshop on Membrane Computing, WMC 2007*, volume 4860 of *Lecture Notes in Computer Science*, pages 257–270. Springer, 2007.

[22] Jack Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[23] Peter Fromherz. Neuroelectronic interfacing: Semiconductor chips with ion channels, nerve cells, and brain. In R. Waser, editor, *Nanoelectronics and Information Technology*, pages 78–810. Wiley-VCH Verlag, Berlin, 2003.

[24] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits and the polynomial-time hierarchy. *Theory of Computing Systems (formerly Mathematical Systems Theory)*, 17(1):13–27, 1984.

[25] Renana Gershoni, Ehud Keinan, Gheorghe Paun, Ron Piran, Tamar Ratner, and Sivan Shoshani. Research topics arising from the (planned) p systems. In *Proceedings of the Sixth Brainstorming Week on Membrane Computing*, number 978-84-612-44, pages 183–192, Sevilla, Spain, 2008. Fénix Editora.

[26] Leslie M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the ACM*, 29(4):1073–1086, 1982.

[27] Raymand Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation:P-completeness Theory.* Oxford University Press, New York, Oxford, 1995.

[28] Rosa Gutiérrez-Escudero, Mario J. Pérez-Jiménez, and Miquel Rius-Font. Characterizing tractability by tissue-like P systems. In *Workshop on Membrane Computing 10*, pages 269–181, 2009.

[29] Miguel Gutiérrez-Naranjo, Mario Pérez-Jiménez, Agustín Riscos-Núñez, and Francisco Romero-Campero. On the power of dissolution in P systems with active membranes. In *6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers*, volume 3850, pages 224–240, 2006.

[30] Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, Agustín Riscos-Núñez, and Francisco J. Romero-Campero. Computational efficiency of dissolution rules in membrane systems. *International Journal of Computer Mathematics*, 83(7):593–611, 2006.

[31] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal of Computing*, 17(5):935–938, 1988.

[32] Neil Immerman. Expressibility and parallel complexity. *SIAM Journal on Computing*, 18(3):625–638, 1989.

[33] Neil Immerman. *Descriptive Complexity.* Springer, 1999.

[34] David S. Johnson. Machine models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 69–161. Elsevier Science/MIT Press, Amsterdam, 1990.

[35] Neil D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85, 1975.

[36] Valentine Kabanets. *Nonuniformly Hard Boolean Functions and Uniform Complexity Classes*. PhD thesis, University of Toronto, 2000.

[37] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85—-103. Plenum, New York, 1972.

[38] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, third edition, 1998.

[39] R. E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, 7(1):18–20, 1975.

[40] Leonid A Levin. Universal search problems (Универсальные задачи перебора). *Problems of Information Transmission (Проблемы передачи информации)*, 9(3):265—-266, 1973.

[41] Henry Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153–160, February 2006.

[42] Giancarlo Mauri, Mario Pérez-Jiménez, and Claudio Zandron. On a Păun's conjecture in membrane systems. In José Mira and José R. Álvarez, editors, *Bio-inspired Modeling of Cognitive Tasks*, volume 4527, pages 180–192. Springer Berlin / Heidelberg, 2007.

[43] W.S. McCulloch and W.H. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[44] Niall Murphy and Damien Woods. Active membrane systems without charges and using only symmetric elementary division characterise P. In *Membrane Computing, 8th International Workshop, WMC 2007 Thessaloniki, Greece, Revised Papers*, volume 4860 of *LNCS*, pages 367–384. Springer-Verlag, 2007.

[45] Niall Murphy and Damien Woods. A characterisation of NL using membrane systems without charges and dissolution. *Unconventional Computing, 7th International Conference, UC 2008, LNCS*, 5204:164–176, 2008.

[46] Van Nguyen, David Kearney, and Gianpaolo Gioiosa. An algorithm for non-deterministic object distribution in p systems and its implementation in hardware. In *Membrane Computing - Ninth International Workshop, WMC 2008*, volume 5391, pages 325–354, 2009.

[47] Adam Obtułowicz. Note on some recursive family of P systems with active membranes. `http://ppage.psystems.eu/index.php/Papers`, 2001.

[48] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.

[49] Ian Parberry. *Circuit complexity and neural networks*. MIT Press, 1994.

[50] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[51] Gheorghe Păun. P Systems with active membranes: Attacking NP-Complete problems. *Journal of Automata, Languages and Combinatorics*, 6(1):75–90, 2001.

[52] Gheorghe Păun. *Membrane Computing. An Introduction.* Springer-Verlag, Berlin, 2002.

[53] Gheorghe Păun. Further twenty six open problems in membrane computing. In *Proceedings of the Third Brainstorming Week on Membrane Computing, Sevilla (Spain)*, pages 249–262, January 2005.

[54] Mario J. Pérez-Jiménez, Agustín Riscos-Núñez, Alvaro Romero–Jiménez, and Damien Woods. *Handbook of Membrane systems*, chapter 12: Complexity – Membrane Division, Membrane Creation. Oxford University Press, 2009.

[55] Mario J. Pérez-Jiménez, Alvaro Romero-Jiménez, and Fernando Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, 2(3):265–285, August 2003.

[56] Gheorghe Păun. Further twenty six open problems in membrane computing. In *Proceedings of the Third Brainstorming Week on Membrane Computing, Sevilla (Spain)*, pages 249–262. Fénix Editoria, January 2005.

[57] Michael O. Rabin. Degree of difficulty of computing a function and a partial ordering of recursive sets. Technical Report Tech Report No. 2, Hebrew University, 1960.

[58] Aleksandr A. Razborov. Lower bounds on the monotone complexity of some Boolean functions (Нижние оценки монотонной сложности некоторых булевых функций). *Доклады Академии Наук СССР*, 281:798–801, 1985. (English translation in Soviet Mathematics Doklady, Vol. 37, No. 6, pp. 887—900, June, 1985.).

[59] Alexander A. Razborov and Steven Rudich. Natural proofs. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 204–213, New York, NY, USA, 1994. ACM Press.

[60] Alexander A. Razborov and Steven Rudich. Natural proofs,. *Journal of Computer and System Sciences*, 55(1):24–35, 1997.

[61] Agustín Riscos-Núñez. *Cellular programming: efficient resolution of NP-complete numerical problems.* PhD thesis, Department of Computer Science and Artificial Intelligence, University of Seville, 2004.

[62] Álvaro Romero-Jiménez. *Complexity and Universality in Cellular Models of Computation (Complejidad y Universalidad en Modelos de Computación Celular).* PhD thesis, Department of Computer Science and Artificial Intelligence, University of Seville, 2003.

[63] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and Systems Sciences*, 4(2):177–192, 1970.

[64] David Soloveichik and Erik Winfree. The computational power of Benenson automata. *Theoretical Computer Science*, 344:279–297, 2005.

[65] David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM J. Comput.*, 36(6):1544–1569, 2007.

[66] Petr Sosík. The computational power of cell division in P systems: Beating down parallel computers? *Natural Computing*, 2(3):287–298, August 2003.

[67] Petr Sosík and Alfonso Rodríguez-Patón. Membrane computing and complexity theory: A characterization of PSPACE. *Journal of Computer and System Sciences*, 73(1):137–152, 2007.

[68] Apostolos Syropoulos. Mathematics of multisets. In Cristian Calude, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Multiset Processing*, volume 2235 of *Lecture Notes in Computer Science*, pages 347–358. Springer, 2001.

[69] Róbert Szelepcsényi. The method of forcing for nondeterministic automata. *Bulletin of the EATCS*, 33:96–99, 1987.

[70] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[71] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings London Mathmatical Society*, 2(42):230–265, 1936.

[72] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[73] Erik Winfree, Furong Liu, Lisa A. Wenzler, and Nadrian C. Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394(6693):539–544, August 1998.

[74] Damien Woods, Niall Murphy, Mario J. Pérez-Jiménez, and Agustín Riscos-Núñez. Membrane dissolution and division in P. In *Unconventional Computation*, volume 5715, pages 262–276, 2009.

[75] Claudio Zandron, Claudio Ferretti, and Giancarlo Mauri. Solving NP-complete problems using P Systems with active membranes. In I. Antoniou, Cristian Calude, and M.J. Dinneen, editors, *UMC '00: Proceedings of the Second International Conference on Unconventional Models of Computation*, pages 289–301, London, UK, February 2000. Springer-Verlag.