Diss. ETH No. 21173

Self-Construction in the Context of Cortical Growth: From One Cell to a Cortex to a Programming Paradigm for Self-Constructing Systems

A dissertation submitted to

ETH Zurich

for the degree of

Doctor of Sciences

presented by

Andreas Hauri

Master of Science in Computer Science ETHZ

born 30 September 1981

citizen of
Hirschthal, Switzerland

accepted on the recommendation of

Prof. Dr. Rodney J. Douglas, examiner

Prof. Dr. Christoph von der Malsburg, co-examiner

Prof. Dr. David Willshaw, co-examiner

2013

Disclaimer

I hereby declare that the work in this thesis is that of the candidate alone except as described below.

The use of “we” in the thesis refers to the candidate, but also contains intellectual input from: Rodney Douglas (thesis supervisor), Fred Zubler (G-Code expert), Sabina Pfister (Gene Regulatory Network expert), Roman Bauer and Michael Pfeiffer.

Abstract

In this thesis we explore the biological development of the neocortex as a model for self-constructing systems. Development begins with a single cell, and expands that cell through cell division and specialization to create a complex organism consisting of trillions (in humans) of cells. We will consider the branch of this process that leads to the neocortex, beginning with precursors in the ventricular zone, and the ganglionic eminences. In biology the instructions for construction of a target organism are encoded in its DNA-code, and these instructions are selectively decoded by a gene regulatory network according to prevailing intra-cellular and extracellular conditions. We approximate this elaborate process with a model that contains the essential elements of biological self-construction, but is sufficiently tractable to be simulated. Because our final goal is to exploit biological construction as a practical engineering technology, we choose a style of simulation that emphasizes physical process in a 3D spatial environment. And, because the compelling feature of biological development is that it is distributed and proceeds without a global controller, our model emphasizes self-construction as the result of locally interacting distributed autonomous agents. The model is simulated using a specialized software platform, Cx3Dp, that is able to simulate the physical development of millions of cells. The biological behavior of these cells is controlled by G-Code, a DNA-like specification language that is able to model cellular behavior as observed in nature. The organization of the G-Code 'genome' that is inserted into the cortical precursor cells of the embryonic ventricular zone is obtained by applying the methods of Pfister et al. [146], by which sparse experimental data are used to estimate a model genome able to express a gene-regulatory network (GRN) that controls mouse corticogenesis. The various cell types of the cortical lineage tree are expressions of the attractor states of the GRN. The various states release G-Code encoded cellular functions that cause the cells to divide, migrate, and differentiate. Using these concepts and methods we are able to simulate the self-construction of a sheet of neocortex composed of two areas containing some 200,000 neurons. The simulation recapitulates the majority of experimentally observed features of this process, including the detailed inter- and intra-laminar axonal connection patterns. The simulation platform is parallelized, and its performance scales well with simulation size, so that the simulation of much larger volumes of cortex are possible. These explorations of cortical development lead us to a new paradigm of engineered self-construction: the Developmental Programming paradigm. At the center of this paradigm is the concept of exponential construction, whereby a single precursor containing a single genome amplifies itself by successive replication and specialization, and so is able construct a vastly more elaborate product than can be achieved with conventional, more linear fabrication methods. There are three crucial components in the construction process: firstly, the genome that encodes what functions are possible, and provides rules for their expression; secondly the machine-like agents that are instantiations of the genomic functions; and thirdly the container (e.g. cell membrane) that provides the spatial and temporal scope of genome and agent activities. The contents of containers work independently of one another, but can be coupled through local messages. This is a highly parallel and distributed process, in which the computational power of the overall system increases with each container replication. In this thesis we have considered this self-construction process as it is embodied in cortical development. However, it is easy to see how these principles could be extended to novel self-replicating technologies as diverse as genetically engineered cells, nano-robots, DNA-computing systems, and self-configuring software for exascale computers.

Zusammenfassung

In dieser Doktorarbeit untersuchen wir die biologische Entwicklung des Neocortex als Modell für selbstkonstruierende Systeme. Die Entwicklung beginnt mit einer einzigen Zelle, und erweitert diese Zelle durch Zellteilung und Spezialisierung in einen komplexen Organismus, bestehend aus Billionen von Zellen (beim Menschen). Wir betrachten den Teil dieses Prozesses welcher zum Neocortex führt, beginnend mit Präkursor-Zellen in der ventrikulären Zone und den Ganglionic-Eminences. In der Biologie sind die Befehle zum Aufbau eines Zielorganismus in seiner DNS codiert. Diese Befehle werden selektiv durch ein Genregulationsnetzwerk decodiert, je nach den Bedingungen die intrazellulär und extrazellulär vorherrschen. Wir approximieren diesen aufwendigen Prozess mit einem Modell, welches die wesentlichen Elemente des biologischen Selbstkonstruktion enthält, jedoch steuerbar genug ist um es zu simulieren. Da es unser Endziel ist, die biologische Bauweise als praktische Ingenieurtechnik zu nutzen, wählen wir eine Art von Simulation, die den physikalischen Prozess in einer 3 dimensionalen Umgebung betont. Und da die faszinierendste Funktion der biologischen Entwicklung die verteilte Funktionsweise ist, welche ohne globale Steuerung auskommt, betont unser Modell, dass die Selbst-Konstruktion das Ergebnis von lokal interagierenden, verteilten und autonomen Agenten ist. Das Modell wird mit Hilfe der speziellen Software-Plattform, Cx3Dp simuliert, welche in der Lage ist die physikalische Entwicklung von Millionen von Zellen zu simulieren. Das biologische Verhalten dieser Zellen wird durch G-Code gesteuert, eine DNS-ähnliche Beschreibungssprache, die in Lage ist zelluläres Verhalten wie es in der Natur beobachtet wird zu modellieren. Die Organisation des G-Code „Genoms“ wird durch die Anwendung der Methoden von Pfister et al. [146] erlangt und in die kortikalen Präkursor-Zellen der embryonalen ventrikulären Zone eingefügt. Bei dieser Methode wird mittels wenigen experimentellen Daten ein Modell Genom bestimmt, welches ein Genregulationsnetzwerk (GRN) ausdrückt, das die Kortikogenese der Maus steuert. Die verschiedenen Zelltypen der kortikalen Entwicklungslinie sind der Ausdruck von den Attraktor Zuständen des GRN. Die verschiedenen Zustände starten in G-Code kodierte zelluläre Funktionen, welche die Zellen teilen, migrieren und differenzieren lassen. Unter Verwendung dieser Konzepte und Methoden können wir die Selbst-Konstruktion einer neocortikalen Platte bestehend aus zwei Zonen mit ca. 200'000 Neuronen simulieren. Die Simulationsergebnisse rekapitulieren die Mehrheit der experimentell beobachteten Merkmale dieses Prozesses, einschliesslich der detaillierten inter- und intra- laminaren axonalen Verbindungsmuster. Die Simulationsplattform ist parallelisiert und Ihre Leistungsfähigkeit skaliert gut mit der Simulationsgrösse, so dass gar Simulationen von grösseren kortikalen Volumen möglich sind. Diese Erkundungen der kortikalen Entwicklung führen uns zu einem neuen Paradigma der technischen Selbst-Konstruktion: Das Developmental-Programming-Paradigma. Im Zentrum dieses Paradigmas ist das Konzept von exponentieller Konstruktion, bei welchem ein einziger Präkursorbehälter mit einem einzigen Genom sich selbst Amplifiziert durch Replikation und Spezialization, und fähig ist ein weitaus elaborierteres Produkt zu konstruieren als es mit konventionellen linearen Fabrikationsmethoden möglich ist. Hier sind drei wichtige Komponenten im Konstruktionsprozess: Erstens, das Genom, welches die möglichen Funktionen kodiert und Regeln für ihren Ausdruck enthält. Zweitens, die maschinenähnlichen Agenten welche die Instanziierung der genomisch enkodierten Funktionen sind. Und drittens, der Behälter (z.B. Zellmembrane) welche das örtlichen und zeitlichen Ausmass des Genoms und der Agenten Aktivitäten bestimmt. Die Inhalte des Behälters arbeiten unabhängig voneinander, können aber voneinander abhängig gemacht werden mittels Nachrichten. Dies ist ein hoch parallelisierter und verteilter Prozess, welcher die Rechenleistung des Gesamtsystems mit jeder Replikation eines Behälters erhöht. In dieser Doktorarbeit haben wir den Prozess der Selbstkonstruktion, wie er in der kortikalen Entwicklung verkörpert ist, untersucht. Allerdings ist es einfach zu sehen wie diese Prinzipien auf neue selbstreplizierende Technologien erweitert werden könnten, Technologien wie artifizielle Zellen, Nanoroboter, DNS-Rechnersysteme und selbstkonfigurierende Software für exascale Computer.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Importance of Self-Construction

In this thesis we want to investigate how from a single cell, the fertilized egg, a complete and fully functional organism can be constructed. How can cells without a global controller, a central construction unit, reliably create organisms that have similar shapes and that can survive? Cells seem merrily to replicate but do not seem to know exactly what they have to do at each stage of development. We want to investigate the principles that let a single cell develop into such a huge diversity of cell types and assemble into a functional organism that consists of billions of cells. It is not yet clear how this developmental process exactly works and it remains a mystery on which we would like to shed light. This process of development is called more abstractly self-construction. This concept extends beyond biology. If we want to learn about self-construction we have to look at development, since biological development is currently the only system that successfully implements this technique. Current engineering techniques usually rely on a central global construction controller and are hardly parallel. Further, these techniques mainly use big machines and construct much tinier objects. Even the most flexible construction techniques such as 3D-Printing work this way and are globally controlled. In self-constructing systems the process is not globally controlled but the control is distributed, highly parallel and the constructing entities are part of the final object. This type of construction has certain advantages over state of the art engineering techniques: self-construction leads to systems that are inherently capable of self-repair. The construction process is distributed, it therefore does not have a single point of failure. This means that even though many errors can occur in development the whole system does not fail. Replication is a necessary property of a self-constructing system, it renders the process very scalable, potentially enabling an exponential speed up of growth.

Self-constructing systems can also potentially evolve as we see in biology. If we were to understand and recreate a system that is self-constructing, it would revolutionise our process of construction. Not only is it currently not possible to build systems that truly mimic the biological process of construction, but it is also not understood how to usefully instruct self-organizing systems. With this thesis a step is taken in this direction and we investigate biological principles and abstract from them useful concepts for engineering self-construction. By abstracting and learning from biological systems we also automatically gain insight into how nature has tackled these problems. We begin to better understand how the interaction between nature's programming languages, genome and development, works.

It is obvious if we want to investigate self-construction that we should focus on the only process that we know already implements self-construction, i.e. biology, and more precisely the development of biological organisms. We need an example of a system that is well studied and sufficiently complex in order to be interesting to investigate. An obvious choice is the development of the cerebral cortex, since the cerebral cortex is a brain region that has been intensely studied for decades and in which many biological processes have been observed and documented. Specifically the study of mouse cortical structure has been the subject of a great deal of literature. The mouse is an obvious choice since it is one of, or even the best studied organism in biology and it is reasonably close in evolution to humans to possess a similar brain structure. Studying the cortex as a brain structure is especially interesting as it is thought be the seat of intelligence in humans. Most genetic tools for mammals such as the gene-knockout techniques are mostly developed for mice, and they are one of the most economic and least controversial animal models.

The cortex is a structure that is in itself very interesting for its computational properties, since it is at the core of human intelligence. It is the only system that evolution has come up with that implements intelligence. The cortex and its development, even without considering the aspect of self-construction, is a fascinating subject to study. In fact it is one of the most studied brain structures and has been intensely investigated to decipher its properties. It is highly regular in structure despite some interareal and interspecies differences. Its pattern of connectivity is reasonably well understood also thanks to collaborators such as Kevan Martin and Henry Kennedy who are at the core of the investigation of the development of the adult mouse and monkey cortices.

We aim at understanding the structure and the computational properties of the cortex by constructing it as it is naturally constructed via the simulation of its development from the genome inserted in the first cell to a complete organism. For example the human genome itself is only on the order of thirty thousand genes [44]; on the order of gigabytes [40] of information. This amount of information is enough to construct a human brain (not considering the human body) that consists of billions of cells. According to the complexity theory of Kolmogorov [105], all the information in the initial code, the environment and the physical rules must be sufficient to generate the final structure since no other information sources are present. According to Kolmogorov, computation cannot create complexity since complexity is defined by the shortest program necessary to describe an output and therefore a structure. This means that if we could decipher the genome and the principles of how it unfolds into structure we would understand the functional principles by which the brain sets itself up into its mode of operation. This implies we could obtain an understanding of the workings of the brain. This approach contrasts strongly with the current mainstream in neuroscience that reconstructs large parts of brains in huge detail, in exabytes of data. These exabytes of data should then be fed into huge simulations with the hope that an intelligent system will emerge, without an exact understanding of the rules that are behind it.

1.2 The Goals of the Thesis

Having explained the importance of self-construction and that we want to investigate this type of construction in the biological framework of cortical development we now specify more clearly what the goals of this thesis are. We want to understand the principles of self-construction based on the example of a developing cortex. We therefore model the development or construction process of the mouse cortical structure in accordance with the literature. For this investigation we have to create new tools that are important for the analysis and a central piece of work for future investigations of development. Even though the latter is not central to the thesis it is still a very important piece of work. Accordingly, the goals of this thesis are as follows:

1. *Simulation of Mouse Cortex Areas 3 and 6:* With Cx3Dp we will simulate the growth of a cortical structure, beginning with simple progenitor cells in the ganglionic eminence, neural epithelium and thalamus. The simulation shall include models of cellular behavior that have been observed in biology such as radial migration of pyramidal cells, tangential migration of smooth cells, in-growth of thalamic afferents in to the cortex, creation of cortical lamination in an inside out manner and differences in thicknesses of the modeled areas 3 and 6. All of these modeled processes shall be related in their relative timing as observed in biology. All of the processes used in this simulation must be encoded in a single genome. The genome will be coded in an improved version of the G-Code language of Zubler et al. [213] which ensures that executed behaviors are only based on biologically plausible processes. All the simulated cells must find their location or fate in the final global structure only based on their local environment, and the genome and physical rules given by Cx3Dp. There must not be any active global controlling instance in the system. The

simulation shall begin with a few thousand cells and end up with a completely laminated cortex with hundreds of thousands of cells.

2. *Extraction of Self-Construction Principles into an Engineering Technique:* Having created the simulation, beginning only with a few progenitor cells and a genome and ending up with a complicated cortical-like structure with hundreds of thousands of cells, will have led to a lot of insights of how biology executes this construction. These insights shall be abstracted and transferred from biological systems to engineering techniques that can be used to build programs for systems that allow self-construction. This new paradigm (Developmental Programming), describing how self-construction systems can be programmed, shall be analyzed for its computational power and potential complexity to determine what types of structures (constructs) can be programmed with such a system, where constructs are defined as globally assembled structures built out of single-cellular entities with no global control over the whole construct. The focus where the paradigm lies and how it compares to other programming paradigms shall be specified. An example of a language implementing the developmental programing paradigm shall be given and explained. How systems have to be designed to allow them to be programmable in a self-organizing self-constructing way shall be analyzed. The idea directly follows the work of Roth [159] and Zubler [213] refining the ideas of a self-construction language.

3. *Parallelization of Cx3D:* Cx3D was written by Zubler et al. [211] as a single-threaded program that is capable of simulating a few thousand cells. In this thesis we aim for to parallelize the single-threaded software to make it capable of exploiting a multi-core and multi-computer environment. For the parallelization the code will undergo a complete revision and will have to be drastically changed. This system (Cx3Dp) should be capable of performing simulations that use more than a million cellular objects. It shall be intended for local area network heterogenous computer environments and not be specifically designed for a specific cluster computer.

In achieving these goals we will have gone from creating a parallel simulation program to a plausible biological simulation to a theory of the developmental process. The theory of biological self-construction serves as a basis for a programming paradigm that can be used by engineers to program systems that are potentially self-constructing.

The understanding of this self-constructing process could lead to a new area of how one constructs and thinks about the process of construction.

1.3 Scientific Context of the Thesis

This thesis is designed to be at the boundary between biology and computer science. We aim at investigating the behavior of cells in a biological system at a more abstract level than the biochemical level. This means that we will neither consider specific genes nor specific biochemical pathways. We model the quorum effect of these proteins contributing to the cellular behavior, by describing the biological behavior of the cells deeply enough that all of the processes we describe with our models are observed and biologically justifiable. Our models might use a different implementation than the real biology does but it is qualitatively achieving the same behavior as is observed. We want to learn about the behavior of the cells and how to use the principles found in biology to then apply this knowledge in a computer science context. We do not conduct biological experiments, and are therefore not reporting on biological findings that have been made by us, but we are interested in the biology and what it can teach us about how development and therefore self-construction works, which is on a more abstract theoretical level then usual biological doctoral theses. All our experiments are done purely in simulation.

It needs to be clarified that when we talk about genetic mechanisms, we abstract the genome with our language G-Code, but we do not use genetic algorithms. Genetic algorithms are for automatically evolving code and/or behavior of software according to a fitness function [63] [200]. Evolving a code according to a fitness function is not our goal, our goal is to explicitly write the genome, since we want to find out about how to engineer code and not how to automatically find solutions to a problem describable by a fitness function. Genetic algorithms often find solutions that work well for the given fitness function but that are not humanly comprehensible in their meaning [84]. Our goal is to understand the relations between genome, cells, the cells' environment and final global structure.

Another common misconception regarding this line of work is that the self-constructing process is an "intelligent" process; concepts such as "swarm intelligence" [22] are not comparable to human intelligence. Even though self-constructing systems and hence development can react directly to the environment this process is nothing more then a feedback loop for very specific, predefined, and usually expected events. Since we use cortical growth as an example of self-construction for a system that is in biology at least partially responsible for the "intelligence" of the organism, the construction of it is not intelligent but follows pre-specified rules. That means that no cell exhibits attributes that should be seen as intelligent behavior. In our simulation we strictly separate this "intelligent"/learning behavior from the construction process and do not even allow for any kind of synaptic or machine "learning" during the simulation. However it can be argued (with good reason) that classical synapse driven learning is happening in the biological cortex [165] [65]. We are using the development of cortex only as a well studied example of a developmental process.

1.4 Readers Guide to the Thesis

The results chapters are self contained with short transitions between them that explain the need for the following chapter. Each part has an introduction, and a conclusion at the end of the chapter. Having explained the biological and theoretical background of this thesis, four results chapters follow: the modelling of the brain areas of the mouse (chapter 3), an analysis of the biological self-construction process (chapter 4), the Developmental Programming paradigm, (chapter 5) that is based on the biological analysis, and finally a chapter about the Cx3Dp software (chapter 6) that was used for the simulations. At the end there is a general discussion and outlook.

Chapter 2

Background

Before the simulation and theory of self-construction is explained, a certain amount of background is needed to follow the text. We first explain the current knowledge in the field of cortical development in mouse areas 3 and 6. This explanation serves later as the basis for the model in the simulation. Second, we explain how cells are able to generate different cell types over the course of development in the simulation this will also serve as a background for the theory of biological self-construction. Third and last, a short overview is given about the language that can be used in Cx3Dp for programming the artificial genome.

2.1 Biological Background: The Story of the Mouse Lamination

2.1.1 Level of Abstraction

In this chapter we assume a fairly abstract view of the developmental process and explain events on a systemic level rather than on a gene expression guided level. We are interested in the behavior of the cells and how they interact with each other but not at the molecular level, but rather why and when they need to interact in order to construct the cortex. We are interested in the orchestration of the cells — how do they manage to construct such a system in an autonomous, distributed and therefore self-constructing fashion? Whenever we arrive at a point where information exchange is necessary we acknowledge that there must be a signal but do not specify or predict what the signal is, just define the nature of this signal. Each part of our model could be taken and refined in an iterative fashion down to the molecular level. This refinement would in each case certainly take years of work. But the epiphenomena of cell behavior we describe here must at least be achieved by molecular processes such that the system works as observed.

Cells constructing the cortex only have a limited amount of patterning of space at their disposal that is present at the beginning of the process. The developmental process must create its own environment during the course of the development. Each cell is an autonomous entity that has to find its position in the construction of the developing cortex. Somehow each cell must gather the information of where its type is needed during the construction process. It is even the case that some developmental stages are only scaffolds for the later and final construction and will be destroyed after having served as an information source for orientation for the construction process. That might even include cells that will undergo apoptosis when they have served their purpose or are positioned at the wrong place in the finishing structure. But each cell has to decide autonomously what it has to do next in the construction process. To do so the cell can only rely on the external information of its local environment, its internal state (expression levels) and the genome. It will integrate this information and define the appropriate action to perform. The cell never has the ability to view the whole cortex at a global level in order to decide which actions to take. The currently active cells have to shape the environment in such a way that throughout the stages of cortical construction, the information is at hand where it is needed by the cells that play a role in the next step of the development. Through time, the construction morphs the organism through different stages of (sometimes) transient scaffolds that are all necessary to end up with the final functional cortex and/or body. We can already see that timing plays a crucial role in the development: if information that a certain cell needs is not on hand at the right time, the construction will not work (e.g. in the reeler mouse [2] [29] [130]).

This chapter provides a fairly abstract view of the development process and covers a lot of literature. We therefore often refer to summarizing reviews rather then the original articles to make our point.

2.1.2 Overview of the Mouse Cortical Construction

We first give the reader a crude overview of the developmental process of cortical development in the mouse which is a field with a huge literature. We will discuss only the parts on which we focus.

Just above the lateral ventricles in the dorsal telencephalon, a layer of neuroepithelial cells is located called the ventricular zone. These cells proliferate and build the preplate. The preplate is then split into marginal zone and subplate by the first developing neurons that are produced in the ventricular zone. [131] [142] [153] [2] The cortical plate consists of the cortical layers 6, 5, 4 and 2/3. These layers will settle below the marginal zone: layer 2/3 topmost and 4 the lowest. [152] Layers 6 and 5 are produced by the ventricular zone that will also produce the subventricular zone that rests above the ventricular zone. [59] [131] Subventricular zone cells will then produce layers 4 and 2/3. Neurons produced in the ventricular and subventricular zone migrate radially from the ventricular site to their specific location. All of these neurons are excitatory cells and are of pyramidal shape. The cells medially begin to grow axons and dendrites to create specific connections between the layers. [148] [23]

A second population of cells (interneurons) are produced in the marginal zone located in the ventral telencephalon. These interneurons migrate in the direction of the cortex, tangentially invading the cortex, and settle in the cortical layers. Interneurons are inhibitory neurons. [18] [118] [129] Their location in the different layers is birth-time dependent and they are produced in the same order as the pyramidal cells of the corresponding layer, layer 6 first and layer 2/3 last. [18] The production of these neurons is believed to be guided by cell internal states represented by a system called a gene regulatory network. This regulation of cell numbers is influenced by internal signals as well as external signals defining the fate and the order in which the cells are born.

The cortical plate is divided into areas with different functions. This division is also reflected in their anatomy, in that different areas have different thicknesses of layers and have different connection targets for cortical-cortical, cortical-thalamic projection and other cortico-fugal projections. [51] They are also subject to different invading projections from other cortical areas, thalamic areas and other brain structures. In order to generate different thicknesses, the gene regulatory network has to behave differently in different cortical areas in the corresponding cells of the proliferative layers. [147]

The cortex connects to the thalamus by sending axons in its direction [130]. The thalamic outgrowth already begins at early stages of development when only the subplate is established [130] [127] [153]. The subplate cells send axons in the direction of the thalamus and the thalamus sends axons in the direction of the cortex. They meet in the internal capsule and match up. They are then both believed to guide each other to their corresponding origins, establishing a connection between cortex and thalamus. [130] Later in development, layers 6 and 5 also send axonal fibers in the direction of the thalamus. [127] [112] The thalamic fibers will connect to layers 6 and 5.

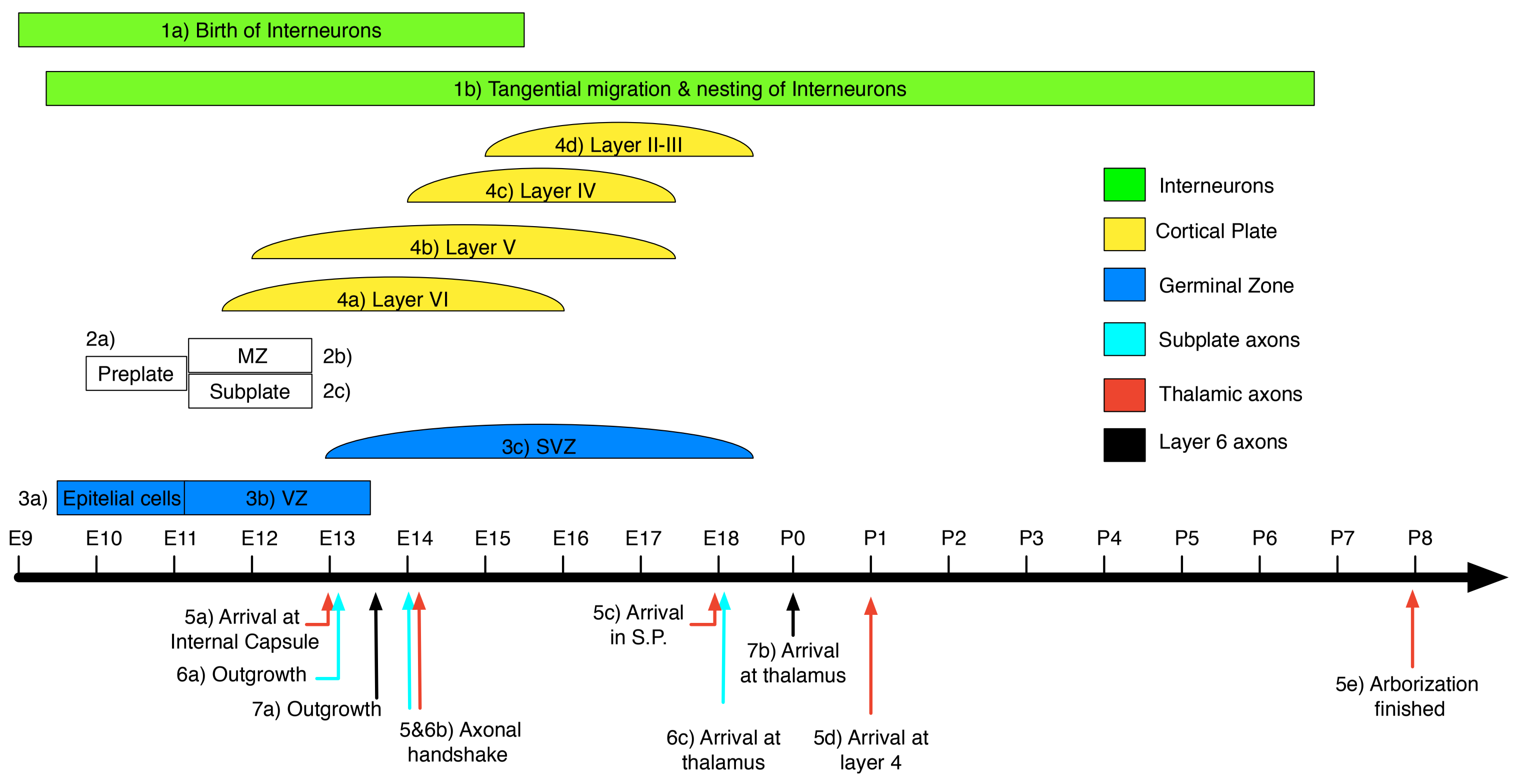


Figure 2.1. An overview of events happening during cortical development from the initial cortical neuroepithelial cells to the fully grown cortex. These are the processes we discuss in this chapter. This figure shows only the time the different structures arise. The color-code is to indicate what type of process is happening. First the germinal zones are born that will then spawn the cells of the cortical layers. During that time the interneurons are born in the ganglionic eminence and travel to the cortical structure where they integrate. The cells grow out axons and dendrites when they arrive at their destination and the connection between the thalamus and the cortex is established via axons.

2.1.3 Inconsistencies in the Literature

The broad spectrum of the literature gives many theories for how the orchestration of particular processes works. We inform the reader of these at the points at which we arrive at them and show the range of ideas that have emerged in the literature concerning this process.

One of the main issues is relative timing of events happening in the self-construction process. The relative timing of these events is of utmost importance. The behavior of cells at a certain point in time can only be explained by information cues that have been distributed intracellularly through lineage and/or state information or extracellularly through markers in space. But in both cases the events that bring the information to the cells must have happened before the cell has to make a decision for the future construction. To give just one example, the thalamic fibers cannot influence the development of the cortex if they have not yet reached the cortex. [51] Being aware of the timing issues and knowing that we have to integrate information from literature that is so far-reaching, this can pose quite a challenge. In the literature, embryonic days Ei start to be counted at different time points (there is approximately half a day uncertainty). Additionally even the starting label varies, beginning at either E0 or E1 leading to quite a spread of the use of these time markers in the literature. The development of the mouse is quite fast, on the order of days rather than weeks, so a shift of one day might make the difference as to whether an event in the development of one part of the cortical assembly can have an influence on the other part. The differences in start dates can be seen again in the interplay between the birth of layer 4 cells and the arrival of the thalamocortical projections in the cortex. [147] [13] That plays directly into the discussion of whether there is a waiting time or whether the layer 4 cells have an influence on the turning of the thalamic axons in the direction of the cortical plate or not. We tried to disentangle the timing issues as far as possible but might still sometimes be off, and it is important that the reader is aware of this fact, also when reading the other literature referred to in this chapter.

Since in different parts of different literature different names have been used that actually describe the same cells or phenomena the reader might find it difficult to find the right key words to search for the phenomena described. Therefore we try to give all the synonyms that we found each time we encounter a new expression.

2.1.4 Inter-Species Differences, the Importance of Timing

Even though the construction process of many parts of the cortex are quite similar in terms of timing and the final appearance of the cortex, there are many subtle differences in the timing, the number of cells and the cell types found in different animals with a cortex or cortical-like structure. Even slight differences can make a tremendous difference in the final shape of the animal's cortex. For example thalamic ingrowth, which seems to play a crucial role for the development of the healthy cortex in primates but not in mouse [99]. Even among rodents there are differences that have a huge effect on the development. If we look at the development of the thalamic fibers there are differences in when which parts of the cortex connect to which parts of the thalamus and vice versa; for example between hamster and mouse development [127] [13]. The differences become even larger going from rodents to carnivores such as cat [2] and ferret [42] or to primates [99]. These differences might also be attributed to the timing incongruences in the literature or different methods of fiber tracking, but still there seems to be a difference. This uncertainty leads us to the problem that not everything is known and not all experiments have been done in the mouse. Therefore we will if necessary make assumptions based on other animals. However, if not otherwise indicated, the results are believed to be true for the mouse.

2.1.5 Germinal Zones

The cell generating zones in cortex are called the germinal zones. There are three zones in the mouse that are part of the germinal zones: the ventricular zone (VZ), the subventricular zone (SVZ) and the intermediate zone (IZ). The IZ is not discussed in this thesis.

The ventricular zone primary source of cortical generation is already established at the very early stages, even before the preplate is created (at Embryonic day E10 to E11 in the mouse). Its pluripotent cells are called the neuroepithelial cells at that time. It will initiate the corticogenesis (figure 2.1 process 3a). Later, the cells of the ventricular zone will be called radial glial cells (RGCs) (figure 2.1 process 3b). These are believed to build the starting pool of the progenitor cells in the mouse cortex [131] [142] [153]. There are also other progenitor cells than the radial glial cells in the ventricular zone, but the radial glial cells are the majority of the cells in that zone. Through symmetrical division, these cells carefully control the pool of progenitor cells in the early stages of the cortical plate development [152].

Before the onset of neurogenesis, neuroepithelial cells self-renew by symmetric divisions. As neurogenesis progresses, neuroepithelial cells differentiate into radial glial cells [77, 128, 140, 141, 7]. Radial glial cells divide at the apical surface of the ventricular zone and undergo stereotypical patterns of cell divisions: symmetric divisions amplify the pool of precursor cells [149], whereas asymmetric divisions (which become predominant) give rise directly and indirectly to the majority of cortical neurons [81, 114, 7]. These cells are the starting point for all of the production of the pyramidal cells of layer 6 to layer 2/3, though the production of the different cell types greatly overlaps. The timings are mostly deduced from Polleux et al. [147] unless otherwise indicated. The reader must keep in mind that timings given here are only approximate and they may vary depending on the cortical area.

The progenitor cells then begin to perform asymmetric division and generate layer 6 cells that will migrate up radially into the middle of the preplate to split it into two: the marginal zone and the subplate [2] beginning at E11.5 and ending production at E16 (figure 2.1 process 4a). The progenitor cells in the VZ therefore divide mostly asymmetrically into one neuron and one progenitor. Following layer 6, layer 5 neurons are the next to be generated by the ventricular zone beginning at E12 and ending at E17.5 (figure 2.1 process 4b). (Layers 5 and 6 are called the infragranular layers.)

Slightly later, beginning at E13, a new zone just above the ventricular zone forms called the subventricular zone [59] [131] (figure 2.1 process 3c). This zone is responsible for generating layer 4 first, starting at E14 and ending at E17.5 (figure 2.1 process 4c). After the generation of layer 4 the subventricular zone generates the layer 2/3 neurons, starting at E15 to E19 (figure 2.1 process 4d). These layers are also called the supragranular layers.

There is evidence that the radial glial cells of the ventricular zone later generate astrocytes [152]. Radial glial cells are present until late neurogenesis before birth, when they differentiate further into glial-restricted precursors and produce, by rapid symmetrical divisions, a cohort of glial cells invading the laminated cortex [45]. Most of the subventricular zone and the ventricular zone is exhausted during the corticogenesis. However a small subpopulation of RGCs retain apical contact and continues to generate neurons and oligodendrocytes in the neonate, or converts into adult SVZ astrocytes, which differentiate into adult neural stem cells in the adult (reviewed in [106]). The differences in cortical layer thickness are attributed to different schemes of proliferation [147]. In this thesis we neglect the role of the intermediate zone.

2.1.6 The Preplate

The preplate (PP) builds the structure that develops prior to the construction of the cortical layers.

In the early stage of mouse development, just above the lateral ventricles in the location where the cortex will be built, a thin sheet of neuroepithelial cells is situated. These neuroepithelial cells are a proliferating pool of cells. This zone in the literature of cortical development is known as the ventricular zone (VZ). Above the ventricular zone is the pia. The neuroepithelial cells will begin to proliferate and build a scaffold, the preplate, for the later construction of the cortex. The preplate cells rest between the pia and the ventricular zone. This development happens at embryonic day E10 to E11 in the mouse (figure 2.1 process 2a) [2] [171] [51] [124]. The neuroepithelial cells are already beginning to build processes and will later be known as the radial glial cells.

Many of the ventricular zone cells (radial glial cells) have apical end feet that form a fiber mesh at the ventricle and a basal process that reaches to the top of the marginal zone and forms a fiber mesh there. These fibers extend constantly during growth and play a crucial role in the radial migration of the pyramidal neurons (glutamatergic neurons / exitatory neurons / projection neurons). NB: Basal and apical orientation in radial glial cells is different to that of the neurons of the cortex. In radial glial cells the apical direction is in the direction of the ventricle, whereas in cortical neurons, apical describes processes going toward the pia.

After the ventricular zone has produced the cells of the preplate, it begins to create the cortical plate. The generated cortical neurons start to migrate from the ventricular zone in the direction of the pia, away from the ventricle. At E11 the preplate is split into two by the first cortical neurons to settle (forming the cortical plate). These cortical neurons will later form layer 6 (also known as layer 6a). The two new zones that are created from the preplate are then called the marginal zone (MZ) (the later layer 1) and the subplate (SP) also called the intermediate zone or layer 6b which later has fibers in the white matter.

Marginal Zone

The marginal zone lies over the cortical plate and later becomes layer 1. The marginal zone provides a stop signal during development for the radially migrating cells (cortical plate neurons). The signal is known to be reelin. The reelin positive cells are called the Cajal-Retzius cells and build a large group of cells in the marginal zone. These cells must obviously already be there before the subplate is split in order to give the signal to the cortical plate cells to stop migrating and create the splitting. The reelin secreted by the Cajal-Retzius cells also seems to provide a directional cue to the radially migrating cells. If these cells are defective we see the reeler mutation [2] [29] [130].

The marginal zone acts also as a migrational zone for tangentially migrating cells (interneurons / gabaergic neurons / inhibitory neurons) that are coming from deeper cortical structures such as the ganglionic eminence. The origin of these cells is debated in the literature. It is claimed that a close relation between the Cajal-Retzius and the other preplate cells / cortical cells exists [171], but there is also the finding of Mayer et al. [124] that these cells originate from the cortical hem and migrate tangentially in to the cortex.

Subplate

The subplate, or later layer 6b, is a structure that lies just below the cortical plate and above the germinal zone (the proliferating zones ventricular zone, intermediate zone (IZ) and subventricular zone (SVZ)). Pyramidal neurons created in the germinal zone will have to cross the germinal zone radially in order to come to rest in the cortex. The roll of the subplate cells is still heavily debated and not extremely well known. They are neurons, seem to be electrically active during the construction process, and seem to be important for the development of the correct working regime of cortical neurons. [96] And the subplate cells are suspected to play a role in the setup of the connection between thalamus and layer 4.

The subplate cells are the first cells sending out axons (part of the corticofugal / corticothalamic connection) in the direction of the thalamus. And they are the ones doing the "handshake" in the internal capsule (IC) with the thalamic axons (thalamocortical connection / thalamic fibers / thalamic afferents) [130] [118]. The name Handshake Hypothesis describes the process whereby the two fiber tracks, corticothalamic and thalamocortical fibers, meet and seem to fasciculate to each other and lead or guide one another to the reciprocal target. Therefore, they seem to play a role in the topography mapping and the setup for the connection from thalamus to cortex. This zone is an area through which many neurons migrate. Interneurons migrate through tangentially; pyramidal neurons migrate through radially [134] [18] [182]. Furthermore it is an in- and out-growth zone for thalamic fibers, corticothalamic fibers [127] [150] and other fibers we do not describe, which together slowly build the white matter.

The final fate of the subplate cells is debated. One theory is that during development these cells undergo apoptosis [2]. Another is that they persist in the white matter [189]. Some of them though seem to build layer 6b [13]. It seems to be clear that the layer that is defined as the subplate is less populated in the adult than in the embryo.

2.1.7 Cortical Plate

The cortex is classically divided vertically into six Layers. During development layer 1 is still called the marginal zone and is not counted as part of the cortical plate [118]. Layers 2 and 3 are usually counted together as one and called layer 2/3. The astonishing thing about cortical development is that it is an inside out development. The order of the layers in the cortical plate is reversed, producing first the lowest layer, layer 6, then 5, 4 and lastly the highest layer, layer 2/3, where all the cells of the upper layers have to migrate trough the lower ones in order to establish the correct cortical lamination [2] [131] [153] [147].

Corticogenesis takes place over several days during the mouse embriogenesis, depending on the source, from the onset of E11–11.5 to end of E17.5–E19 [131] [147]. There are areal differences in the development [147]. The estimation of these dates also depends on whether one takes into account just the date of birth of the cells or also their migration time. If one counts the settling time of the tangentially inwards migrating interneurons, corticogenesis takes even longer, up to P7 [129]. Indeed cortical development is special; there are no cells born in the cortical plate. Cortical cells migrate along different routes into the cortical plate and settle in the appropriate place.

There are two main different types of migration; the radial migration of the excitatory pyramidal neurons and the tangential migration of the inhibitory interneurons. These different types of neurons are born, in the case of pyramidal cells, below the cortex in the germinal zone [147] [152] [153] [153], and in the case of the interneurons in the ganglionic eminence [129]. The tangential migration process has to be timed to match the right kind of interneurons with the right kind of pyramidal cells to arrive at the right cortical configuration. Since the birth places of these two neuron types are rather far away (at least in the mouse) the mystery of the timing between the birth of the inhibitory and the excitatory populations is still unsolved. Somehow they have to be matched up but the locality of information principle we are using tells us that they somehow must have a way to achieve this careful orchestration through communication, since cells born below the cortex can not communicate directly with the cells born in the ganglionic eminence — they are too far apart. The lack of proximity means that there must be another way for communication to take place between these cells. This cellular synchronization could be achieved by: signaling through diffusion or another messenger type, such as process ingrowth; timing through gene regulation; or even exchange of information via a messenger in the cerebral spinal fluid in the ventricle. Control of early proliferative events in the ventricular zone is provided by patterning of signaling molecules such as FGF8, SHH, WNTs and BMPs [178, 154]; intrinsic regulation is achieved by the expression of transcription factors such as PAX6 and TBR2 [59], TLX [161], OTX1 [66], FEZ1 [37], CUX1 and CUX2 [207, 138, 48], SATB2 [30], NEX [204], NGN1 and NGN2 [166, 121] as well as non-coding RNA Svet1 [183]; or external regulation through diffusible messengers such as retinoic acid [173] or signals provided by axons descending to the germinal zones.

2.1.8 Migrational Processes

Radial Migration

The progenitor cells of the radial glial cells have fibers that anchor them at the apical end at the ventricular surface and at the basal end at the pia [152]. These radial glial fibers continue to grow over the course of the development, making space or probably rather forced to make space for the inward migrating cortical neurons. The radial glial cells produce neurons one layer after another in the inside out fashion described. These newly born pyramidal neurons then undergo radial migration.

The radial migration is executed in four basic steps. First the neuron finds a radial glial fiber. Secondly the neuron adheres to it. Thirdly, it migrates up in the direction of the pia, through the germinal zone, the subplate, and the already established parts of the cortical plate. Fourthly, it receives a stop signal and releases from the fiber. The signal is supposed to be given by the marginal zone [118] [134] [152]. If the migration process is disturbed and the release signal or guidance signal never arrives at the pyramidal neurons, the reeler phenotype is the consequence. The neurons seem not to migrate actively up the fiber and seem not to release from the fiber, blocking the way for later born neurons, only being pushed up passively. No migration through to the marginal zone is possible and therefore the cortex is inverted. The signal is generally known to be the reelin produced by the Cajal-Retzius cells of the marginal zone, though other disturbances of the radial migration process by other participating proteins have been reported [134]. Multiple neurons can use the same fiber at the same time (behind one another) building the cortex inside out. During the migration, the cells morph into a bipolar form extending apical and basal processes that later on become the axon (basal process) and the apical dendrite (apical process). Right after migration, when the pyramidal neurons have settled into the cortical plate, the outgrowth of the neurities begins. Axons and dendrites are formed [148].

This type of migration hints at the fact that the cells produced from one progenitor cell will also stay close together at their final position, giving rise to Rakic's radial unit hypothesis [153]. This hypothesis says that the cells born from the same progenitor will stay in the same cortical column, horizontally staying in the same location but vertically being part of different layers.

Tangential Migration

The cortical plate consists not only of excitatory neurons but also of inhibitory neurons that are thought to tune the neural network into a working regime.

In the mouse it is generally accepted that these interneurons are mostly born in the ganglionic eminence [18] [118] [129]. The ganglionic eminence interneurons are some of the best studied. There are other birth places of interneurons that will migrate into the cortex, but we do not consider them here. The ganglionic eminence is ventral to the developing cortical plate. It is sometimes subdivided into the Lateral Ganglionic Eminence (LGE), the Medial Ganglionic Eminence (MGE), and the Caudal Ganglionic Eminence (CGE) The timing of the birth of the interneurons in the ganglionic eminence is such that they start to be born at approximately E9 and finish at E15.5, having an onset just slightly earlier than the pyramidal neurons [18] (figure 2.1 process 1a). These times are again dependent on who did the experiment and how they counted the embryonic days.

Interneurons are born far away from the cortical plate and need to migrate into the cortical plate tangentially. Since they are born so far away they need somehow to settle in the cortex. But how do they find their radial and tangential position in the cortical plate? The radial positioning in the cortex seems to follow the following scheme: the radial target in the cortex for these neurons is birth time dependent much like that of the pyramidal neurons in the ventricular zone. Even more intriguing is that they share the same inside out positioning system as the pyramidal neurons born in the germinal zone. This means that neurons that are born early will end up in layer 6 and neurons that are born late will end up in layer 2/3 [18] However, it is still uncertain how many of the interneurons follow this regime of layer matching. But it is acknowledged that it exists [129].

How these neurons find their target tangentially in the cortex though has not been described. What is known on the other hand is that these neurons use two different migrational routes. They start at the ganglionic eminence. They avoid going through the striatum, but split up into two streams: a more medial and a more ventral lateral stream. They enter the developing cortex either through the intermediate zone or the marginal zone, i.e. above and below the cortical plate where they take up their tangential position [118] [134] [182] [168]. The interneurons arrive at their tangential position during the time the cortical layers are formed up to P1, i.e. just after birth, but before eye opening (figure 2.1 process 1b). This migration seems not to be guided by fibers the way the radial migration is. Having found their tangential position, the interneurons turn towards the cortical plate and migrate in radially either diving into the cortex from above or from below depending on the stream they have chosen. Even this radially oriented movement is not guided by any fibers. The radial positioning can take place up to postnatal day P7, that in the mouse [129] coincides approximately with the earliest eye opening date. It has been shown that the radial positions of the interneurons do not coincide with the zone they migrate through tangentially. One would have thought that the early born neurons take the medial route through the intermediate zone and the later born ones the route through the marginal zone, since the early ones target layer 6 and the later ones rather layer 2/3, however this is not the case [182]. The selection of the migratory route is unclear. They settle into the cortex and begin to form networks with the pyramidal neurons, inhibiting their excitation. [18]

The apparent interplay between the birth times of pyramidal neurons and their corresponding interneurons is very interesting. This interplay is happening at two locations which are very distant from one another. It is a mystery how the ganglionic eminence and the germinal zone know when the other is producing what. This matching of birth dates seems too well orchestrated for there to be no communication involved; somehow this must be timed. Is it timed through chemical means releasing a diffusible substance to inform the other component of the production change? Or could it be that the timing is accomplished through the developmental process by gene regulatory networks where each cell contains some sort of timing function? It has been shown by Sessa [168] that there are fewer interneurons in the cortex if the tbr2 cells in the subventricular zone are disturbed, because the chemoattractant source for the tangential migration is no longer present. This however only relates to the guidance to the cortex. It does not tell us how the proportions of interneurons and pyramidal neurons come to be what they are. How is the regulatory network of the ganglionic eminence and (sub)ventricular zone synchronized such that they produce the right proportions of neurons? Is this controlled by the germinal zones of the developing cortex? But then again the interneurons of the different layers tend to be born earlier then their counterpart pyramidal neurons in the cortex. The influence of the germinal zones on the ganglionic eminence in terms of cell production could maybe be tested in an experiment that would remove the cortical influence through some means and see whether the ganglionic eminence still behaves the same.

Another possibility would be that the communication takes place through the cerebral spinal fluid, since both progenitor pools are located at the ventricle allowing the release of morphogens through the fluid to communicate with the other. Thinking of evolution, somehow these areas must have evolved together to create functional circuits. If the cortical plate increases in size, the interneurons produced must also increase in number. This is harder to achieve if their gene regulatory networks have to be synchronized only through evolution and not through environmental communication signals.

Furthermore, are these gene regulatory networks somehow evolutionarily related? If so, the cells from the progenitors from the ganglionic eminence must share a common ancestry with the cortical progenitors from the germinal zone.

2.1.9 Thalamocortical and Corticothalamic Ingrowth

During the development of the cortex it is connected to a variety of areas in the brain. The thalamus as a source of sensory input is one of the most studied inputs. The development of this connection is very different in different animals [2] [69] [130] [42] [150] [127]. Even among rodents there are differences. We are trying to reconstruct the story of the thalamocortical connections in the mouse. In general the thalamocortical connection is established between E13 and E18 [112] (figure 2.1 processes 5, 6 & 7). There are three different layers involved in the corticothalamic connection. The subplate or layer 6b, layer 6a, and layer 5. The timing of which arrives where when is debated. We are going to rely mostly on the work of the lab of Molnar. Again this thesis is very much restricted to the area of the cortex we are discussing.

Thalamus' Projection to Cortex

The thalamus sends its axons in the reciprocal direction to those from the subplate. There are many nuclei in the thalamus that project to the cortex. Especially the LGN is often studied. At E13 the thalamic projections have reached the Internal Capsule (IC). (figure 2.1 process 5a). (How they are guided to the internal capsule is not discussed here.) There they meet up with the subplate axons [13] and establish the "handshake" of the handshake hypothesis [130] (figure 2.1 process 5b). There, the fibres fasciculate and they guide each other reciprocally to each others points of origin. Whether fasciculation actually takes place is debated. In opposition to the fasciculation theory there are reports of close but completely separated paths of growth of the corticothalamic and the thalamocortical fiber tracks [127].

By E15 they cross the pallial-subpallial boundary into the cortex and invade the subplate. By E18 they have reached the subplate (figure 2.1 process 5c). Different arrival times have been reported down to E14 [13]. At this point in the developmental story we arrive at the question of whether in mouse a "waiting period" of the thalamic axons exists. In primate and carnivores, a clear waiting period can be seen when the arriving thalamic axons gather in the subplate and only start to grow up into the cortical plate later [130] [127] [153]. In the subplate, these axons are supposed to influence the progenitor cells in the germinal zone to alter the production of neurons [51] and begin to invade the the cortex later when the layer 4 cells are starting to being born, where the migrating layer 4 cells might act as a signal to start growing into the cortex [127]. In mouse there might be a waiting period, but it might not be clearly visible due to the fast development in mouse compared to other animals [130]. It seems that the timing of cortical plate development and thalamic axon innervation of the cortical plate are highly dependent. In the mouse, the thalamic axons innervate the cortical plate right away and follow up the still developing cortical plate. By P2 they have reached layer 4 (figure 2.1 process 5d) and by P8 they have created larger arborizations in their target layers (figure 2.1 process 5e) [150].

The thalamic axons create arborizations in layer 6 and layer 4, though layer 4 is more heavily innervated [150] [127]. There are also other layers affected and connected to, but with fewer arbors [127] [13]. The arbors of the thalamus in the cortical plate are subject to heavy pruning and reorganization (which we will not go into in detail here) in the days before and after eye opening.

Subplate Projections to Thalamus

At E13, the subplate cells begin to grow out axons that follow the ventricular border parallel to the direction of the thalamus (figure 2.1 process 6a), and reach the pallial-subpallial boundary (PSBS) at E13.5 where they wait (E14) for the thalamic axons to arrive (figure 2.1 process 6b). At E18 we see the subplate axons reach the thalamus (figure 2.1 process 6c) [112]. Other sources report the arrival of the subplate projections (subplate axons / corticofugal projections) at the internal capsule at E14 and the establishment of a connection to the thalamus by E14.5–E15 [13]. This discrepancy in the literature might be because the layer 5 cells are also sending out corticofugal projections that are supposed to begin later in development but overtake the subplate projections and are supposed to arrive earlier at the thalamus. This overtaking of number of projecting fibers from layer 5 versus the subplate was reported in the hamster but is thought to hold true also in the mouse [127].

Layer 6a

Layer 6a or layer 6 afferents follow the thalamic afferents down to thalamus to the same nuclei that innervated their cortical region [112]. The layer 6 growth is thought to be guided by the thalamic afferents. Whether this guidance is due to fasciculation is debated [127]. The growth of layer 6 afferents to the thalamus happens after the outgrowth of subplate cells up to P8. They begin with the outgrowth directly after having settled in the cortical plate around E13.5–E14 (figure 2.1 process 7a) [51] [148], but might grow at different speeds or with pauses during the development [127]. At P0 the first axons of layer 6a and subplate arrive at the thalamus (figure 2.1 process 7b). At P3 though, the innervation of layer 5 becomes stronger in terms of numbers of connections than of layer 6a. And at P7, layer 6 overtakes layer 5 again in numbers of connections [127].

Layer 5

Besides layers 6 and subplate, layer 5 also projects to the thalamus, but in a slightly different regime. Layer 5 is probably not guided by the thalamocortical projections but rather by other cues. That means that layer 5 axons seem not to fasciculate to thalamic afferents [112] [127]. This non fasciculation of layer 5 makes sense, since layer 5 neurons do not target exactly the reciprocal nuclei that the thalamus is projecting to them from [127] [42] [112]. Layer 5 axons also project to the spinal cord, collateral hemisphere or the brain stem [42]. Layer 5 projections seem to obey very different rules than layer 6 projections. And the layer has a completely different type of corticofugal projection. The layer 5 neuronal fibers will exceed the number of layer 6 fibers to the thalamus at about P3 and become the most numerous of of the corticothalamic projections, though around P7 subplate and layer 6 will gain in numbers of connections again and overtake layer 5 once more [127].

2.1.10 Arealization

The cortex, even though similarly constructed over the whole plate, serves many functional purposes. These different functions are thought to be processed in different areas of the cortex. These areas reflect that they serve different purposes in their anatomy. Layer thicknesses are significantly different at different locations. And projections to other brain areas also vary by area. In the fully developed cortex explicit areas can be identified. These areas must develop somehow. There used to be the idea that all of cortex is pluripotent. The areas would only arise because of the input that was provided by projections from other brain areas such as thalamus, each part of cortex being able to create any cortical area. This was known as the protocortex theory [152].

Nowadays the protocortex theory has been disproven by transplantation experiments and the protomap theory has emerged. The protomap theory hypothesizes that the cortical plate is split into areas before the cortex emerges and before projections of axons are formed. These protomap areas are built by gradients that are developing during the development [152] [51]. The number and diversity of neurons in each layer or area is directly correlated with changes in the rate of neuron production, that is the control over neural birth time [52] [113]. Two cell-cycle parameters are especially important for determining the number of neurons produced: the rate of cell-cycle progression and cell-cycle exit, which are determined by the control over cell cycle machinery and differential expression of transcription factors in daughter cells. Or that different parts of the gene regulatory network are active at different regions in the cortex.

But the individual areas are still proven to be subject to the influence of the ingrowing axons from other areas [103]. The thalamic influence on the areas of cortex has been impressively shown by removing the thalamic ingrowth to the cortical areas 17 and 18 in primate [99] [98]. One might imagine that the fibers exercise influence through signals to the proliferative cells that would then adapt their gene regulatory network adjusting the developmental program slightly and spawning different numbers of cells in different areas [147]. In the case of the formation of areas, again the timing of certain events is crucial to the normal development of the cortex. The borders between these areas are reported to be very sharp [99] [147].

2.2 From Genes to Cellular Behavior

2.2.1 Gene Translation and Transcription

Proteins are constructed according to the DNA information through a decoding process comprising two stages: transcription and translation. Transcription is the process that transcribes information contained in the DNA into a single strand of messenger RNA (mRNA) and translation describes the process of converting information in the mRNA into a protein. The entire process consist of a sequence of processes:

- The transcription complex binds to core promoter regions, just upstream of the coding sequence of a gene.
- If unhindered, the transcription complex scans the DNA and produces mRNA.
- The transcription complex is released from the DNA strand at the stop coding sequence and is ready to begin transcription again.
- The mRNA is used as a template by ribosomes to build proteins.

The transcription of every gene can be selectively regulated by transcription factors. Transcription factors bind to regulatory regions upstream of the core promoter sequence. Multiple transcription factors can bind to the regulatory region and modulate the activity of the transcription complex by either enhancing or repressing the transcription rate. Every protein embodies both a function (influencing the behavior of other components) and a signal (information that can be read by other proteins). In the case of transcription factors we are interested in the signal property because it can be read by the transcription complex, telling the complex how much to express its target protein. If the transcription factor also has an active part that influences the behavior of the cell, we look at it as a separate entity taking part in a gene-translation machinery. There are proteins where we are not interested in their information content, but only in their active component. But these nevertheless provide at least the one bit of information as to whether they are present or not.

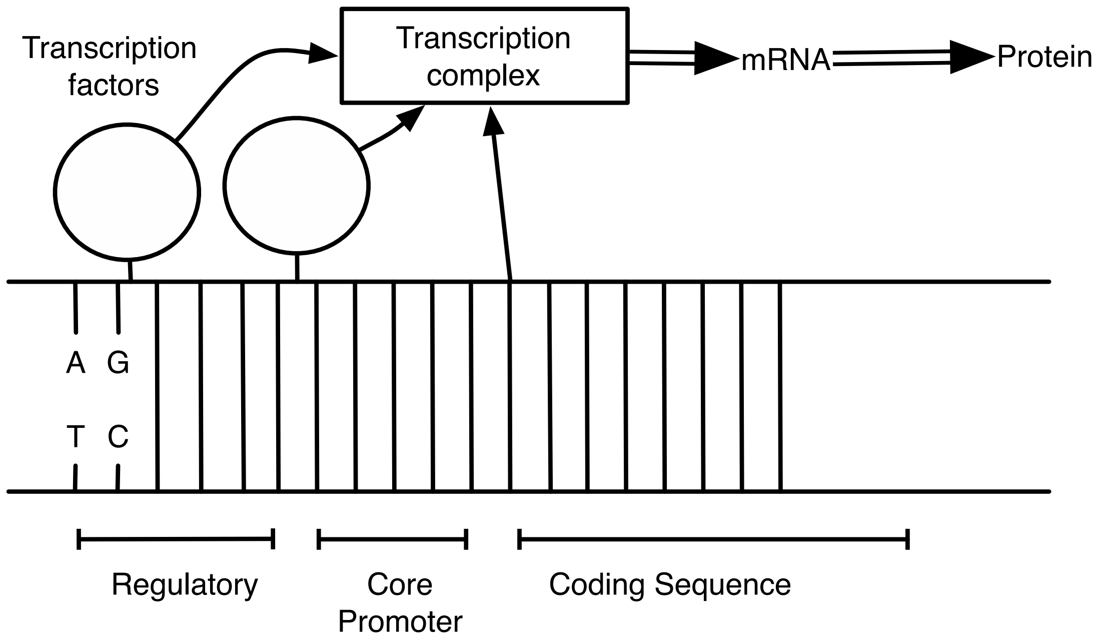


Figure 2.2. The transcription of a gene. Transcription factors connect to the DNA at the promoter region of the gene and bind to a transcription complex that then produces mRNA from the gene coding sequence. This mRNA will then be translated into a protein.

In our abstraction the DNA follows this dual view. The pure information in the codons of the DNA can be seen as a passive signal to be read. On the other hand the DNA has a vey definite physical structure. This structure is active, proteins can bind to the structure and operate with it. DNA alone can not perform any transcription, it needs the whole machinery of the transcription factors and transcription complex in order to translate a gene sequence into a specific protein. The combination of transcription factors and transcription complex form a specific production complex for every gene. This combination is only able to work on very specific genes.

We call this gene specific complex the production machinery. The production machinery can only form on the DNA and is not separable from the gene's specific position on the DNA. Once formed, the production-machinery will produce the protein encoded by the gene. We neglect here the detailed transcription, splicing and translation processes and just name this process *production*. This machinery bound at this very specific point on the DNA is the only way to access the information of this particular gene. Whenever the corresponding protein is needed this machinery has to be activated in order to produce it. Therefore the gene-specific sequence can be associated with this production machinery as we see in figure 4.1

The different transcription factors can only bind in a very specific way to each other expressing a quasi-logical mathematical function AND, OR and NOT functions. This defines how much of the protein can be produced by the transcription complex. The concentration of a transcription factor defines the probability of binding. The higher the concentration the more the transcription factor will bind and the more it can influence the production of the protein.

2.2.2 Gene Regulatory Network

During cortical development dividing cells undergo progressive fate restriction, that is a restriction in the types of differentiated cells that a progenitor can produce [86]. This information is represented in the form of cell lineage trees, which describe the developmental trajectory in the form of a binary tree: the root is the initial precursor cell; the terminal nodes are cells that have reached a terminal phenotype; and the tree topology represents the relationship between all cells that exist at given time point during development. Formally the lineage description defines the reachable states (cell types) in which a cell can be found in, and the possibility of transition between those states.

How the ordered sequence of cell divisions through the cell lineage leads to terminal cell fates is dictated by the transcriptional network, which is in turn regulated by levels of transcription factor expression, epigenetic mechanisms and cell-cell signaling. Genetic regulation consists of networks of genes and their protein products, or transcription factors (TFs), which can influence each other's expression over time by binding onto specific gene regulatory regions [110]. Stable profiles of gene expression represent defined attractors and can be interpreted as distinct cell fates [97] [87]. During mitosis the genetic information is replicated with high fidelity and equally distributed to the two daughter cells, though the cells are not necessarily identical. The content of the mother can be distributed asymmetrically or the local environment may be different, causing a difference in morphology or behavior. As a result different pattern of division are observed: symmetric proliferative (daughter cells are identical and continue proliferation), symmetric differentiative (daughter cells are identical and both differentiate), asymmetric proliferative (daughter cells are not identical in their fate but will both continue to proliferate), and asymmetric differentiative (daughter cells are different, and either one is or both are post-mitotic).

2.2.3 An Abstraction of the Gene Regulatory Network

The generation of different cell subpopulations during cortical development is the result of concurring complex processes, which involves a variety of different possible cell states and transitions between those states. A representation of the sequence of cell divisions and differentiations is provided by the genealogical history of every precursor cell, the cell lineage tree. From experiments in which the progeny of single progenitors was tracked using radioactive tracers [147], we can extrapolate the sequences of cell division and differentiation that lead to particular distributions of neurons in different area of the murine cortex.

Cell decisions regarding acquisition of an appropriate cell fate relay the ability to commit to different stable states. These decisions are controlled by transcriptional networks. Gene regulatory networks consists of networks of genes, which can either code for transcription factors (which we represent as passive substances) or G-Machines (functional proteins that will trigger complex enzymatic functions). Transcription factors influence each other's expression by binding onto specific gene regulatory regions: many genes are controlled by a number of different transcription factors and different arrangements of binding sites can compute logic operations on multiple inputs.

We choose to express the dynamics of transcription factor binding with a multilinear polynomial function. The function describes the binding of different transcription factors to a promoter region and their interactions and can be seen as a tree of continuous Boolean logic gates that take as input the transcription factor concentrations. We represent the DNA information of the gene regulatory network with a high level G-Machine, which has a list of all genes and all regulatory sequences. Each gene is characterized by a core promoter region responsible for the transcription of the gene coding sequence, and a regulatory sequence composed of a combination of binding sites. Every gene actively computes its transcriptional activity depending on the concentrations of binding transcription factors and their interactions, and actively produces either a transcription factor or a functional protein (figure 2.3). Additionally, the genetic network G-Machine contains an abstract representation of the cell cycle molecular machinery, which controls the progression through the cell cycle in the form of a timer. Cell fate decisions are taken at the moment after cell division, when cells compute their current state based on the concentrations of transcription factors they inherited from the mother cell.

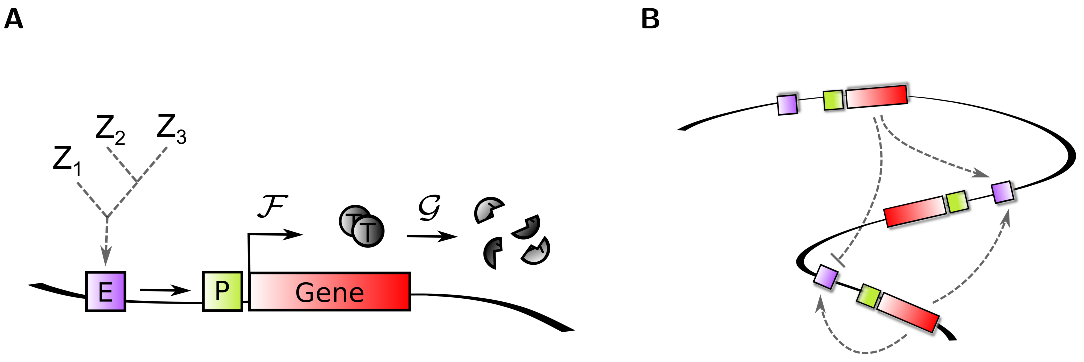


Figure 2.3. Transcriptional network. (**A**) A typical eukaryotic gene consists of a coding region (Gene), a core promoter region (P), which is the minimal sequence of bases required to properly initiate transcription, and a *cis*-regulatory sequence (E), a proximal sequence upstream of the gene that contains primary regulatory elements. *Cis*-regulatory modules receive and process informational inputs in form of transcription factors. Function Z describes the binding probability of transcription factor T or any other transcription factors to the E sequence. Function $\mathcal{F}$ computes the protein T synthesis rate given a combination of enhancing or repressing transcription factors bound to E, whereas $\mathcal{G}$ describes the dynamic of regulated degradation. (**B**) Example of a small regulatory network of genes with interactions between gene regulatory regions and transcription factors.

The simplest system that can control state transitions is a bistable switch [139] [88], in which two transcription factors X_1 and X_2 negatively regulate the expression of the other and generate two stable states, given a constant input. In the first state, the gene for one repressor is turned *on* while the synthesis of the second repressor is turned *off*. The reverse is true for the second state. A third meta-stable state forms at the border between the two attractor basins. In our simulation a set of interacting bistable switches builds the gene regulatory network.

The translation of state lineage graphs into a gene regulatory network follows some simple rules. Every cell division is controlled by a bistable switch, which depending on some parameters will determine how the daughter cells will divide. At mitosis cells can either divide symmetrically and stay in the same position in the expression space or the concentrations can jump depending on the values of the asymmetry constants. The effect on the cell division outcome is described by the parameter ω, which describes the angle of the division in respect to some arbitrary reference. The values taken by ω completely describe symmetric and asymmetric cell divisions, since cells will fall into different attractors. This description is completely deterministic and will always generate instances of the lineage tree that are all identical with each other. A more interesting case is when the state diagram does not represent exact transitions but is instead probabilistic. In the genetic network, probabilistic transitions are implemented by making asymmetric cell divisions stochastic, that is ω represents the probability of division with a given angle.

In our simulation the value assumed by ω is constant. This is very simplistic, since a relaxation of this assumption creates many interesting effects. Firstly changing asymmetry constants means that although the sequence of states is determined genetically by the interaction links in the regulatory network, the total number of different cell types produced can be modulated by external factors for example during cell-cell interactions. Furthermore the asymmetry constant could also be a function of the state generation or the cell age, which would further constrain the evolution of the tree and avoid the disadvantages of Markov Models, namely the high variability in the output if the state transition probabilities are very low.

In the simulation an initial precursor pool of symmetrically dividing neuroepithelial cells, which display radial glial fibers, divides into two radial glial cells, one of which will be maintained till the end of corticogenesis. We are not currently simulating the gliogenesis from radial glial cells, given that this was not measured by Pollaux [147] and we thus don't have quantitative data on the process. We are therefore employing a simplification in which we use a fixed population of glial cells that form the initial scaffold for migration within the cortical plate and are kept until the end of neurogenesis. The second radial glial cell daughter will either divide symmetrically or asymmetrically producing intermediate precursors or terminally differentiated neurons. The expression of transcription factors related to the intermediate precursor fate triggers the activation of machines responsible for the migration in the subventricular zone just above the ventricular zone (figure 3.10), whereas expression of transcription factors specific to different types of neurons will trigger the exit from the cell cycle and the expression of the radial migration G-Machine. There is a trade-off between the accuracy of the distribution of neurons generated and the number of genes that constitutes the gene regulatory network. For illustrative purposes, we use here a relatively small network that has the property of coarsely following the cell distributions while being still visually understandable.

In the simulation we activate two different gene regulatory networks based on patterning of the initial neural plate by two morphogens. The two gene regulatory networks are almost identical in topology (elements and links), but display different parameters, which reflect differences in the cell cycle parameters for the production of areas 3 and 6 in the mouse. The gene regulatory network was further manually modified to enhance the visual difference between areas 3 and 6.

2.2.4 G-Code: An Instruction Language for Cells

Cells have a range of different behaviors. Their behaviors are consequences of proteins interacting with each other. Proteins that are active, in the sense that they take part in the metabolism, give rise to the behavior we observe in biology. Observable behavior is not exclusively a consequence of a single expressed protein but of a network of proteins interacting with each other. These behaviors can be abstracted into behavior modules. Modules are responsible for functions that an observer would consider to be one functional entity. In G-Code [210] we capture the idea of these behavioral modules in an abstraction called a G-Machine. G-Machines are composed of primitives that can be linked together. In figure 2.4 the primitives are given in a list along with their meanings. All of these primitives in G-Code are biologically plausible in the sense that there are mechanisms described that allow for these primitives. Primitives therefore express a subset of the behavior that the G-Machine is expressing, and are therefore also associated with proteins, abeit qualitatively rather than directly.

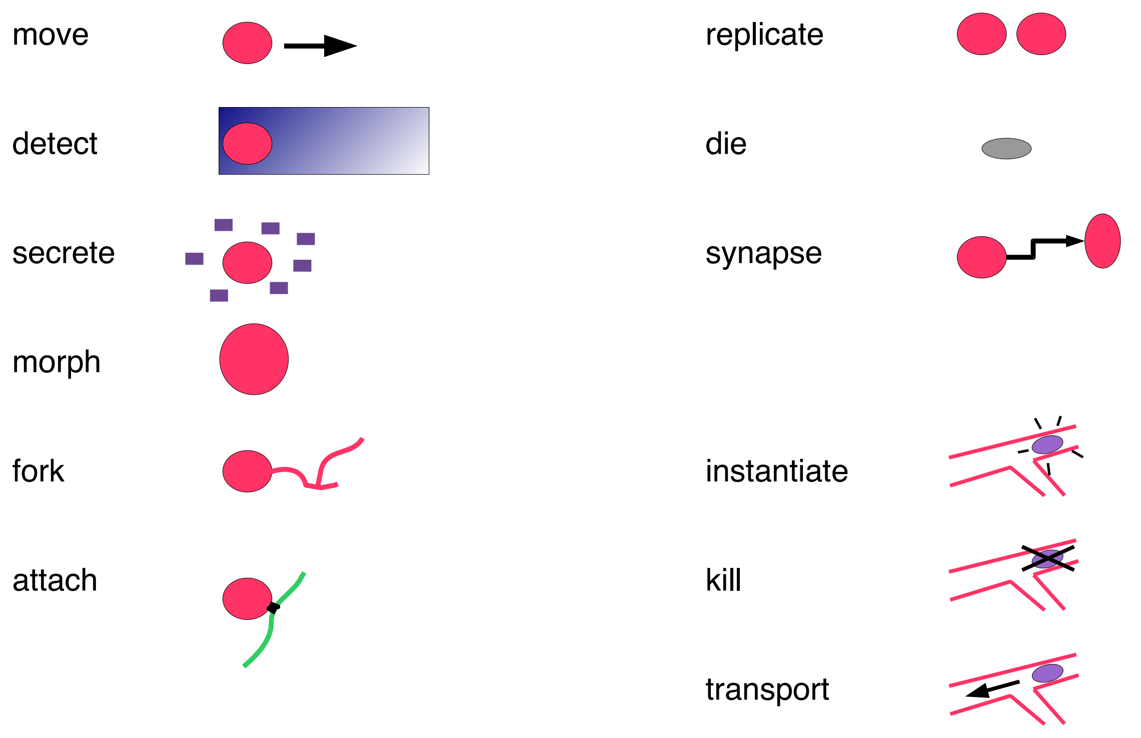


Figure 2.4. These are the potential primitives that can be interlinked in a G-Machine: move for the translocation of a soma/neurite; detect for the detection of environmental /intracellular chemicals; secrete for the production of intra/extra cellular chemicals; morph for changing morphological properties of the soma/neurite; attach for establishing physical bonds to other objects in the vicinity; replicate for cellular proliferation; die for apoptosis; synapse for establishing a synaptic connection to another cell; instantiate for invoking the translation of another G-Machine; kill for stopping a G-Machine; transport for transporting a G-Machine between different cellular compartments.

The G-Code framework is written explicitly for the simulation environment Cx3D that tries to give a biophysical environment for tissue growth. A G-Machine is running either in a soma of a simulated neuron or in one of the neurities or its sub-branches. Once defined, G-Machines can be reused. A program written in G-Code is the equivalent to a DNA strand in a real cell. A G-Code program is a sequence of descriptions of G-Machines in their textual inactive form and does not influence the cells in this form. Each simulated cell in Cx3D has access to the G-Code program and can if the necessary conditions arise express G-Machines from the G-Code. This allows, as in biology, that all cells have the same "genetic" code available but only express the necessary parts of it for the current task at hand. The G-Machines are explicitly written by an engineer to define what the cells have to do in what situation. And therefore to define the developmental program the simulated organism undergoes during its course of execution. In figure 2.5 we give an example of a G-Machine that is translated into action.

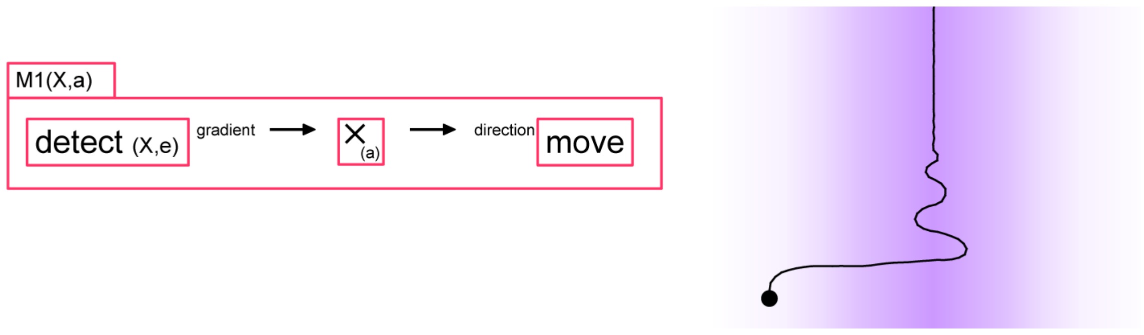


Figure 2.5. An example of a G-Machine that is inserted into the growth cone of a neurite. The G-Machine detects a gradient and makes the neurite grow towards its max value with two primitives that are linked through a signal processing filter. Detect and move are these two primitives. Left: the assembled machine as a cartoon. Right: the executed behavior.

With this system we are in the position to describe a developmental process in Cx3D that begins from one cell and ends in a whole organism, respecting the biological limitations while using its possibilities.

Chapter 3

Modelling of Cortical Development of Mouse Area 3 and 6

This chapter describes the model that has been used in order to simulate the cortical growth of the mouse cortex. Each of the G-Machines are explained. To make this chapter comparable to the underlying biological findings, it follows the same structure as the biological background chapter 2.

3.1 Simulation and Modeling

We will use 'simulation' to denote the process whereby a general purpose computer (GPC) reads an encoded model and its data; executes the model using the data provided; and produces the model's output. In this view the simulation is a non-specific methodology, and bears no scientific relationship to the content of the model being simulated. Thus a simulation cannot independently evaluate the correctness of the model that it is executing: the GPC will simulate correct models as well as incorrect ones. By contrast, we use 'model' to refer to the scientific process being explored. The model specifies the organization and dynamics of the system under investigation. Usually the model can be seen as a Kolmogorov-like compression of experimental data into a more compact form consisting of an algorithm plus its input data, which on execution is able to regenerate the uncompressed experimental data.

In order to simulate a model, the model must be encoded in form which the simulator can consume, and the resulting output of the simulator must be decoded into the model domain. This output of the simulated model is expected to agree, within some tolerance, with the experimentally observed target process. This agreement is taken to provide support for the explanatory model tested. However, the explanatory strengths of such models vary widely. For example, a model that trivially returns as output the same series of observations that it read as input is formally correct in its simulation of the data, but this model has no explanatory power because it provides no compression of the data and does not generalize beyond them. Similarly, but less trivially, models of development that assume pre-labeling of physical and functional space and then use forward models to essentially draw neuronal morphologies in that space, may yield satisfying shapes, but such models do not get to the heart of the distributed autonomous nature of biological self-construction.

Models of neural development have focused directly on subcellular processes such as gene expression and protein networks, rather than the relationship between the subcellular processes and the emergent cellular and organismal behavior that they generate. We have explored this important relationship in the context of neocortical development, using a simulation framework in which cellular mechanisms act in a physically realistic three-dimensional environment and so express the overall self-construction of the cortex. This process is steered by genome-like instructions inserted into cortical progenitor cells. This code controls the mitosis, migration and differentiation of individual cells, and thereby gives rise to the collective process of development in a biologically plausible manner.

3.2 Simulation Results: In Silico Simulation of Mouse Corticogenesis

3.2.1 Cx3Dp

Cx3Dp (see chapter 6) is a software platform that provides for the simulation of models of neural development and growth in a physical 3D environment. CX3Dp respects biological processes such as cell division, cell-cell interactions, movement, and the secretion and detection of membrane attached or diffusible signal molecules. For reasons of computational discretization, cells are abstracted as spheres; and neurons are discretized as spheres (somata) that give rise to trees of connected compartments (dendrites and axons). These spheres and compartments behave as physical objects in 3D space. Forces ensure that objects do not overlap in space; compartments are able to grow, and extend or contract; and chemicals secreted by cells are able to diffuse through extracellular and intracellular space. Cells are able to move by traction against the intracellular matrix; they can anchor themselves to the matrix; and also form physical connections such as bonds or synapses with one another.

Each cell is an autonomous agent exerting only local actions, and using only locally available information. The behaviors of the simulated cells are determined by intracellular molecular-gene-like codes that are expressed according to intracellular or extracellular conditions. For specifying this ‘genome’ we introduced G-code [213], a formal language based on a set of primitive neural actions. The G-code of a cell is an inert informational structure that specifies networks of these primitive instructions, and so provides potentially complex functions to cells. Under appropriate physical conditions, various coherent regions of this G-code will be instantiated (expressed). The expression products of G-code are intracellular, spatially distributed G-machines composed of the primitives as specified by the relevant region of G-code. The overall behavior of the cell at any time is the collective result of its currently instantiated G-machines.

Cx3Dp is abstracted into three conceptual layers: a spatial layer that maintains the spatial location of objects and their neighbors; a physical layer, that expresses the physical interactions between the objects; and a biological layer that expresses biological function and can be modified depending on the model that is run in Cx3Dp. Cx3Dp is optimized to create models for biological processes that respect the locality of these processes. Each compartment in a model only has access to the neighboring compartments and never to the global structure of the whole simulation, so constraining the model to remain as biologically plausible as possible.

Space is simplified to be a 3D extracellular matrix. The extracellular matrix allows for growth and migration of any cell in any direction. However, because a modeler may wish to simulate the behavior of a subpopulation of cells that is embedded within a larger population, or abuts other cell populations, Cx3D allows for the specification of a bounding box that constrains cells to a volume of space, and can be used to provide suitable boundary conditions for the simulated cells.

All actions possible for a cell arise out of mechanisms that must be explicitly declared by the modeler. For example, cellular processes such as mitosis have impact on the cell shape: the two daughter cells have smaller volumes (roughly half) than their mother. Consequently, if the daughters should each regain the volume of their mother, then the modeler must explicitly specify a cellular mechanism that causes the cell to grow towards a steady state (if required). In the absence of such a growth mechanism, repeated divisions would lead to ever smaller cells and the entire population would have a volume nearly equivalent to the initial progenitor cell.

3.2.2 A Simulation of Murine Cortical Development

A video overview over the entire simulation can be seen at http://youtu.be/pBqZ8SouWdQ ‘Development and 3D rotation of selected neurons’. The video is composed of successive images of simulation data. The images are chosen to illustrate various aspects of the simulation, and do not show complete data. For example, for clarity some cell types may be suppressed during image rendering. In general, there is very much more data in the simulation than the selected video images depict. Similarly, the figures in the description below are snapshots taken during the same simulation used for the video, and the particular snapshot rendering is chosen to illustrate a specific aspect of the developmental data.

Initial Arrangement of Progenitor Cells

For practical and computational resource reasons, we do not model the transformation of the entire neural tube to a complete brain. Instead, we choose as our starting point a small region of neuroepitheium located in the telencephalic pallium [131] [142] [153]. These neuroepithelial cells are the progenitors of the cells that will form the fundamental laminated neocortex of areas A3 and A6 [147]. In the mouse these pallial neuroepithelial cells give rise to the excitatory, pyramidal type neurons of cortex, as well as neuroglial cells. By contrast, the inhibitory neurons are derived from precursors in the ganglionic lateral and caudal eminences of the basal telecephalon[129]. These precursors must migrate tangentially from the eminences into the developing pallium[134] [18] [182]. Because our model does not yet consider the development of all the embedding brain structures, we simplify the eminencies as a free-standing group of progenitors displaced from the pallial neuroepithelium. Figure 3.1 shows how the neuroepithelial cells (gray) of the A3 and A6 VZ are laid out as a monolayer plate. The ganglionic eminence population (yellow) is located at the left ('medial') end of the VZ plate. In addition, we consider a representitive population of thalamic precursors (black), which are located some distance below the eminence. The relative locations of these three populations approximate very roughly their relationships in the developing neural tube. The absolute numbers of cells in these three populations are not known experimentally. Numbers were chosen that are qualitatively reasonable initially, and lead to ratios of mature cell numbers that are close to what is observed experimentally [23].

The developing neural tube contains a number of neuromorphic signal gradients that developing cells may use for their orientation [134]. Therefore, our model assumes two embedding gradients along the long axis of the VZ. One gradient increases linearly from left (medial) to right (lateral) and the other is disposed in the reverse direction (figure 3.1, planar view). They cross in the middle of the neuroepithelial plate. The boundary conditions asserted by adjacent regions of un-modeled neurons, are imposed by a bounding box that provides the necessary boundary force due to the apposing cells (figure 3.2).

G-Code

Each cell is an autonomous agent exerting only local actions, and using only locally available information. The behaviors of the simulated cells are determined by intracellular molecular-gene-like codes that are expressed according to intracellular or extracellular conditions. For specifying this 'genome' we introduced G-Code [213], a formal language based on a set of primitive neural actions. G-Code is now in its second revision (see Section 2.2.4 and Appendix A).

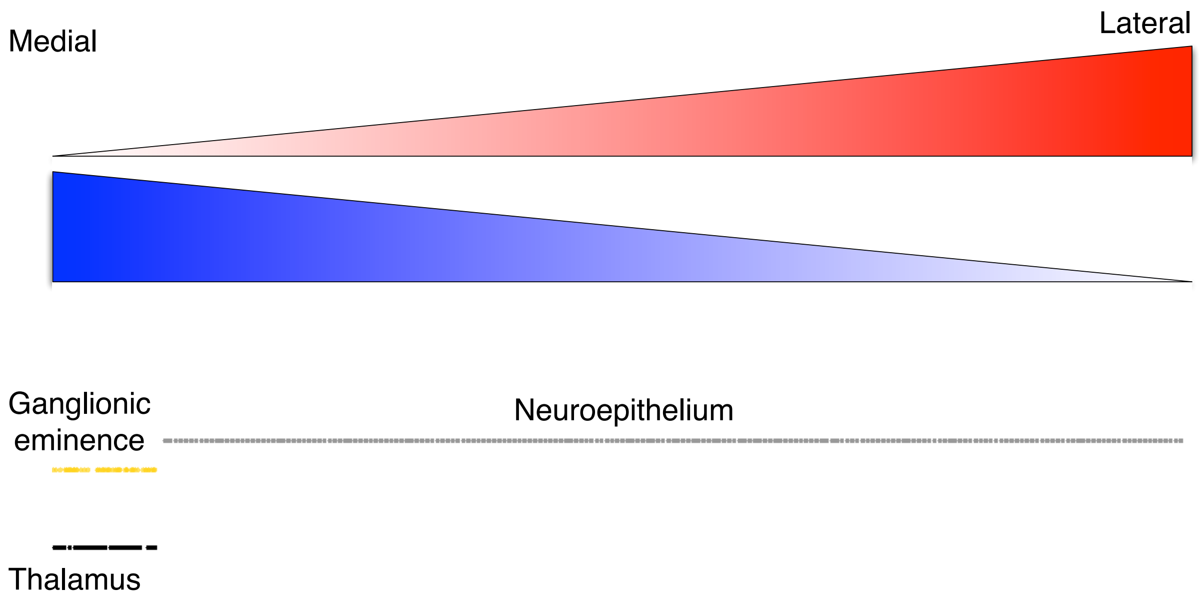


Figure 3.1. The starting point of the simulation. In yellow are the simulated cells of the Ganglionic Eminence, in black the simulated cells of the thalamus and in gray the neuroepithelial cells. There are two gradients (blue and red), from left to right and from right to left. From these cells the cortex will be constructed.

Figure 3.2. Black dashed line: The force boundary that is applied to the cells that enter this zone. This artificial boundary is a means of the simulation to ensure that the cells do not escape into the open space where there is no simulation relevant objects around. In biology the space around the cells would be filled with other cells, but we do not have the means to simulate the complete brain structures and therefore have to introduce artificial boundaries that support the simulation.

G-Code allows the modeler to design and encode cellular mechanisms in a biologically plausible way. The G-Code of a cell is an inert informational structure that specifies networks of the primitive instructions, and so provides potential complex functions to cells. Under appropriate physical conditions, various coherent regions of this G-Code will be instantiated (expressed). The expression products of G-Code are intracellular, spatially distributed G-Machines composed of the primitives as specified by the relevant region of G-Code. The overall behavior of the cell at any time is the collective result of its currently instantiated G-Machines.

The G-Code of a cell can encode many G-Machines, and any number of these machines can be expressed within a given cell. However, the operation of a G-Machine is restricted to exactly one cellular compartment and it can control the behavior of only that compartment. Thus, G-Machines automatically respect the locality property, meaning that these mechanisms can only act locally in each cellular element and cannot exert global control or effects. This means that if machines that are located in different compartments of the same cell, or in different cells, and they must co-ordinate their activities, then explicit signalling mechanisms (machines) that provide the necessary physical communication must be G-Coded and appropriately instantiated. Thus, G-Code permits the modeler to easily design biologically plausible mechanisms, but at the same time G-Code prevents the modeler from violating the essentially physically localized nature of these mechanisms. The clear separation between technical simulation support, and actual model implementation leads to some interesting effects. For example, the coloring of cells for visualization can be generated by a GRN controlled G-Machine which expresses the 'color' of a cell type as if it were expressing a protein such as GFP. In the following description, for simplicity, we explain the operation of the G-Machines in words rather than using their formal definition. Since we will often wish to differentiate between specific G-Machines, we will for convenience use the notation G-Machine(f), or GM(f) to refer to a G-Machine that performs 'f'.

3.2.3 Preplate

Since all three initial populations derive from the same neural stem cell, they must all contain the same G-Code. Thus, in principle, the entire genome is available to each and every cell. If we chose to model development from the original stem-cell, the simulation would begin with one cell having access to its G-Code and one (possibly more) cell-type specific G-Machine activated in it. The fact that different cell types arise during the developmental sequence is a consequence of the GRNs of cells expressing a lineage of different states according to the various intra- and extra-cellular conditions that arise through the developmental orchestration. Because our simulation begins with three different cell types, the cells of each of those types must be initialized in the GRN state that expresses the appropriate cell-type G-Machine(s). This means the model has three starting points: one for the neuroepithelial cells [2] [131] [142] [153]; one for the cells of the

Ganglionic Eminence cells [134] [18] [182] [118]; and one for the thalamic cells [127] [150] [130].

The preplate is the first structure to emerge from neuroepithelial proliferation. Each neuroepithelial cell undergoes about 14 divisions. Each division leads to one neuroepithelial cell, and one preplate cell. The first seven divisions generate the denser upper part of the preplate that will become the marginal zone, while the later seven divisions give rise to the more loosely packed lower part of the expanding preplate that will later become the subplate. The physics of cell-cell forces cause the first-born cells to rise to the top, and the later born cells to remain at the bottom of the preplate. As the presumptive MZ cells stack upward in the cortex, they begin to form bonds between themselves and so stick together. Here, our model deviates from observed biology in two ways. Firstly, no pia is created. Instead, the topmost cells become a surrogate for the pial-boundary. Secondly, the Cajal-Retzius cells arise from the underlying neuroepithelium rather than the cortical hem [124]. This process is shown in figure 3.3, and the simulation of the subplate in figure 3.4. The dynamics can been seen in the video http://youtu.be/l6BOL4suYYg Cortical Plate development.

The neuroepithelial G-Machine that governs preplate repeats the sequence: ⟨Grow cell; divide asymmetrically⟩ through about seven cell cycles. Each differentiating preplate cell exiting from the mitotic cycle instantiates a marginal zone G-Machine(MZ) that characterizes an MZ cell type [2]. This machine performs the following operations:

- Instantiates a further machine G-Machine(MZbond) that creates physical bonds between at most two neighbors of the same type. The overall effect of this bonding is to form a tight mesh of bonded MZ cells.
- Instantiates a growth G-Machine(grow) that causes cells to grow to their target size.
- Instantiates an apoptosis G-Machine(apoptosis) that checks continually whether the cell is currently in the vicinity of other cells of the same type. If there are too few neighbours of the same cell type, then the cell considers itself to be misplaced and undergoes apoptosis. If this machine has not detected misplacement over a period of some hours it will terminates itself and the cell continues its existence. Such conditional apoptosis makes the overall assembly of the cortex more stable.

The lower half of the cell stack born from the mitotic neuroepithelial cells are the presumptive subplate cells [2]. We do not assign any function to these cells other than their mechanical expansion of preplate. We therefore simulate only a small portion of the future subplate cells, but each cell will occupy more space than a regular cell and has a larger forcefield around it to make up for the small cell numbers but to achieve nonetheless a large expansion of this zone. The G-Machine started in these cells begins the growth of the forcefield of the cells. We do this

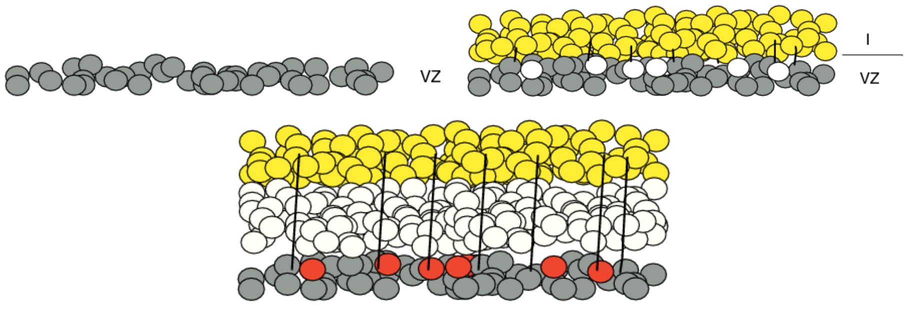


Figure 3.3. Top left: a cartoon of the development of the neuroepithelial cells (gray). Top right: the birth of the future marginal zone cells (yellow). Bottom: the subplate (white) with the first layer 6 cells being born (red). These cartoons indicate the initial development of the preplate that will later on support the formation the cortical plate. The neuroepithelial cells will develop into radial glial cells and generate the cortical cells of layers 5 and 6 and will create also the subventricular zone.

to reduce the simulation load, since we are not yet simulating the biological function of the cells. The growth G-Machine will also be started in these cells to expand their volume to their final size.

The mitotic neuroepithelial cell line becomes the ventricular zone during the development of the preplate (see figure 3.5) [131] [142] [153]. These cells then begin to grow fibers radially in the direction of the surrogate pia [77, 128, 140, 141, 7]. It is these fibers that characterize the radial glia cells. These radial glial fibers extend upward until they reach the cells of the presumptive marginal zone. The overall preplate is expanding in thickness, and this raises the question of how the radial fibers adjust their length accordingly. One possibility is that the distal ends of the fibers are firmly attached to the MZ cells, and simply stretch as the preplate thickens. An alternative is that the fibers continue to grow actively in length, their tips holding station in relation to the MZ cells. We chose to use the latter mechanism as follows. The fiber G-Machine(fiber) of the radial glial cell extends its fiber radially toward the MZ in much the same manner as the growth cone G-Machine(gc) of a neurite. The tip idles (but does not terminally stop) growing when it senses the MZ (surrogate for the pia). The tip also idles on the condition that there are no cells surrounding it, in which case it has overshot the MZ layer. If the MZ cell layer continues to be pushed upward past the fiber tip, the idle condition lapses, and the tip grows upward to touch the MZ layer again. This active elongation machine allows the radial glial fibers to hold station in the MZ despite the expansion of cortex. Figure 3.6 shows this process. The simulation of the radial glial fiber extension can be observed in the following video: http://youtu.be/ligAWNjcFEw ‘The development of the radial glial fibers’.

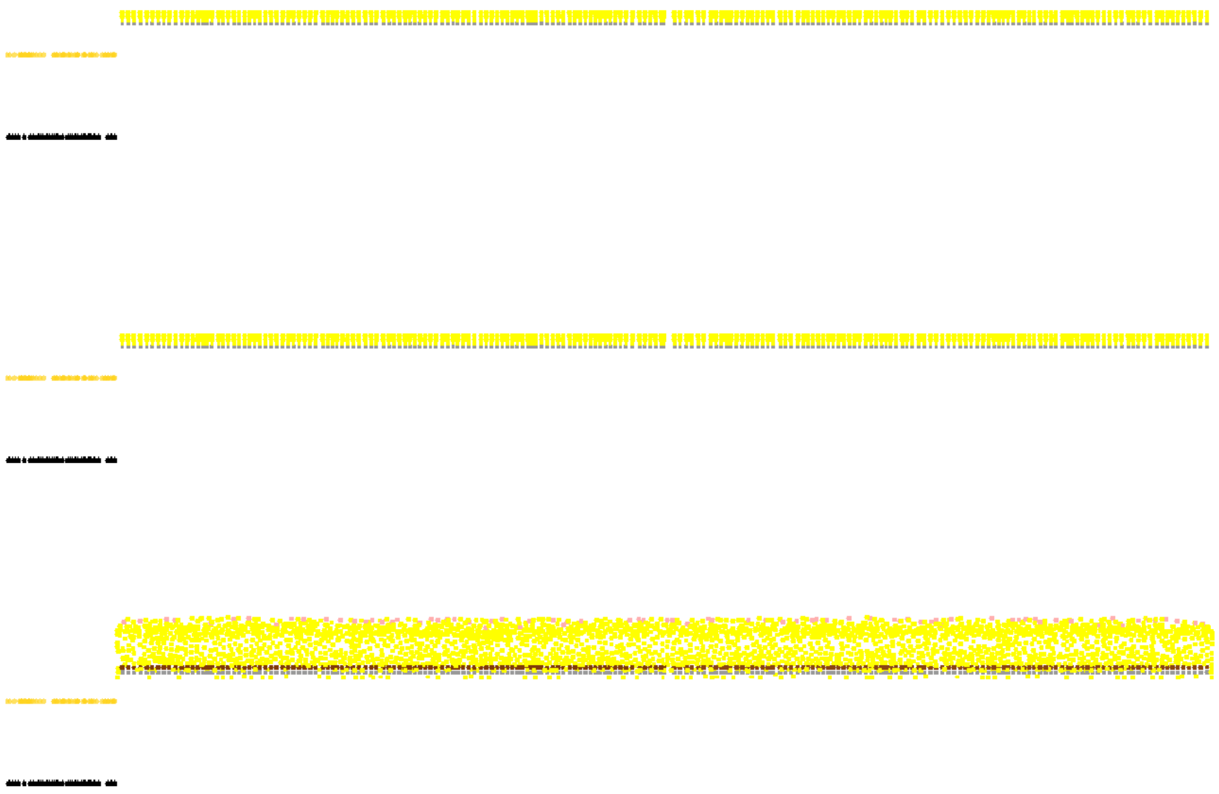


Figure 3.4. From top to bottom the development in the simulation of the preplate (yellow), from the neuroepithelial cells (gray) to the emergence of the future marginal zone cells (top yellow cells) and the upward movement of the future subplate cells (bottom yellow cells). The brown cells are the first progenitor cells for the cortical plate (bottom). The simulation sows here the formation of the preplate in different steps this is happening as a smooth transition in the actual simulation and has to be depicted here in 3 steps. The preplate structure will support later on the formation of the cortical plate in the middle of the marginal zone and the subplate. The gray cells already have radial fibers that the later born excitatory neurons can use to migrate tangentially into the cortex.

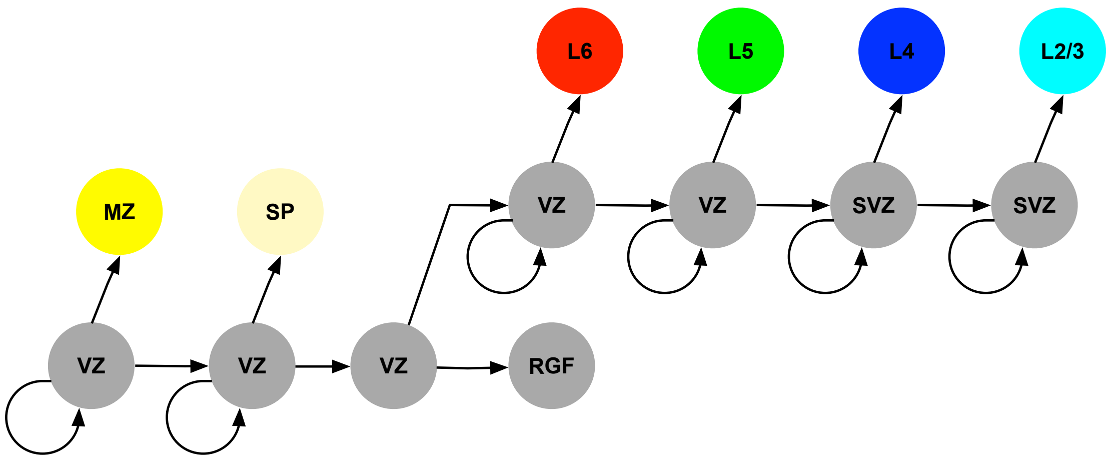


Figure 3.5. The mouse cell lineage of the ventricular zone. The different cell types emerging from the progenitor cells, to the last produced layer 2/3 cells. This is a pictogram of how the gene regulatory network plays out in the progenitor cells. The progenitor cells that are in the neuroepithelium at the beginning of the simulation produce essentially most of the cells in the cortical plate. This cartoon depicts the development and the sequential generation of the cortical cells over time.

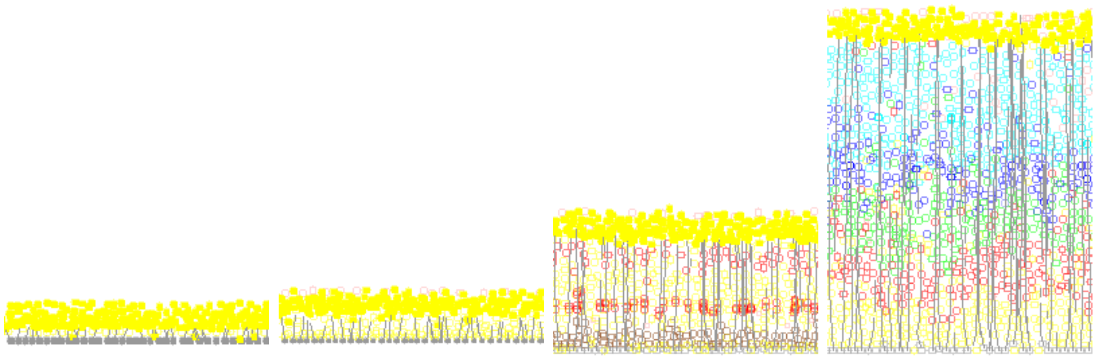


Figure 3.6. The growth of the radial glial fibers. From left to right the different stages of the development, from the neuron-epithelial cells to the fully grown cortical plate. We can clearly observe the inside out development of the cortical plate where each of the different colors stand for a different layer from layer 6 in red to layer 2/3 in cyan. The exitatory neurons of the cortex use the radial glial fibers as a guide for the radial migration to their final nesting destination.

3.2.4 Cortical Plate

During production of the preplate the radial glia cells continue to divide and so produce intermediate progenitor cells (IPCs) that contribute to the VZ pool and which will later produce the SVZ pool [152] [147]. The VZ IPCs will give rise to the various neuronal types in the deep layers of the cortex, while the SVZ gives rise to the more superficial types. In addition, the IPCs will, according to their location, give rise to two different profiles of neuronal types, which characterize cortical areas A3 and A6 respectively.

Whether a particular IPC will contribute to A3 or A6 depends on the state of its GRN at its cell birth. This state is determined by the local concentrations of the two long-range orientation gradients inherited from the embedding neural tube, which are initialized at the beginning of the simulation. If the concentration of the left gradient is larger than that of the right, then these cells will activate the gene regulatory network necessary for generating Area 3. If the right one is bigger they will activate the gene regulatory network for Area 6 [147].

The G-Machine that is activated at the birth of the progenitors senses the concentrations of the environmental gradients at its location. These cues enable the G-Machines to activate the appropriate GRN sub-network of each progenitor cell. Even though there is a rather sharp threshold in switching between the Area 3 and Area 6 sub-GRNs the emerging cortex exhibits a smoother transition between Area 3 and Area 6, as can be seen in figure 3.7.

3.2.5 Migrational Processes

Radial Migration

Asymmetrical divisions of intermediate precursors in the subventricular zone and ventricular zone generate neuronal precursors that must then migrate to take up their appropriate location in the cortical plate. The precursors do this by first executing a random migration until they encounter a radial glial fiber. They attach to the fiber, and then migrate upwards in the radial direction passing through the layers of previously migrated cells, until they encounter the MZ. Here the precursor neurons release from the fiber and nest into the accumulating layer of their peers.

The radial migration process is shown in figure 3.8. The model exhibits the expected inside-out lamination of the various neuronal types. That is to say the layer 6 pyramidal neurons are born before the layer 5 neurons, layer 5 before layer 4, and layer 4 before layer 2/3 pyamidal neurons[147] [142] [131] [51]. The cells, though, migrate and settle immediately into the cortex (shown as a pictogram in figure 3.9 and the simulation in figure 3.10) [152]. The layer 6 cells are the first to arrive at the MZ, and as they accumulate they begin to form the cortical plate, which splits the preplate into the overlying marginal zone, and underlying subplate. The

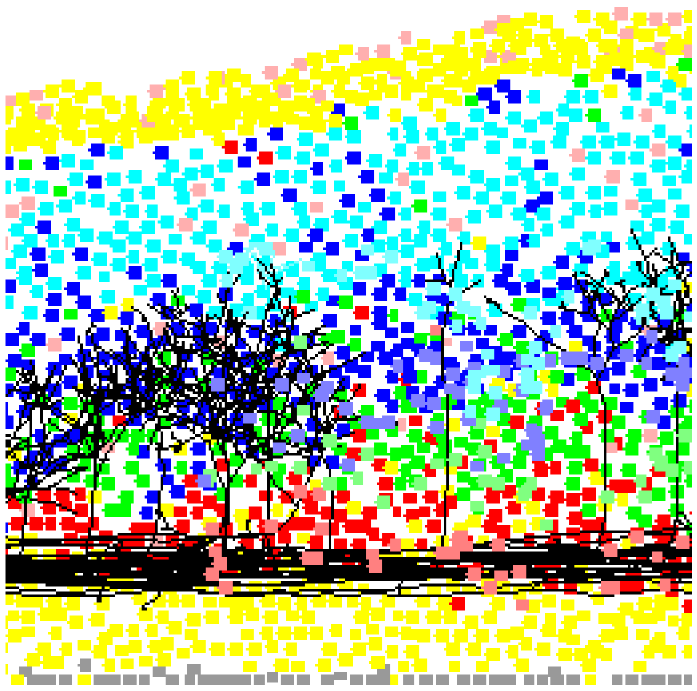


Figure 3.7. The boundary formation between Area 3 and Area 6.

successive neuronal types contribute to the continuing thickening of the cortical plate. In the video the process can be observed in motion: http://youtu.be/l6BOL4suYYg 'Cortical Plate development'.

The same type of radial migration machine is used in all neuronal precursors. However, the state of the precursor's GRN determines the type of neuron that it will differentiate into, and the neuronal type-specific G-Machines that will express the characteristic dendritic and axonal growth patterns are actived as the precursor beds into its layer.

The production of the cortical plate is governed by the dynamics of the GRNs of the intermediate progenitors of the VZ and SVZ [131] [142] [153]. Asymmetrical division of progenitors can lead to one or both daughters exiting the mitotic cycle and differentiating towards neuronal precursors. The various types of neuronal precursors will express a common family of G-Machines that provide for migration, nesting, neurite growth etc. However, the particular machines are slightly specialized according to neuronal type. It is these relatively sparse specializations on a common family of machines that cause individual precursors to exhibit type-specific behaviors and to generate type-specific neuronal connection patterns. The state of the GRN at the time a neural precursor is born determines its final neuronal type, and so determines the type-specificity of the G-Machines that it will subsequently instantiate during the course of its migration and differentiation. The type of the cell is declared by the secretion of a type-specfic membrane-bound marker.

When a neuronal precursor is born it instantiates a G-Machine for radial migration. This machine causes the neuron to wander randomly until it encounters a fiber of an RGC. The precursor then attaches itself to the fiber and migrates along the fiber in the direction of the marginal zone. During this migration the precursor is able to sense the cell types surrounding it due to their type-specific markers. When it encounters nest cells of its own type, it detaches from the RGC fiber and initiates nesting behavior. If it does not encounter any already nested peers, the precursor continues migrating along the fiber until it detects MZ cells [134] [118], which trigger detachment and nesting. In biology the precursors stretch vertically as they ascend the fiber. We model this stretching as a reduction in force field of the neuron, so thinning its horizontal expansion, which enables it to penetrate more easily the layers of cells that have already nested.

Nesting is the process whereby a neuronal precursor detaches from the RGC fiber [118] and transforms its morphology of bipolar into multipolar. In our simulations this transformation is modeled by the expansion the cell's size and force field. The action of nesting also activates the appropriate neurite G-Machines, specialized for generating the intra- and inter-laminar axonal projection patterns that characterize that neuronal type.

As in the case of the preplate development, it sometimes happens that some neuronal precursors become displaced and isolated from their peers. These cells detect their isolation and undergo apoptosis using the same kind of G-Machine deployed in the preplate cells.

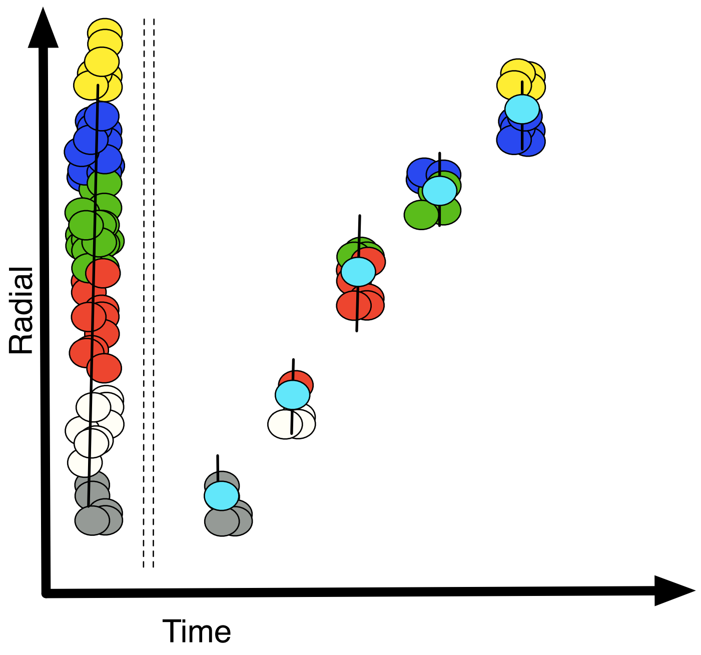


Figure 3.8. A cartoon of the radial migration of a single layer 2/3 pyramidal cell (cyan) and what the cell can sense over the course of its migration along the radial fiber (black). In white the subplate cells, in red the layer 6 cells, in green the layer 5 cells, in blue the layer 4 cells and in yellow the cells of the marginal zone

Tangential Migration

The inhibitory neurons of the murine cortex are derived from the medial and caudal ganglionic eminences of the basal telencephalon[18] [118] [129]. The inhibitory precursors migrate laterally from there into the pallium, where they take up their nesting positions in the developing cortical plate. In biology these precursors migrate by two routes. The major route is through the Intermediate Zone immediately beneath the developing cortical plate. The second, minor stream, is through the Marginal Zone[118] [134] [182] [168]. It is observed that the birth times of the precursors in the eminence correlates with their target layer in the cortex [18].

In the model, the progenitor cells of the eminence contain the same genome as their pallial

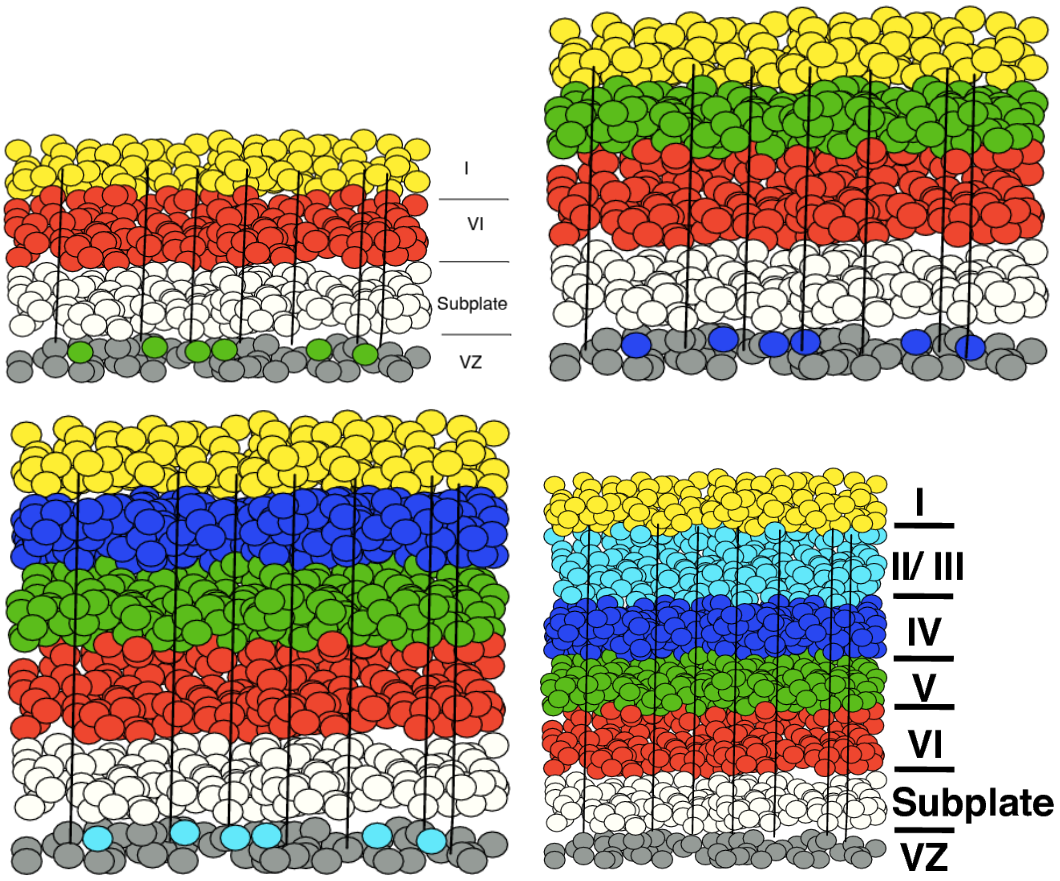


Figure 3.9. A cartoon of the generation of layers 6 to 2/3. Top left: the Layer 6 cells (red) and how they split the preplate into the subplate (white) and marginal zone (yellow). Top right: the nesting of layer 5 cells (green) in the cortical plate. Bottom left: the nesting layer 4 cells (blue) in the cortical plate. The bottom right the nesting layer 2/3 cells (cyan) in the cortical plate. Note the inside out development of layer 6 (red) then 5 (green) then 4 (blue) then 2/3 (cyan).

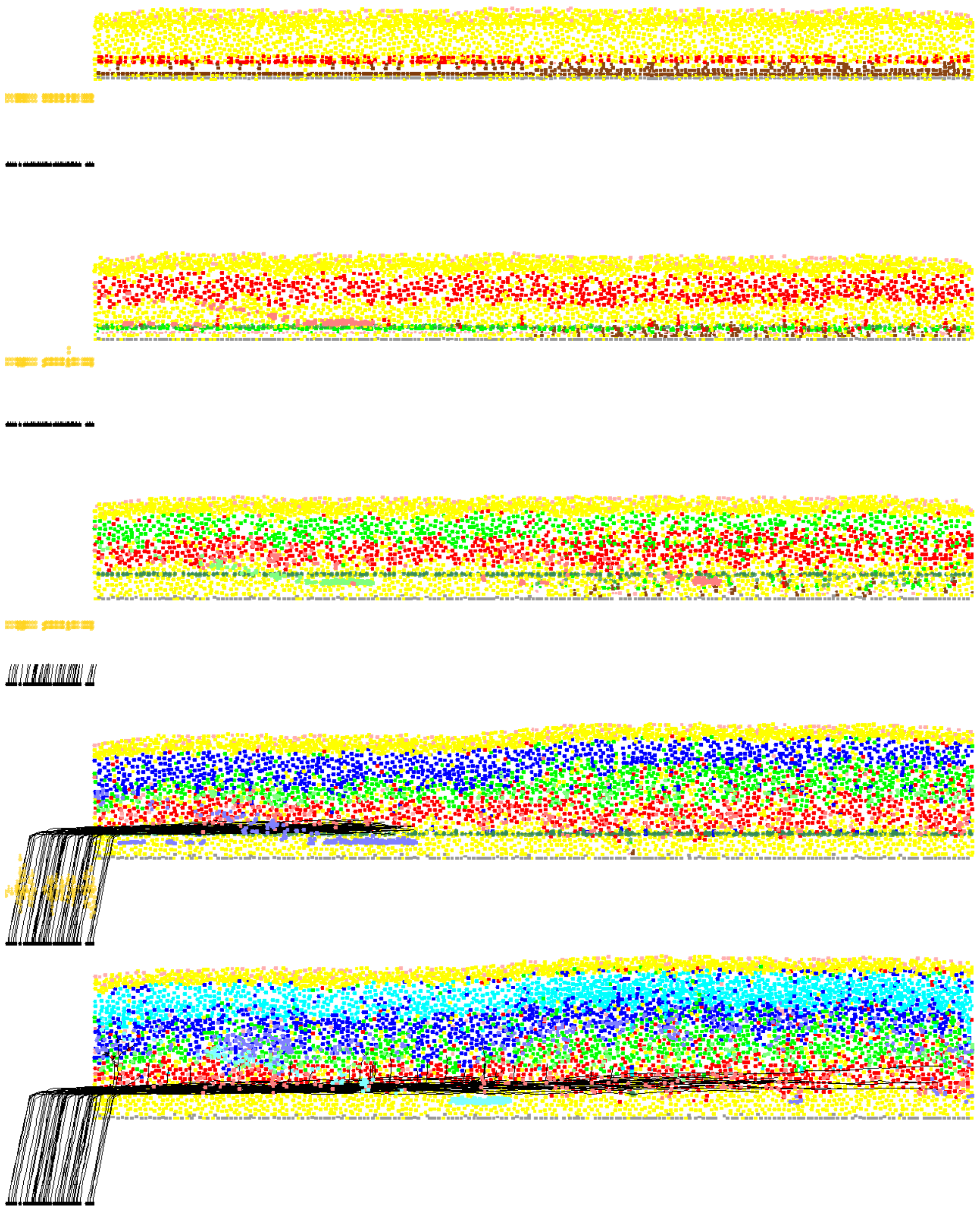


Figure 3.10. Top to bottom: (1) The emergence of layer 6 cells (red) migrating into the preplate, splitting it. (2) The generation of layer 5 cells (green) and the settling in the cortex. (3) The formation of the subventricular zone in the middle of the subplate in dark green. (4) The formation of layer 4 in dark blue and (5) the settling of the cyan layer 2/3 cells finishing the cortical plate formation. All of these effects are shown in the real simulation with other aspects that are discussed later in this chapter.

counterparts. This must be true, since both populations must inherit from a common embryonic progenitor. However, the model GRNs of the eminence progenitors are initialized to a slightly different state than those of the pallium. This means that the generation of inhibitory precursors in the eminence will follow the same principles as the generation of excitatory precursors in the pallium, but the specific types and behaviors of the eminence precursors will be different from those of the pallium. In particular, their migratory behavior will be different, and they will nest and differentiate with inhibitory characteristics. For convenience we consider only the major IZ migratory route in these simulations. Little is know from biology about the mechanism whereby the eminence precursors come to distribute themselves across the breadth and layers of the developing cortical plate. We propose a mechanism for this process.

In figure 3.11 we show the tangential migration of the interneurons. The tangential migration and the nesting of the interneurons in the simulation can be observed in the video at
http://youtu.be/RXYjvSscvow

We propose that the inhibitory precursors use the basic orientation gradients of the telencephalic wall (that are taken as initial conditions in the model) to guide their migration. The progenitors of the ganglionic eminence region are mapped onto the cortical plate by simple transformation of source concentration (in the ganglionic eminence region) to target concentration (in the cortical plate region). This mechanism establishes a simple addressing system that maps the birthplace of the interneuronal precursor in the eminence to its horizontal target location in the cortical plate. The location of the inhibitory precursor in the cortical lamination is determined by the temporal order of birth, as is the case for the generation of laminar types in the pallium [18]. This simplification follows from the fact that the same GRN is used in the progenitors of both the eminence and the pallium. Therefore the same basic cell lineage and G-Machine patterns are available in both germinal locations, but can be specialized for specific behaviors. For example, the migratory G-Machines of the pallium assert short-range random tangential migrations followed by radial migration; whereas the migratory G-Machines of the eminence assert long-range gradient following tangential migration, followed by radial migration.

The (tangential) migratory G-Machine of the eminence is instantiated by local promotion conditions, directly after the birth and initial differentiation of the inhibitory precursor. The G-Machine reads the local concentrations of the orientation gradients. These are used to determine its target concentrations in the cortical plate. We propose a mechanism whereby the cells read at their position of birth the concentration of an external gradient. This concentration is translated into a cell internal signal that is remembered by the cell. The strength of this internal signal is matched to a gradient in the cortical plate. The strength of the internal signal tells the cell at what concentration of the external chemical gradient it has reached its tangential goal. This mechanism need not be exact but can be noisy. A dispersion along the tangential axis

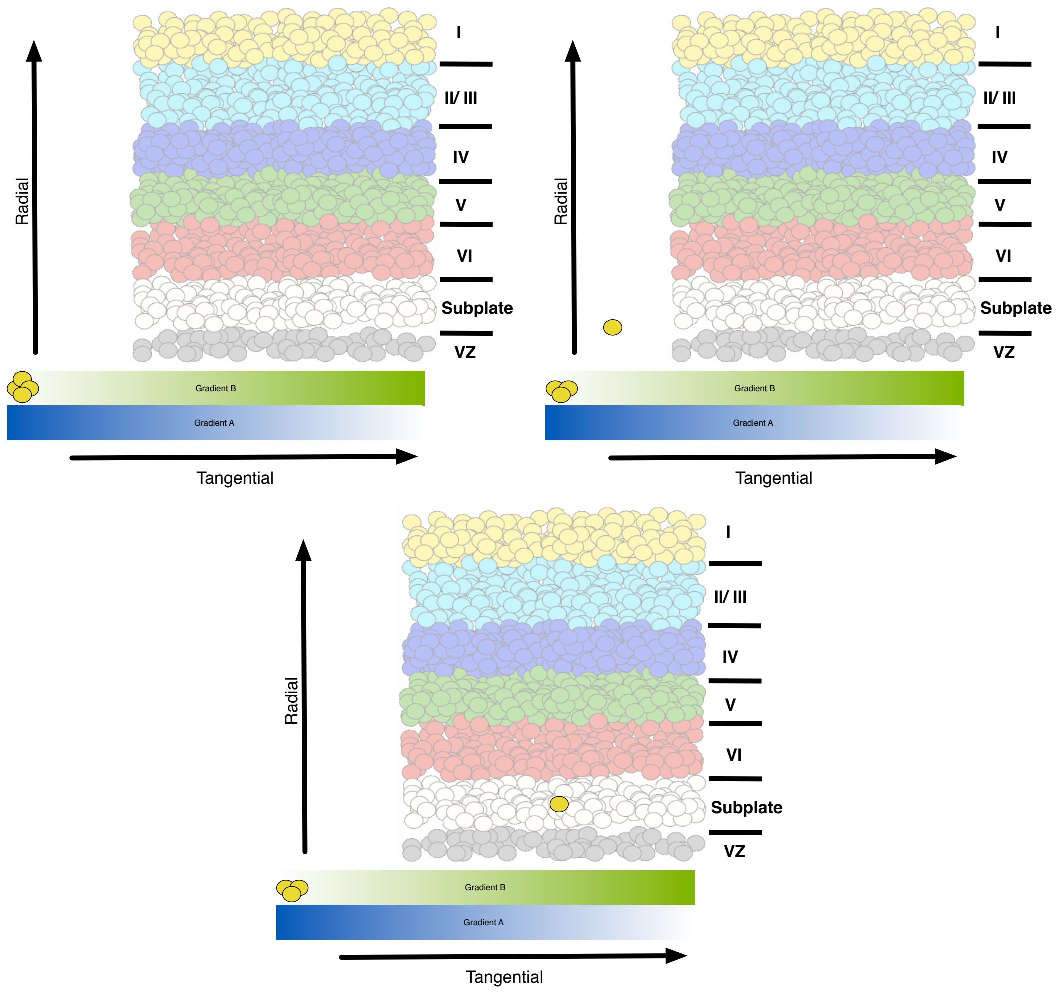


Figure 3.11. Pictogram of a tangentially migrating cell (yellow) being born from the ganglionic eminence at a distance from the cortical plate. Top left: the birth of the cell. The cell reads information about the position where it is born from the blue gradient, Gradient A. This will later be matched to the green gradient (Gradient B) in order to find the final tangential position of the cell. Top right: the migration up the green chemical gradient (Gradient B) through the subplate (white) to (bottom) its tangential position in the cortical plate.

in the cortex can still be achieved. The mechanism of reading the gradient and using it as an address is much like mechanisms that allows cells to decide on their cellular fate dependent on an external gradient [107]. The interneuron will begin to migrate horizontally as soon as it is born. It migrates through the intermediate zone below the cortical plate, following the orientation gradient, and continually compares the current concentration with that observed at its birth time. In our model, there is no attraction signal coming from above or below the intermediate zone that the tangentially migrating cells react to, therefore the cells will stay in the intermediate zone during tangential migration. When the two concentrations match, the interneuron has found its horizontal location and stops its tangential migration. The G-Machine now switches to another migration mode which is radial search for the target layer where it will settle. This target layer is determined by the state of the GRN at cell birth, which determines its type and destination in a similar manner to the developmental sequence of the cortical plate in the pallium.

The interneuron detects its destination layer by sensing which neurons surround it. If they are of correct type, the interneuron will nest with a certain probability. As in biology the radial migration of interneurons does not make use of the RGC fibers. It might well be that its destination layer is still under construction at the time that an interneuron begins its tangential journey. By the time they arrive at their horizontal target location, the layers will have formed and they can satisfy their radial nesting conditions.

Figure 3.12 shows the migration of the different interneurons into the cortex. In the case of the interneurons in layer 2/3, the tangential migration G-Machine will instantiate a machine that grows axons and dendrites. As yet, only the machinery for the outgrowth of layer 2/3 interneurons is modeled, but there is no inherent limitation to also modeling the interneurons of other layers. The exact outgrowth pattern is explained later in this chapter.

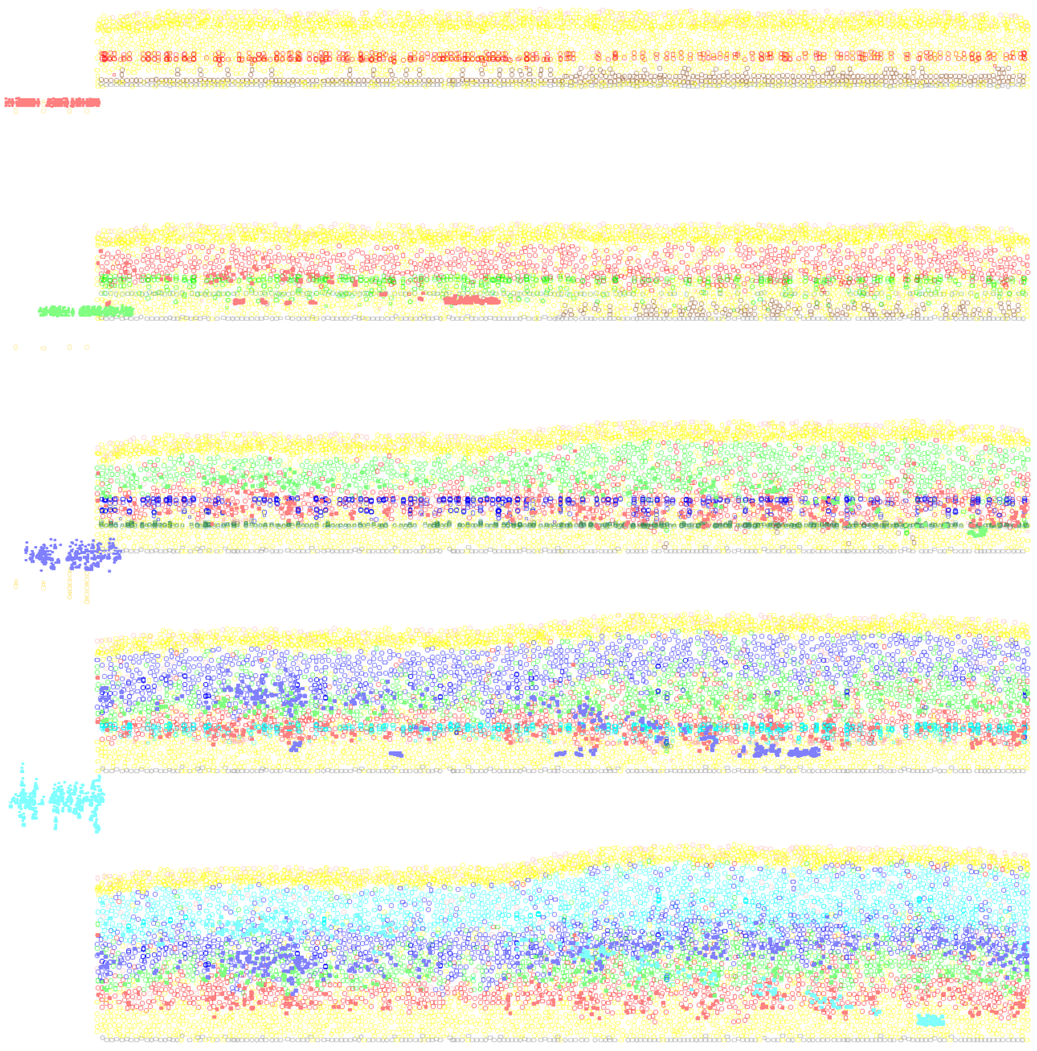


Figure 3.12. The simulation of the interneuron migration. From top to bottom, the birth of the different types of interneurons that will settle into the cortical plate after undergoing a tangential migration through the subplate. The cells follow a chemical gradient to the right, using the concentration of this chemical gradient as a source to calculate their final tangential destination in the cortical plate by matching the concentration at the birthplace with the concentration at the destination via a biologically plausible transformation function. In light red we see the birth of layer 6 interneurons in the ganglionic eminence, their migration below the cortical plate and the settling of the cells in layer 6. In light green layer 5, in light blue layer 4 and in light cyan the layer 2/3 interneurons. All are born at nearly the same time as their matching pyramidal cells and all settle radially in the appropriate layers.

3.2.6 Thalamocortical and Corticothalamic Ingrowth

Thalamus

The role of the thalamic cells in our model is only to provide thalalamocortical afferents. We do not consider the development of the thalamus per se, at this stage. The position of the thalamic cell population is set as an initial condition of the model. Each thalamic neuron is initialized with a growth cone G-Machine that grows first up in the direction of the cortex and then turns to follow the orientation gradient tangentially towards the right (laterally). The model deviates from what is observed in biology. There the thalamocortical (TC) axons route into the internal capsule in cooperation ('handshake') with cortico-fugal axons emanating from from the subplate [13] [130]. We do not model the internal capsule, nor the handshake mechanism as yet. However we still simulate the fasciculation of the cortico-fugal axons from layers 6 and 5 (see section 3.2.6) [127]. Moreover, the growth rules of the TC axons until they reach the model VZ are tailored for this simplified location of the representative thalamic population, and cannot conform to biology. However, once the model TC fibers enter the developing cortex, their behavior becomes biologically plausible. In the mouse cortex it is observed that fibers to grow up into the cortex distributed over the cortical plate, turning up radially into the cortex at different tangential locations [150] [127].

In our model the TC axons employ an addressing system that maps the thalamic origin to the cortical destination that is similar to that used by the ganglionic eminence interneurons. The first growth step of the TC axons uses the concentration of the linear orientation gradients to generate a mapping to the concentrations in the cortex. This mapping distributes the axons nicely in the horizontal plane of the developing cortex. The TC axons arrive at their horizontal destinations beneath the cortical plate at a time when the plate is still under development [130] [127] [153] [150]. The axons wait below the cortex at their destined horizontal position until they receive a signal from the developing plate that it is ready to be innervated. The signal the model uses is a membrane bound chemical that is expressed by the migrating pyramidal neurons of layer 2/3, this signaling mechanism ensures that all the destination layers for innervation are ready for the TC axons. On receiving the growth signal the axons turn upward in the direction of the superficial cortex. As the growth cone passes through layer 6 it will create sidebrances with a certain probability. These branches create axonal patches in layer 6. The trunk TC axon continues growing through layer 5 until it encounters layer 4, where it generates a much larger patch than its layer 6 branches do. The patches are currently only first approximations to the branching morphology observed in reconstructed neurons of the real mouse cortex. Further work is required to improve the growth cone model's branching behavior so that the model axonal arborizations conform to observed branching statistics. This is true for the model axonal branching patterns of all cell types: they have a strong resemblence to observed patterns, but their statistics need to be improved.

The growth of a representative thalamocortical axon is shown in the figure 3.13 as a cartoon and in figure 3.14. The ingrowth of thalamic axons can be followed in the video: http://youtu.be/vGBXP1RF 'Thalamic Cell'.

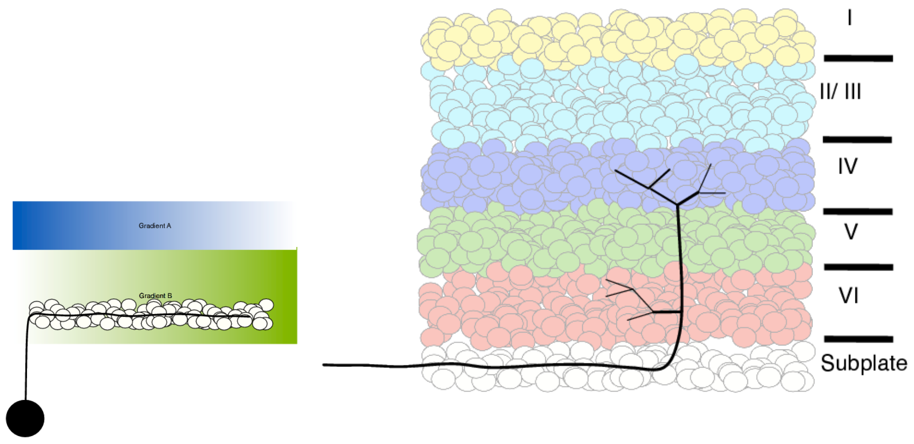


Figure 3.13. A cartoon of a thalamic axon (black) growing up to the subplate (left), then tangentially up the green gradient in the subplate (white), and (right) finally turning up into the cortical plate at the appropriate tangential location with arborization in layer 6 (red) and in layer 4 (blue)

The axonal G-Machines generate their behavior using the following rules. The G-Machines for tangential ingrowth are located in the tips of budding TC neurites. These G-Machines are activated at the very beginning of the simulation, however they lie dormant with an internal timer for a first phase of development. We are not concerned with the development of the thalamus in our model up to the point where it begins to send its axons in the direction of the cortical plate. Therefore we introduced this waiting period at the beginning of the simulation. This waiting period is a purely technical means to keep the beginning of the thalamic axonal growth in sync with the cortical development. During this phase the machine senses the orientation gradient and remembers its initial position in terms of the chemical concentration of the gradient at that location. When it begins its work, the growth cone machine will grow in the direction 'up' until it senses that it has reached the height to turn right. At this point the ingrowth G-Machine switches its behavior to follow the cortical orientation gradient through the subplate, beneath the developing cortical plate. The initially sensed concentration of the orientation gradient at its origin provides a reference against which to decide where in the subplate it should stop

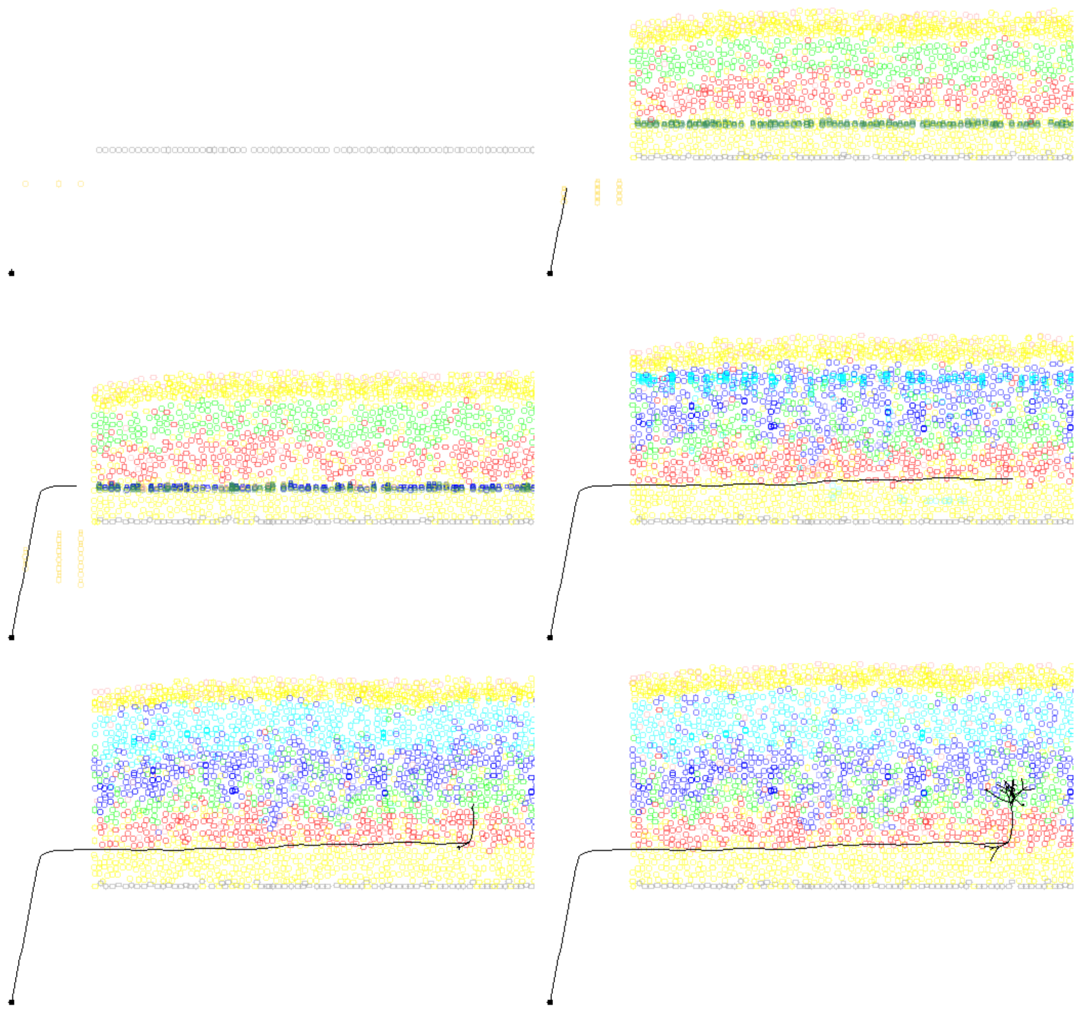


Figure 3.14. From top left to bottom right, the simulation of the outgrowth of a thalamic cell (black) through the subplate tangentially into the cortex. The growth cone settles in the subplate until layer 4 has emerged and it receives a signal from the layer 2/3 cells that migrate by. This signal makes the growth cone turn up into the cortical plate. On its way it produces a small arborization in layer 6 and a large and final arborization in layer 4. The tangential position where it should turn upwards is found by matching the concentration of the gradient at the staring place with the concentration of the gradient at the target through a matching function.

tangential migration and wait until it senses layer 2/3 cells migrating radially past it. Since the axon lies above, or in, the proliferating subventricular zone, it is sure to detect the radially ascending passage of migrating layer 2/3 precursors. Having received the signal, the G-Machine will steer the growth of the axon in the radial upward direction. It is still debated whether the murine TC growth-cones do in fact wait in the subplate as they do in primate development [51] [130] [127] [153]. We have implemented one plausible mechanism for waiting. We chose layer 2/3 to be the trigger condition for radial migration, but there are alternative triggers, for example the passage of layer 4 neurons. The G-Machine is rather generic.

The G-Machine senses when it is surrounded by layer 6 cells, and uses that cue to instantiate with a certain probability a side-branch machine in layer 6. This growth cone contains a further G-Machine that will, when triggered, generate small axonal patches. After generating the layer 6 offspring growth cone, the G-Machine(TC) continues its growth toward the marginal zone probing for layer 4 cells as it goes. Having found layer 4 cells it stops upward growth and instantiates the large-patch generating G-Machine.

The present G-Machine for generating small patches uses probabilities to generate its growth. It follows the Galton-Watson branching model [94] [24]. This model has three probabilities. Each time the model is executed it draws a random number and compares it to the three probabilities. One is the probability to stop, another is the probability to bifurcate and re-invoke the same G-Machine again, and the third is the probability to make a growth step. The direction of the growth is given by the outgrowth direction of the axon plus some randomness. In this way it is possible to generate a self-similar growth pattern giving the emerging axonal tree a fractal shape that resembles shapes observed in real axons. The G-Machine for creating big patches in layer 4 also uses the Galton-Watson model but with probabilities that produce larger trees.

These models are a simple abstraction of the axonal growth process, and do not properly reflect the underlying biological axonal physical growth mechanism. We are currently developing such models. Meanwhile the Galton-Watson approach provides a good first approximation to the observed outgrowth patterns.

Corticothalamic Projections

As soon as layer 6 pyramidal neurons have completed their migration and started to nest, a machine that is activated by the radial migration process begins the outgrowth of axons and dendrites. The G-Machine(CT) responsible for this growth is instantiated by the differentiation machine of the layer 6 pyramidal cell, that has in turn been activated by the radial migration G-Machine. The G-Machine will begin to grow down in the direction of the subplate. Initially the GM(CT) axon grows ‘downward’ into the underlying subplate in the direction of the white matter, en route to the thalamus. After entering the subplate these corticothalamic (CT) axons stop and idle, waiting for the arrival in the subplate of the thalamo-cortical afferents. When TC and CT axons meet, they fasciculate (zip) together, and the CT outgrow continues. The CT axons use their fasiculation with the TC axons to guide their growth toward their thalamic target, as is proposed to be the case in biology [130] [51] [148] [127]. GM(CT) growth stops when it reaches the TC axonal initial segment, next to the soma of the TC axon. Because of the limitation on computational resources in our simulations, we arrange that CT fibers arise probablistically from only a sample of layer 6 cells. This reduces computational load, and simplifies the visualization of the axonal growth processes. We have confirmed separately for smaller cortical simulations that complete axonal growth (from all cells) does not affect the integrity of the simulation.

CT outgrowth is shown schematically in figure 3.15, and the simulation of the process is shown in figure 3.16. Fasciculation can be observed in the following video:
http://youtu.be/5riTXo7SjTY 'Fasciculation of Corticothalamic Axons'.

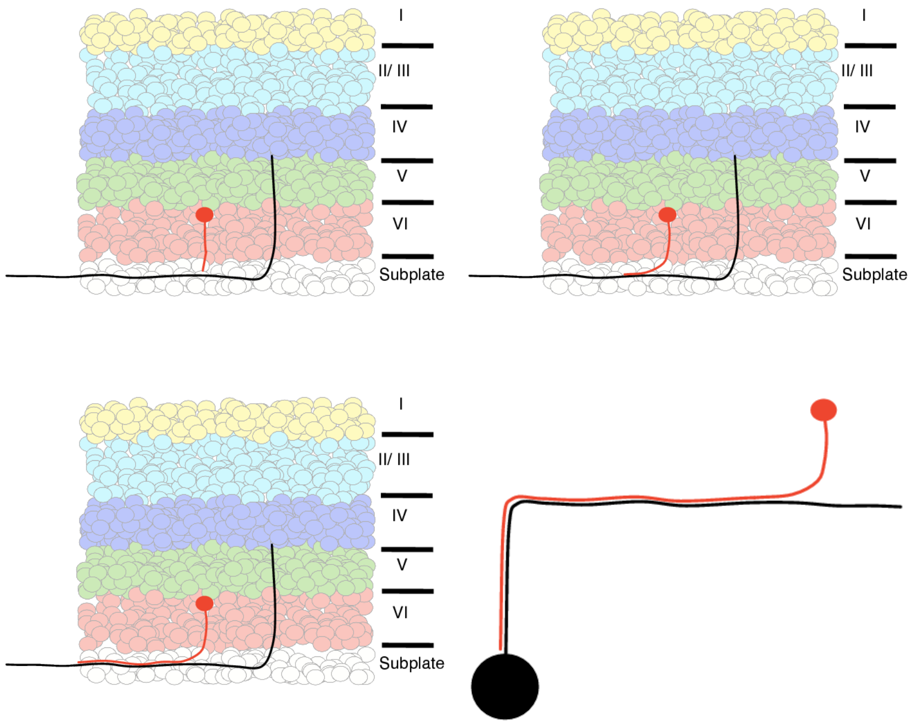


Figure 3.15. Top right to bottom left: The outgrowth of a layer 6 pyramidal axon (red) growing down in the direction of the subplate until it finds a fiber originating from the thalamus (black). This allows the axon to fasiculate to the fiber from the thalamic cell and follow it back until it reaches the origin of the fiber, the thalamus.

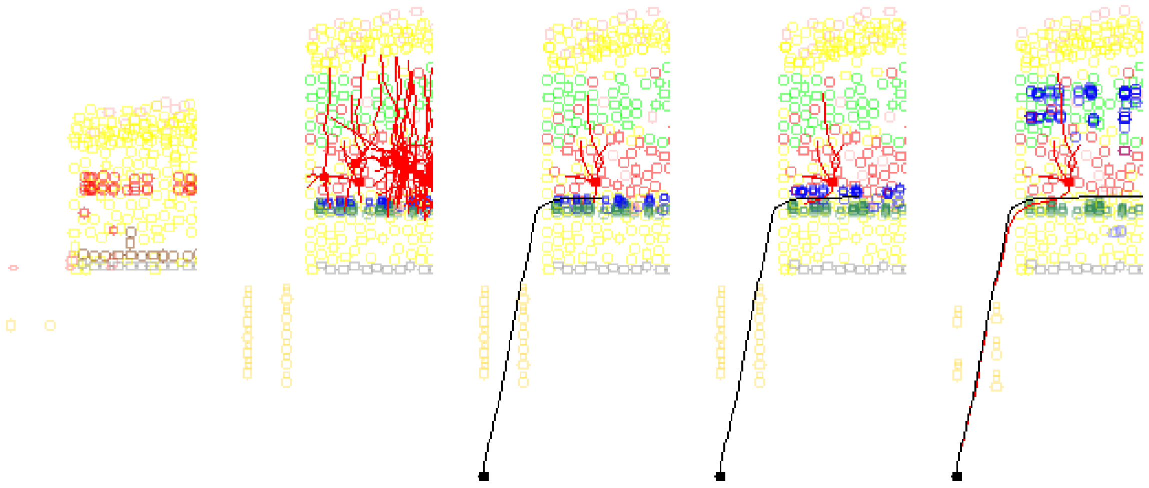


Figure 3.16. From left to right: The outgrowth of layer 6 axons (red) in the the subplate (hollow yellow cells). The axon waits in the subplate for the arrival of a thalamic fiber (black) from the thalamus and then fasiculates to it. Here one example is highlighted. The display of the other layer 6 pyramidal cells is suppressed after the fasciculation of the first one. The layer 6 pyramidal axon follows the thalamic fiber back to its origin while the thalamic fiber is still growing into the cortex.

3.2.7 Neurite Outgrowth and Connection Scheme

The generation of cortico-cortical connections follows similar principles to those of the CT connections. When neuronal precursor cells arrive at their target cortical plate and nest, the G-Machine responsible for the radial migration instantiates a type-specific G-Machine that guides differentiation of the precursor toward its final phenotype. One of the actions of the GM(diff) is to instantiate the machines that cause dendritic and axonal outgrowth. We base our models on the outgrowth patterns reconstructed and analysed by Binzegger et al. [23].

Although neurons contain type specific machines for constructing their arbors, these machines all employ the same principles. Thus, a small family of axonal G-Machines, with slightly different parameter settings create a variety of different types of morphologies. A primary axonal G-Machine extends the trunk axon toward the primary target of the source neuron. In the pyramidal neuron case these are usually the long range axonal arbors that provide inter-areal or subcortical projections through the white matter. The trunk usually creates side branches using a GM(Sidebranch) that steers the branches toward interesting target regions, where the side branches might bifurcate to increase coverage. Finally a GM(patch) is inserted that generates a patchy arborization by recursive branching. We expect that the axonal connection motifs observed in biology arise because of the composition of just a few growth mechanisms. Indeed, the model axonal motifs are in good qualitative agreement with biologically observed axonal morphology. In future work the growth cone model will be improved to obtain a better correspondence with the topology and branch statistics of biological axons.

Representative axonal outgrowth patterns can be observed in 3D in the following video: http://youtu.be/pBqZ8SouWdQ ‘Development and 3D Rotation of Selected Neurons’.

In this section we will only show the machine descriptions for the machines of the layer 6 pyramidal cell. All the other pyramidal cells modelled and the interneuron are described in appendix C.

Basal dendrite

Amongst its actions GM(differentiation) instantiates a basal dendrite outgrowth machine, which causes the soma to grow three primary dendrites in random radial (with respect to the soma) directions. Each of the three growth cones contains the same G-Machine(BasalDendrite). We have randomly chosen three initial dendritic branches; the basal dendrite only resembles a real basal dendrite and is not based on biological data.

The Basal Dendrite machine performs the following operations:

BasalDendrite:

- Grow the current dendrite in the current direction.
- Reduce the diameter.
- Check if the diameter of the dendrite is not too small and check probabilistically whether to bifurcate
  - Yes: bifurcate and create two new growth cones containing instances of G-Machine GM(BasalDendrite)
- Check if the diameter is too small
  - Yes: stop growing.

This machine results in a radial outgrowth pattern of basal dendrites from the soma.

Layer 6 pyramidal cells

Besides the basal dendrite, the GM(diff) of layer 6 pyramidal neurons will start an apical dendrite machine GM(P6ApicalDendriteMain) in the direction of the marginal zone.

P6ApicalDendriteMain:

- Grow the current dendrite in the direction of the marginal zone.
- Check if it has reached the marginal zone.
  - Yes: wait for a while. (Layers 2/3 and 4 might not yet be established.)
  - Check whether the marginal zone has in the meantime been pushed upward
    - Yes: Grow further up.
- Check if in layer 2/3
  - Yes: stop growing.
- Check if in layer 4 and check probabilistically whether to bifurcate
  - Yes: bifurcate and create two new growth cones containing instances of G-Machine GM(P6ApicalDendriteSide)

P6ApicalDendriteSide;

- Grow the current dendrite in the direction of the marginal zone.

- Check if it has reached the marginal zone.
  - Yes: wait for a while. (Layers 2/3 and 4 might not yet be established.)
  - Check whether the marginal zone has in the meantime been pushed upward
    - Yes: Grow further up.
- Check if in layer 2/3
  - Yes: stop growing.
- Reduce diameter
- Check if the diameter of the dendrite is not too small and check probabilistically whether to bifurcate
  - Yes: bifurcate and create two new growth cones containing instances of G-Machine GM(P6ApicalDendriteSide)

With these two machines nice bouquet apical dendrites are produced. The ceasing of growth is necessary because the neurons of layer 6 begin to grow out immediately even before layers 5, 4 and 2/3 are established and while layer 6 still has a border to the marginal zone. But the axons should not grow into the marginal zone and therefore wait for the establishment of the necessary layers. Layer 2/3 is not permissive for the growth of layer 6 pyramidal cell apical dendrites as we observe no innervation of this layer in the mouse cortex[23].

The growth of a layer 6 cell is shown as a cartoon in figure 3.17 and the simulation is shown in figure 3.18. The development of a layer 6 pyramidal cell can be observed in the following video: http://youtu.be/iGa5F6gYiZ8 'Development of a Layer 6 Pyramidal Neuron'.

The differentiation machinery of layer 6 will also start an axon that grows down into the white matter to the thalamus G-Machine(PyramidalAxonMainL6). This is described in the sections on corticothalamic projections.

PyramidalAxonMainL6:

- Grow in the direction of the subplate.
- Check if in layer 6 and check probabilistically whether to sidebranch
  - Yes: Grow out two to three side branches with G-Machines GM(P6Side1Outgrowth)
- Check if the growth cone is in reach of a thalamic fiber.
  - Yes: fasiculate to it and change the growth mode to fasiculation growth.

  - Grow along the thalamic fiber
  - Check if the growth has reached the thalamus.
    - Yes: stop growing.

P6Side1:

- Grow in the direction of the marginal zone.
- Check if it has reached the marginal zone.
  - Yes: wait for a while. (Layers 5, 4 and 2/3 might not yet be established.)
  - Check whether the marginal zone has in the meantime been pushed upward
    - Yes: Grow further up.
- Check if in layer 2/3
  - Yes: stop growing.
- Check if in layer 4
  - Yes: stop the machine and instantiate a new G-Machine(BranchLayer4) in the same growth cone.

BranchLayer4:

- Grow in the outgrowth direction.
- Check probabilistically whether to stop
  - Yes: stop growing.
- Check if in layer 2/3, 5, 6 or marginal zone
  - Yes: stop growing.
- Check probabilistically whether to bifurcate
  - Yes: bifurcate and create two new growth cones containing instances of the G-Machine(BranchLayer4)

With these G-Machines the axons side branch in layer 6, grow up to layer 4, and create a patch in layer 4. (See figures 3.17 and 3.18.)

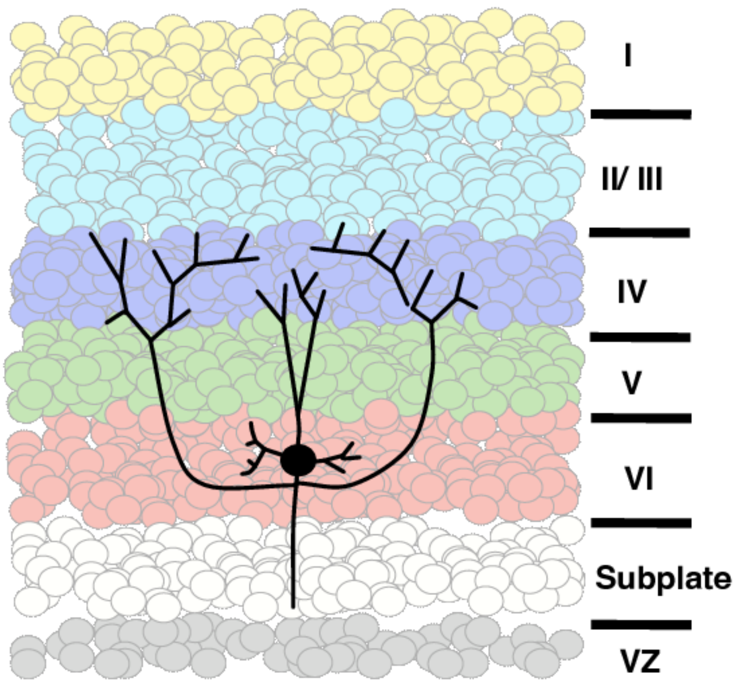


Figure 3.17. The connection pattern of a layer 6 pyramidal cell with its axons growing down and creating offspring side branches in layer 6 that grow up to layer 4, the apical dendrite that innervates layer 4, and the basal dendrite that stays in layer 6.

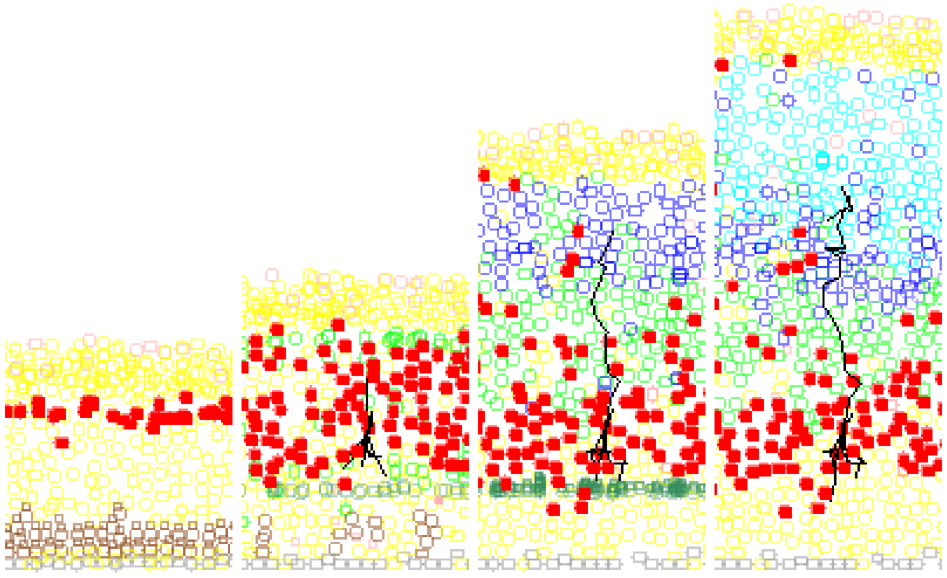


Figure 3.18. From left to right: The simulation of the growth of a layer 6 pyramidal cell through different stages of the development with its apical dendrite going up and the axon going down. The basal dendrite forms a local arborization pattern. Parts of the dendrites wait for the appropriate layers to appear in the cortical plate in order to fulfill their destined growth patterns.

3.3 Discussion

The mammalian brain is one of the most complex structures known to man, and the neocortex is crucial for the intelligent behavior exhibited by higher vertebrates. In the case of humans the cortex consists of billions of cells. It is arguably the most complicated and effective computational device on Earth. Although very complex in structure, the cortex essentially constructs itself from relatively few precursor cells in the dorsal telencephalon. This construction process is entirely distributed; there is no global control instance that dictates to each cell what it must do. Although all cells have a copy of the entire genome of the organism, how each cell makes use of that genome depends completely on its developmental lineage and the local environment it interacts with. The cells perform their tasks autonomously, and in doing so they collectively and collaboratively construct the brain. Our model starts out in a nearly unlabeled space. We introduce three sources of cells and two gradients in the tangential plane. We did this because we wanted to simulate cortical growth and wanted to start at the point in time when the first preplate cells emerge at E10 [171]. We show though that the labeling in the radial plane of space is unfolded entirely autonomously by the developmental process of the model. Any orientation needed for fibers and cells is self-constructed by the cells. The sources of the approximately four thousand cells that emerge are only three basic precursor types. Out of these progenitors more than two hundred thousand cells are created, which can be divided into twenty different cell types. Each of these cells is completely autonomous and has to act out its fate in the environment.

Most experimental studies of development focus only on partial aspects of brain development. Cx3D simulations offer a complementary method, in which simulations of the physical development of large regions of brain (here, neocortex) can be performed. To do so, we needed to integrate findings about gene regulation for the generation of different cell types, radial and tangential migration, the role of the radial glial fibers, the expansion of the cortical plate, the layering of the cortical structure, the integration of interneurons into the cortex and the connections of corticothalamic and thalamocortical fibers all together into a model that acts as one to construct the cortex. While assembling all these partial experimental data into one model it became apparent that not all aspects of corticogenesis have been thought of. Our model had to include assumptions in order to bridge gaps in the knowledge of cortical development.

One of the most interesting assumptions is the mechanism of tangential dispersal of interneurons during migration. It is not known how the cells distribute themselves in the tangential plane in the cortex. We propose an addressing-mechanism that relies on the reading of a local gradient at birth. When the cell is born in the ganglionic eminence it reads a local gradient, translates this gradient into a cell internal value and remembers it throughout the migration process. On arrival in the cortex, the value is matched to a gradient in the cortex, and the cell places itself where these values match. Through this mechanism a biologically plausible mechanism for the tangential distribution of cells is achieved. We propose that it could be checked in experimental studies whether cells from the same region in the ganglionic eminence end up tangentially in the same region of cortex. It is important to check whether such a gradient behavior could even be possible and if cell-internal signals could be found that change with the intensity of this (yet to be found) gradient. If this assumption were not in our model, the interneurons would have no means to figure out where to place themselves tangentially in the cortex and when to switch from tangential migration to radial migration.

Another unconfirmed assumption in the model is that the trigger for the cortical ingrowth of thalamic fibers is the generation of layer 4 cells in the subventricular zone. This has been hinted at before by Miller et al. [127] but never confirmed. If this were not in our model, the fibers from the thalamus would arrive in the cortex much earlier, even before the cortical layers were present, and the axons would turn up immediately into the cortical structure and grow directly into the marginal zone without connecting to layer 4.

Finally we assumed a mechanism quite similar to lateral inhibition that is present in each cortical neuron in case of misplacement in the cortex. These neurons would undergo a process of programmed cell death or apoptosis if they are misplaced in the cortex. It is known that several cells do undergo apoptosis in the cortex [79] but it is not yet certain what the triggering mechanism for this is. In our model we assume that a cell in contact with a majority of cells of a different cell type will trigger apoptosis. This mechanism was actually necessary in order to achieve clearly layered structures, otherwise many disturbances in the layering would have been present in our model.

One of the most subtle but most important assumptions is that of the structure of our model. The model has as its basis a gene regulatory network (GRN) that produces different types of cells at the right time. The GRN also ensures that the generation of cells stops at some point and that no more new cells are produced. If this did not happen, the consequence would be uncontrolled growth that would destroy the whole structure. In order for the gene regulatory mechanism to work and produce different types of cells, two principle mechanisms play an important role, symmetric and asymmetric division. Without these no diversity of cells would be possible. We will discuss this in chapter 4 in more detail. Only having the GRN causing the production of cells would create a mass of cells that are not organized at all so it is crucial to have the right type of cell being born at the right time. That is why the cells need to be able to express behavior and for that reason the gene regulatory network activates behavioral G-machines in each of the different cell types. All of these different G-machines are very simple sequential biological programs, that each on its own does not show any complex or interesting behavior. Only in association do they start to show interesting emergent behavior and build distinct cortical structures.

Our model is constrained through Cx3Dp (see chapter 6) and G-Code (see appendix A and chapter 2.2.4). Cx3Dp imposes on our model the constraints that cells are physical entities that occupy space and that two cells cannot occupy the same space at once. G-Code on the other hand implies that the model can only use behavioral primitives that are biologically plausible. Cx3Dp and G-code set bounds on how cells can be placed in space and how they can behave. Additionally to these physical and biological constraints, the documented behavior of development as described in chapter 2 poses constraints on our model of how cells organize themselves during corticogenesis. For many cell types specific behaviors have been observed that the model incorporates. The literature and the collected data on cortical development is vast and the model follows the most common opinions in the literature. All of the behaviors documented in the literature put constraints on our model that it has to fulfill in order to be biologically plausible. Even though the constrains were not all known at the time that the G-code language was created, G-code is powerful enough to allow for the modeling of all these aspects without exception.

Looking at the biology of the processes that the model implements from a computer science point of view there are a few interesting aspects. For example the construct of the cell as an entity. The processes of DNA replication and reproduction could work without the containment of a cell. The introduction of the containment achieved via the membrane brings distinct advantages; it allows for a very controlled cell internal environment that cannot be disturbed easily from the outside. It allows the transportation of all the necessary chemicals at once without having to rely on the fact that all chemicals are spread around through the entire space. Through the membrane a cell has very localized control over its cell-internal microenvironment and can therefore elicit very specific control over the parts of the system. The cell can at different times express specific receptors in the membrane for specific signals in its local (exterior) environment. Without a membrane this would all not be possible. It creates an encapsulation of functionality that is not accessible from outside, and only offers distinct interaction protocols with the outside world. Much like a well-designed computer program.

The communication between cells is especially interesting. It is very sparse in our model. A very complex communication scheme between cells was never needed and simple signals were enough. The cells express receptors that act through simple chemical messages with present / not present responses. These signals then trigger cascades inside the cell that can be rather complex in their chemical nature (which we model more abstractly with G-machines). A sign of a well designed computer program is that it has very selective and well defined interaction points between the different components of the system. This decouples the parts of the system and makes it easier to extend later on. This enables modularity where different additional parts of the system can then later be more easily integrated. For example we would imagine that when a cortical structure first emerged in evolution, the tangential innervating interneurons would evolutionarily come later in time and could be easily integrated because the tangential migration could rely on selective markers of space in the cortical structure and would not disturb the original communication protocols for generating the layering of the cortex.

Another interesting point is that during the cortical construction the cells actually build structures that serve other cells as a scaffold for building even more complex structures. Like the preplate generating orientation for the emergence of the cortical plate, the radial glial fibers that give orientation to the pyramidal cells migrating to the cortical plate, the marginal zone that acts as a stop signal for the radially migrating cells and the cortical plate with only pyramidal neurons acting as a scaffold for the tangentially incoming interneurons. This process of building scaffold after scaffold on top of each other is also known from design principles in software engineering, for a continuously improving product.

Our model shows very complicated emergent behavior. How is it possible to control this behavior and for a modeler to create a self-constructing system that shows this complexity without globally controlling the whole process from a central point without getting lost in the complexity? We credit this to the abstraction we used for our model, G-code. G-code uses as a central abstraction, the G-machine. Each G-machine is supposed to model one simple behavior of one cell. A cell can have many behaviors (G-machines) that are running in parallel to one another or in sequence. This abstraction allows the modeler to neatly separate the problem into sub-problems where he does not have to consider the whole simulation at once. Being able to define well separated sub-problems is a very well known and often used engineering technique such as it exists in object-oriented or functional programming. In object-oriented programming the programmer has to think in terms of what the responsibilities of the object are and how they can encapsulate this as much as possible and only offer very specific interactions with the outside objects with clearly defined interaction points. In G-Code, the modeler has to take the perspective of a single cell, a single cell behavior and ask what the interaction points are with the surrounding cells and the environment and how the G-machine should make the cell behave during the time it is active. This gives a very focused view where each G-machine only has to be looked at separately and only very sparse interaction points between G-machines have to be defined. Out of this follows behavior as we can see in the cortical development simulation that we present here. This separation into different G-Machines follows our understanding of how biological development is able to create structures like the cortex (we discuss this further in chapter 4). G-Code itself was built bottom up, which means that all the primitives that the G-machines are able to use are all biologically explainable and follow the rules imposed by biology. On the other hand, the modeler that uses G-Code can begin to design the problem top-down. In our case we went from the observation of development of the whole cortex to the smaller problem sets and eventually separate cellular behaviors. These behaviors could then be implemented using the primitives offered without the need to worry whether this is plausible because G-Code ensures this. This mixture of bottom up from G-code and top down design for modeling allowed us to make more complex behaving systems that show very complex behavior, without the danger that the modeler gets lost in the biological details of how this process has to be implemented with actual proteins. Van Ooyen [190] observes that there are these two types of approach for modeling and we combine both in order to achieve better results. With G-code the right abstraction for self-constructing developmental models has been found, we discuss this further in chapter 5.

Of course the molecular mechanisms behind each of the behaviors that we present as G-machines are very important and themselves extremely complex. For studying the construction process of the cortex looking at each single mechanism in detail would have led us to be lost in detail. Now that the model is established each G-Machine could be taken and redefined to be explained by specific molecular mechanisms. We are aware that it can be that our G-Machines will not always represent the exact mechanisms that nature chose to implement the observed behavior, but it is clear that such a mechanism must exist in order for development to work.

We are not the first to try to model development, many interesting approaches have been taken; Zubler's investigations [212] have shown that: For example simple organisms [120] [160], the growth of organs such as blood vessels [123], pancreas [169] or mouse limb buds [116] were modeled. In cortical development as nicely summarized by van Ooyen [190] there are many models ranging from neural tube formation [38] and regionalization of the neural tube [101] [117], to proliferation [162] [172] and migration [32]. Other models include very specific mechanisms such as delta-notch signaling [43] to more general models of arealization [70] and even cortical construction [211] created by Zubler et al. in our lab. It is also interesting that especially the growth of dendritic trees [192] [193] [56] [49] [177], axonal trees [186] [164] [185] and growth cone behavior [163] [205] have received a lot of attention, often based on ideas dating back to the Lindenmayer Systems [111]. Also other aspects of development have been looked at such as fasciculation [83] and synapse formation [191] (neatly summarized in Zubler's thesis [209]).

Where our model excels compared to others is that our model acts in a minimally labeled space, the cells that grow are completely autonomous, and every single cell in our simulation is born on the fly. All of the above models only focus on a very narrow topic and do not aim at reproducing the self-construction (except for [211]) of the whole cortical structure. We try to integrate all known and necessary processes from the early preplate production, the innervation of thalamic axons, the integration of the interneurons in the cortical plate and the axonal connectivity between the different cortical layers. Often the models do not integrate the environment as an active part for construction of the modeled process and we assume this to be crucial in our model so that the cells actively create an environment that is then exploited by other generations of cells in order to achieve self-construction. Our model does not purely rely on statistical measurements but also incorporates observations of very deterministic events (such as radial migration), although the GRN is inferred from statistical data in order to accurately reproduce the numbers of cells in different layers over time. (For more information on this see Pfister's thesis [146].) Our model is an algorithmic model that works with abstractions of behaviors rather than being purely mathematical as we find in many cases. Even though we expect behind each of the G-machines well defined interactions between proteins that could be described in mathematical terms. We chose an algorithmic approach because it allowed us to more easily compartmentalize the problem and to have a more descriptive form of implementation that enhances the understanding of people not as versed in mathematics. Moreover it allowed us also to separate the problem computationally into chunks of easily calculable parts that can be run in a parallel framework such as Cx3Dp (see chapter 6). With other methods this might not have been possible or might have been computationally even more expensive.

Looking at the model described one might wonder what the difference is in the simulation of cortical development as we approach it and the simulation of other complex systems such as weather simulations or simulations of self-organizing systems such as flocks of birds or other swarms [36]. In all three cases the simulation is compartmentalized; in the cortical simulation the cells and their neurite elements are the compartments, in the weather simulation the atmosphere is compartmentalized and in the case of the simulations of swarms and flocks the individuals are the compartments to be simulated. In all three cases these compartments need to communicate with each other, and all of the compartments contain data that represent their current state. Importantly, in all cases there is an emergent behavior of the overall system that is hardly predictable by looking at the individual compartments. The biggest difference is that in the case of our model, the computational compartments do not only contain different data but actually also contain different behavioral models. That is not the case in the weather simulation, the simulation substrate there is uniform. Also in the swarm or flock case the individuals are all behaving the same. In the case of cortical development there are many different cell types that behave completely differently and hold completely different data, but still work together to generate one structure, in our case areas 3 and 6 of the mouse cortex. This orchestration is possible is due to the encapsulation that G-code achieves, the plurality is possible through the symmetrical and asymmetrical divisions and the very specific and selective communication between the different cells. This will be discussed in more depth in chapter 5.

As we pointed out, our simulation is not complete and the model is a first step towards an overall cortical development model. There are many questions that are still unanswered, for example how to construct axonal arbors that are realistic. We have not found an explanation or a model in the literature that would explain the generation of cortical neurons such that they appear to be realistic. Nonetheless we were able to generate a general model that allowed us with very few G-Machines to construct axonal arbors that connect to the right layers and arborize there.

Other interesting projects that could be spun out of this idea of cortical construction would be to implement a model starting from one single cell, creating the neural tube and eventually ending up with the final cortex. Or to see how much of the model needs to be changed in order to achieve other mammalian cortical construction such as that of monkey. Additionally it is currently being investigated with G-code how whole cortical circuits could be established that exhibit actual learning [19]. Also the growth of other organs might be an interesting target to try and eventually whole organisms. The interesting aspects of models of the kind we present are that through its complexity new theories about developmental processes can be tested in the model settings before needing to perform very expensive and time consuming biological experiments. A screening of potential mechanisms can be done directly on the model and theories can be tested (these need of course then to be confirmed later in actual biological experiments). The model might have a predictive nature but can never replace the biological confirmation.

We have studied development with the example of mouse corticogenesis. We have created a convincing simulation that shows how cells can assemble themselves into a complex structure such as the cortex using only local rules encoded in their genome. We have learned how these cells have to behave in order to cooperate to self-construct a concrete example. But what have we learned in general about the process of biological self-construction? What are the principle elements biology has at its disposal to control a developmental process? Each of our G-machines only contains a few lines of code that are sufficient to control the cells such that they show global emergent behavior without being globally controlled. How can these simple rules generate such a complex global emerging behavior? How can all of this be compressed into the initial cells and create such a large structure in the end? And if these principles are clear, could we derive engineering techniques from this to program artificial systems to behave like biological ones and essentially self-construct? How could these principles be implemented in technology, hardware or software? In the next two chapters, chapters 4 and 5, we will study self-construction more theoretically from the biological and the engineering side.

Chapter 4

Theory of Self-construction of Biological Organisms

The basic principles of the production of proteins were explained in chapter 2. The task at hand is to create an abstraction that can explain how we can get from gene expression (the production of proteins) to the behavior of a cell as observed in biology. In this chapter some initial steps to give a framework that allows for such an abstraction are described.

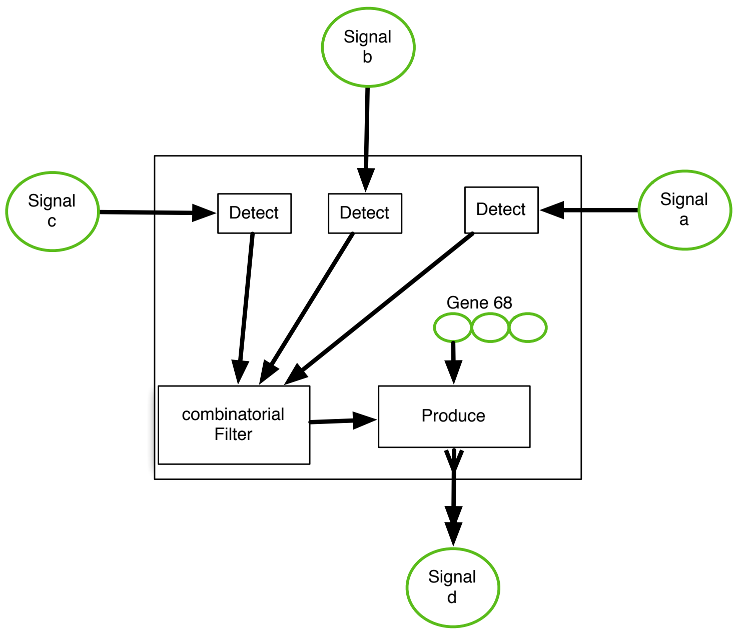
We start with the idea that proteins can be seen in two ways: as active metabolizing entities that bring energy into the developing system, or as passive information carriers. From this single gene oriented view, we make the transition to a network of genes that interact with each other, but where it is still possible to map the network of genes onto a single strand of DNA in a textual form.

This network of genes can be viewed as a high dimensional function represented by an expression landscape. In this landscape the cells of an organism operate and essentially move through a sequence of states defined by the expression landscape. The states in the expression landscape are perceived by an observer of a developing cell as cell types. We show in this chapter that the same notion of production of genes not only gives the cells a state or type but also seamlessly transitions to functional behaviors of the cells, including the observed changes in morphology. The goal of this abstraction is to give a framework in which development can be understood from single genes to cellular behavior.

4.1 Active vs. Passive Signals (Proteins)

The genome is pure information until it is translated into proteins. These proteins serve a function and have certain properties. We begin the task of describing our model by defining how we abstract proteins. Models of the real world never carry the complete description of an object into the abstract model space; it is up to the modeller to choose the properties necessary such that the model captures the important aspects of the real world object. The same goes for our abstraction of proteins, we use two abstractions of proteins in this chapter. They are either metabolising, actively bringing energy into the developing system and making the cell behave in a particular way, or they are passive and act as pure information carriers. The distinction between active and passive is purely abstract and depends on the aspect of the produced protein in which we are interested. In the figures we draw a circle for a passive signalling protein and a square for an active machinery protein depending on whether we abstract a protein as active or passive.

Transcription factors are an example of proteins that can be seen in these two ways. On the one hand we abstract the concentration of a transcription factor protein as a passive signal telling how much it influences the transcription of the promoted gene. On the other hand we abstract the transcription proteins functionality as the detection of its signal and the combination with other transcription factors mimicking the function of the regulatory region of the gene. The functional part of the transcription factors is shown as the 'Detect's and the 'Combinatorial Filter's in figure 4.1. Only the exact combination of these transcription factors will lead to the activation of this specific gene. Therefore the translational machinery can be seen as having exclusive access to that specific gene and it can be thought of as if the specific production machinery encapsulates this gene. The transcription complex itself can be seen as the 'Produce' in the diagram. 'Produce' also incorporates the whole transcription and translation process. The 'Produce' in our abstraction has two possibilities corresponding to our view of active and passive proteins. It can produce a protein we interpret as a passive signal such as a transcription factor, or it can be seen as producing a functional element that influences the behavior of the cell.



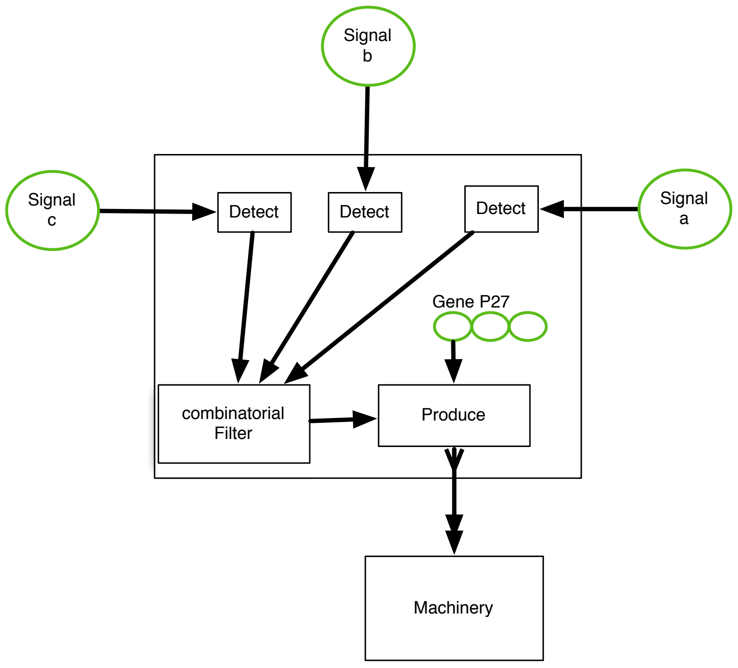


Figure 4.1. Two production-machineries as combinations of transcription factors (signals and detect's), core promotor (detects, combinatorial filters) and translation complex (combinatorial filter and producte) regions. The gene translated by the machinery is embedded in the machinery, bound by the transcription factors and the translation complex, exclusively translatable by this production machinery. The passive properties of each transcription factor, defining how much it is expressed in the cell, are shown in green. Dependent on what aspect we are interested in we draw the produced protein as a passive signal, or an active Machinery. In order to distinguish between a cooperating information transmitting relationship and a production relationship we use two different types of arrows: The normal single-headed arrow stands for a cooperative relationship and the double-headed and feathered arrow for a creational relationship.

4.2 Cellular States as a Consequence of Transcriptional Networks

Considering an ensemble of many production machineries as described above we can make the link with the genetic networks as described in chapter 2. In this chapter we investigate and analyze interacting genes that regulate each other through transcription factors generating transcriptional networks and integrate the resulting ideas into our abstraction. The analysis is based on the production machinery described above.

4.2.1 Interaction of Multiple Production Machines

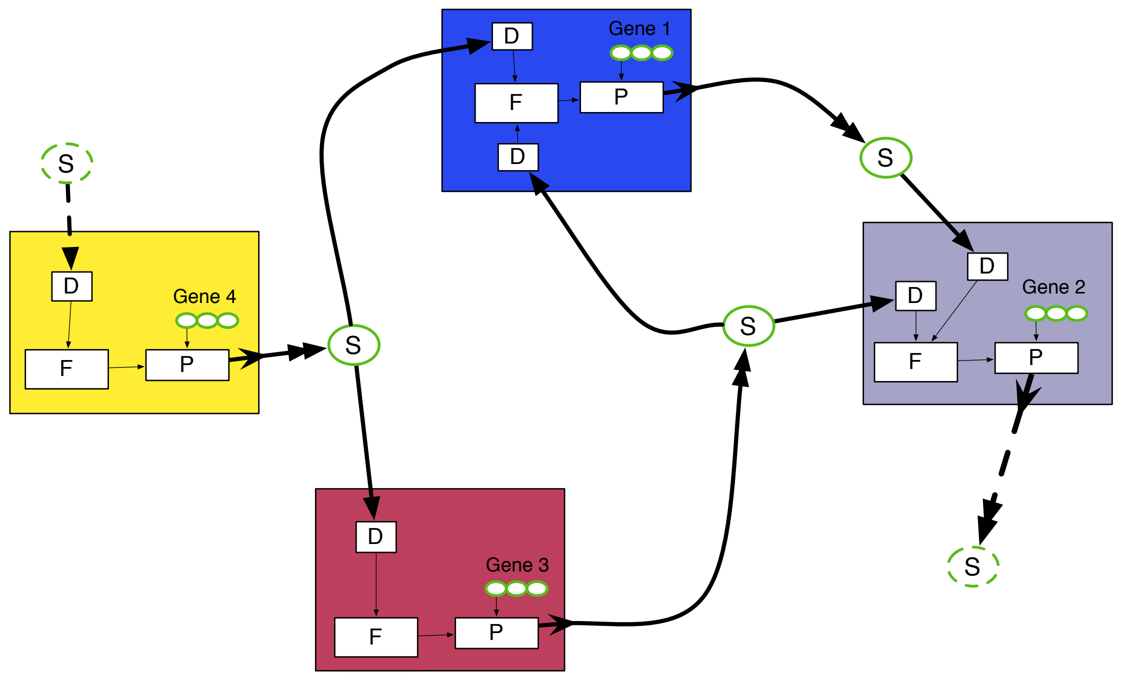
A developing organism begins with one cell containing a genome. The initial cell is placed in a particular environment and begins to proliferate and generate an organism. During this process the genome is inherited by all the cells.

The genome does not change over the lifetime of the organism (we idealize here by ignoring cases such as random mutation during development, inclusion of genetic material from viruses etc.). This code defines the final shape of the organism. The genes that build the genome can be expressed as proteins. Each gene builds many of the same type of proteins when it is activated and we speak of a concentration of proteins in a cell. The concentration can be regulated: there can be more or less of a protein expressed in a cell. The concentration is the readable signal of a transcription factor in our abstraction. The expressed genes and therefore the protein concentrations can activate or inhibit each other. This activation and inhibition is steered through transcription factors and their production-machinery (figure 4.1). This regulatory process happens inside each single cell separately. We can view this activation-inhibition process as a network, the so called gene regulatory network (figure 4.2). Each gene can be abstracted as a production-machinery and the transcription factors explained above (figure 4.1).

We have made a distinction between regulatory genes (production-machines) and genes that influence the behavior of the cell (machines). We are aware that this distinction is a simplification but it is helpful for the comprehension of what we describe here.

Taking a description of all these regulatory genes and behavioral genes together and aligning them in a string we end up with the genome for the organism (figure 4.3) in which production-machines that produce transcription factors (signals) are regulatory and production machines that produce machines (active) are behavioral genes. If we show the DNA as a strand we can imagine that the production machines are aligned in a sequential fashion all working in parallel on the DNA code producing the activated proteins (figure 4.4).

The important message in these figures is that it is possible to have a textual, completely passive form of the genome present in a string-like fashion that is sufficient as a description for



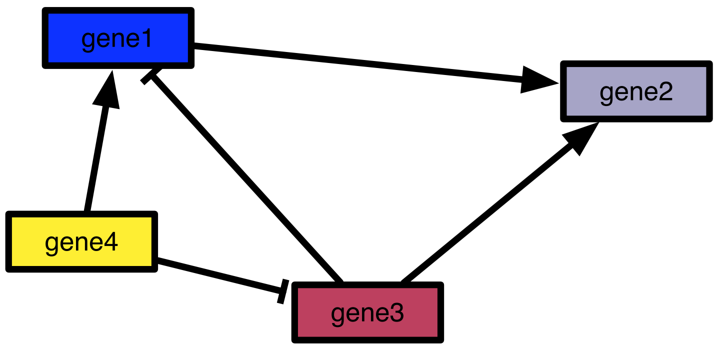


Figure 4.2. A gene regulatory network showing the interaction between different genes as a network using production machineries and transcription factor signals activating and inhibiting each other. Top: a network of production machineries. The normal arrows stand for cooperative relationships and the double-headed arrows for creational relationships. Bottom: a simplification of the upper diagram. Arrows indicate activation and lines with T-shaped ends indicate inhibition of the gene.

the creation of a self-constructing organism.

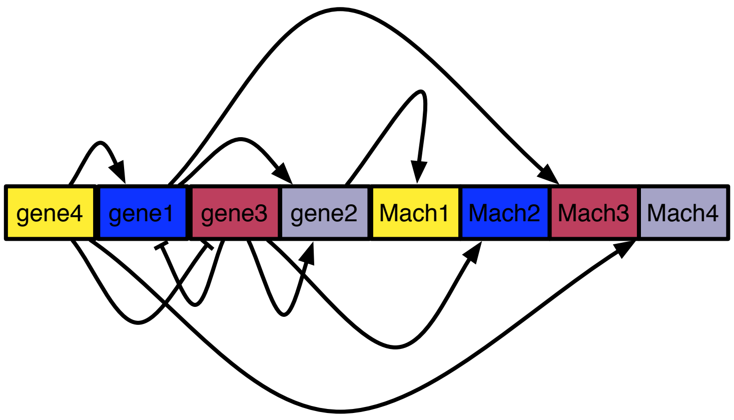


Figure 4.3. Different genes can be aligned in a strand where some genes act in their expressed form as transcription factors acting on other genes (labelled gene), and some of the genes act as machines that will influence the behavior of the cell (labelled Mach). The arrows indicate the direction of the influence between the genes.

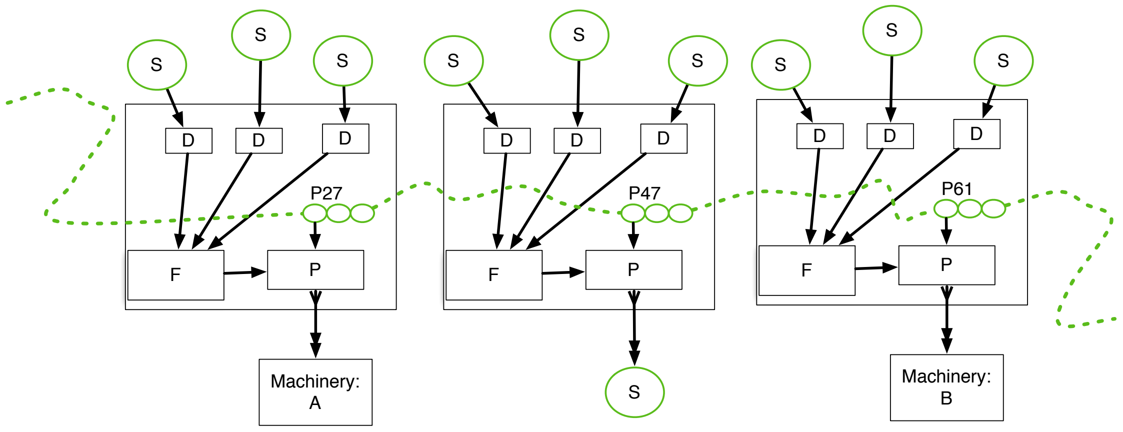


Figure 4.4. Production-machineries bound to the DNA, encapsulating it and interacting with the signals in the cell to produce the appropriate protein. The normal arrows stands for cooperative relationships and the double-headed arrows for creational relationships. Different genes and their production machines can create other machineries or signals that act together in the cells for its functionality. Here we show three different exemplars that are on the genetic string right next to each other. These three show a concept and do not represent any real existing gene production machines.

4.2.2 Genetic Landscape

Taking a closer look at the gene interaction as a gene regulatory network as described above, it can be seen that this network describes a function of gene interactions in a multidimensional space. This function and therefore the gene regulatory network spans an abstract landscape called the GRN landscape and shapes its valleys and mountains. This landscape is a continuous space where each dimension corresponds to the expression level of one gene creating a huge dimensional space. Here we draw an extremely simplified version of a two dimensional cut through the GRN landscape showing stable regions as valleys and unstable regions as mountains (figure 4.5).

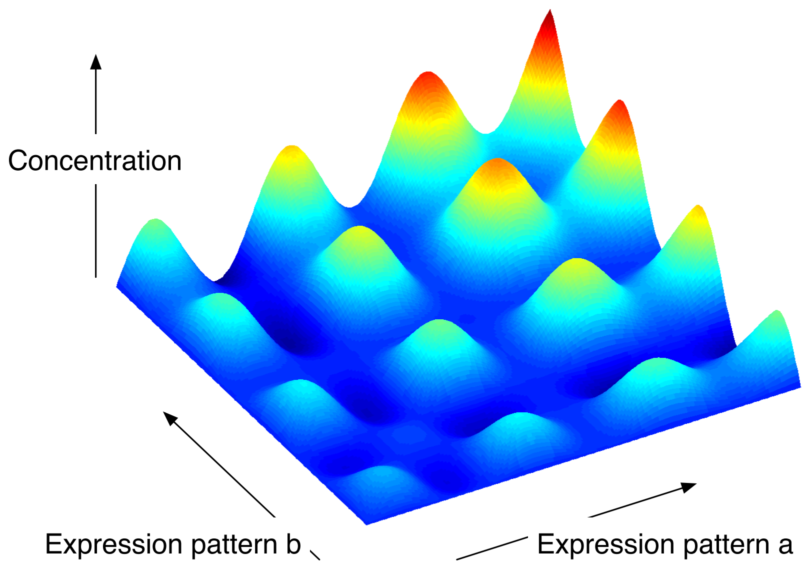


Figure 4.5. A simplified version of the complete huge dimensional space of gene expressions in two dimensions with the intensity of the gene expression (concentration) in the third dimension. In this projection onto two dimensions the axes of the projection plane are a linear combination of different gene expressions. The landscape slopes down from the top to the bottom overlain with a pattern of mountains and valleys. Even though the landscape appears very regular and traversing between valleys extremely simple, this figure is of course only an illustration of a much more complicated multidimensional landscape.

Each point in the GRN landscape is defined by an expression pattern of all the genes in the system with a unique address in the GRN landscape. Since the genome will not change during development, the landscape that is defined by the gene regulatory network will also not change during development and will remain constant. On the contrary the expression pattern of the gene regulatory network will change over time in each cell. And in each cell the expression pattern is different, in some cases only slightly different and in others grossly different. The valleys in the diagram are stable regions where the gene expression will not change much, whereas the steep mountain slopes (unstable regions) will generate extremely fast changing expression patterns. At any point in time the expression pattern might be seen as the state the cell is in.

Waddington's ideas come quite close to the idea of the GRN landscape [196]. (An illustration of the so-called epigenetic landscape can be found in his book). He describes a multi-dimensional space that is defined by the genes, and casts this idea into a mental image of a two dimensional sheet that is being formed by genes. The genes act as poles with ropes that then pull on the sheet and make it rugged. These ropes can be influenced by other genes through an attached rope and one pole can have multiple ropes attached therefore forming a kind of a rope network that would correspond to the influences of genes on one another, essentially forming a regulatory network for the epigenetic landscape. On this epigenetic landscape Waddington imagines a ball rolling down a valley as an analogy of a cell following its path to differentiation. We can clearly see the similarities of the epigenetic landscape and the GRN landscape but we will go one step further than Waddington. We will show later in this chapter how multiple cells can navigate in the GRN landscape, what replication means in the context of the landscape and how behavior emerges as an epiphenomenon of the GRN landscape.

To give the reader a better understanding of how a gene regulatory network creates a GRN landscape with the function it builds, the example of a bistable switch is shown in figure 4.6. The bistable switch is a motif of gene regulation that can be observed in real biological gene networks [39]. In this example the genome consists of only two genes, gene A and gene B. These two genes interact with each other. Gene A produces a protein that enhances the expression of gene A and inhibits the expression of gene B using the binding sites in the promotor region to influence its production (as described in the background chapter, chapter 2). Gene B's protein on the other hand activates its own expression and inhibits gene A. Through these interactions the gene regulatory network forms a function that expresses a landscape with two dimensions that has two regions of stability: only gene A is expressed or only gene B is expressed. These two regions are attractors. There is a third region that is an equilibrium in which A and B are expressed exactly equally but this region is a ridge and therefore unstable. Any other configuration of expression patterns will lead to one of the two attractors. A more detailed description workings of gene-regulation are to be found in the thesis of Pfister [146].

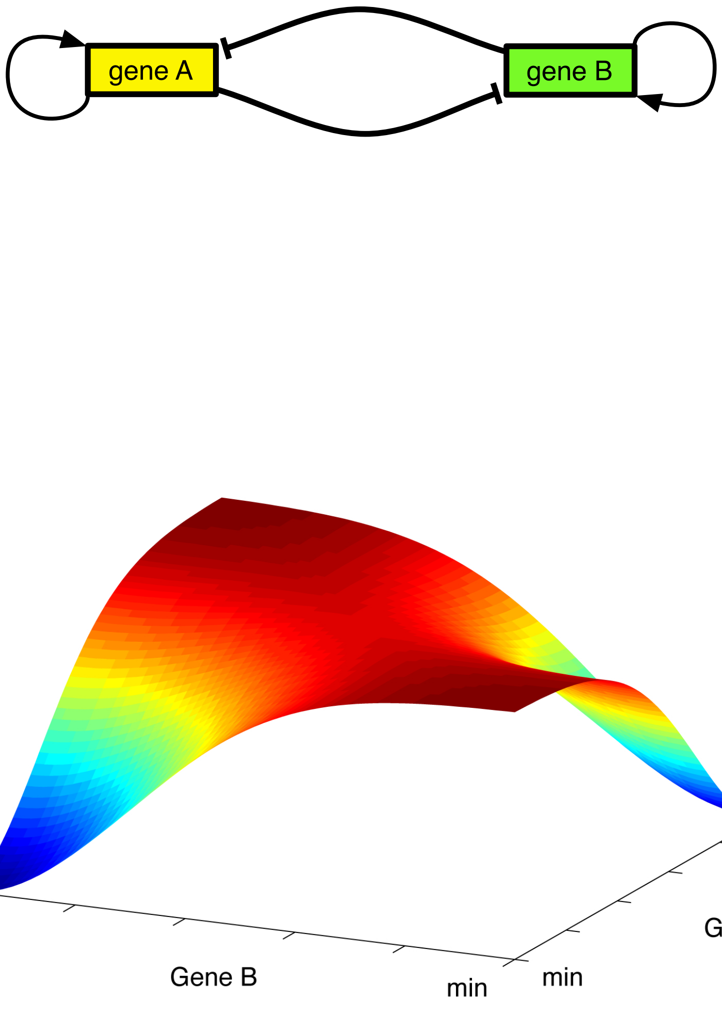


Figure 4.6. Top: A gene regulatory network consisting of two genes: gene A and gene B. Gene A activates gene A and inhibits gene B and vice versa. The gene regulation leads to a two dimensional GRN landscape function (Bottom) that has two attractor states: gene A fully expressed or gene B fully expressed.

Each point in the GRN landscape corresponds to a particular expression pattern. Cells with similar expression patterns [167] and therefore similar behavior [80] and morphology [167] are classified in biology as a cell-type. Cell types in the GRN landscape can be seen as confidence intervals (regions in space) within which a cell is considered to be of a certain type because of its expression pattern. Therefore cells in the same region of space within a confidence interval are classified as the same cell type. This means that a certain state region defines a cell type. Figure 4.7 shows certain cell types as regions in the landscape defined by the black lines surrounding them. The black line can be seen as the confidence interval in which we would classify cells as being of the same cell type.

4.2.3 Transcription Networks Seen as Classical Dynamical Systems

We can examine the state changes of the cell including the influence of the environment with the following dynamical system equation:

$$\vec{x_{t+1}} = A\vec{x_t} + B(\vec{x_t})\vec{x}_{text}$$

In this equation $\vec{x_t}$ is the current state of the cell; the expression pattern it has seen in the gene expression. A represents the function of the GRN landscape that will modify the current state into the next one following the gradient of the function A. The equation $\vec{x_{t+1}} = A\vec{x_t}$ therefore describes the cell internal state dynamics ignoring the external input coming from the local environment. $B(\vec{x_t})$ describes the different inputs that the cell can read in the state $\vec{x_t}$. $B(\vec{x_t})$ is state dependent because depending on the gene expression of the cell it has activated different machinery to read the input. Since all the signals that are being read by the cell must be actively read, a machine for detecting the external input must exist. (This is explained in greater detail in section 4.3.3). $\vec{x}_{text}$ describes all the signals in the environment that the cells can potentially read.

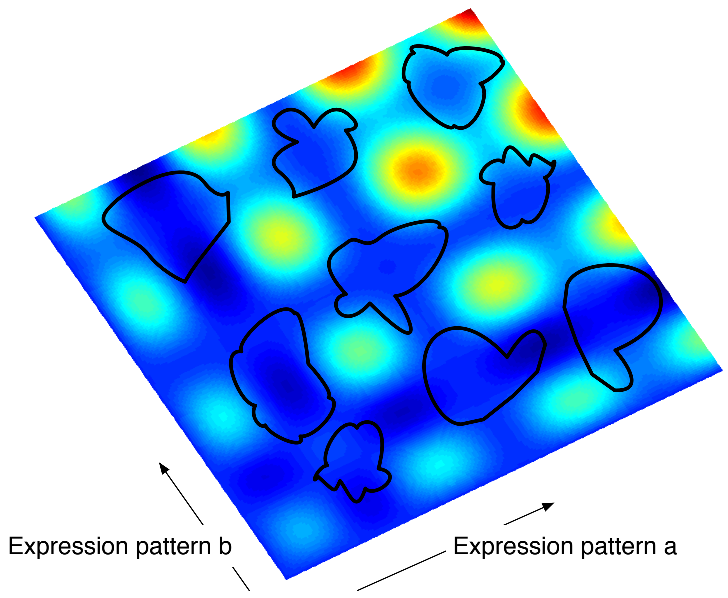


Figure 4.7. Black borders describe the confidence intervals within which we associate a cell with a certain cell type. There can be multiple such cell types in one GRN landscape. Cell types are only human-perceived categories for cells. This figure is actually the same landscape as depicted in 4.5 but seen from above. The stable regions that represent cell types are the vallys of the landscape, where the cells usually can be observed and keep a stable gene pattern over a period of time. Cells will populate this landscape and especially the stable regions.

4.2.4 Cellular Development as a Walk through the GRN Landscape

As we already described, each point in the GRN landscape is unique in its gene expression pattern. Cells from the same organism navigate in this expression pattern landscape and describe over time a trajectory between different points of this landscape figure (4.8). The trajectory of a cell in the landscape does not necessarily mean that the cell will leave the region of space where we consider it to be of one cell type, but it could do so. Of course there are also stochastic processes at work that make the gene expression pattern change which lead to variations and not exact trajectories along the gradient. However this noise in regulation and therefore in trajectory is largely compensated by the effect of stabilization that occurs through the attractors in the landscape. Waddington describes a similar gene regulation process in his epigenetic landscape [196]. In his analogy of a ball rolling down the slope of the epigenetic landscape, the valleys have a stabilising effect on the ball and therefore on the cellular development. Waddington calls the stabilising effect that the valleys create the *creode* (a combination of the greek roots for *necessary* and *path*).

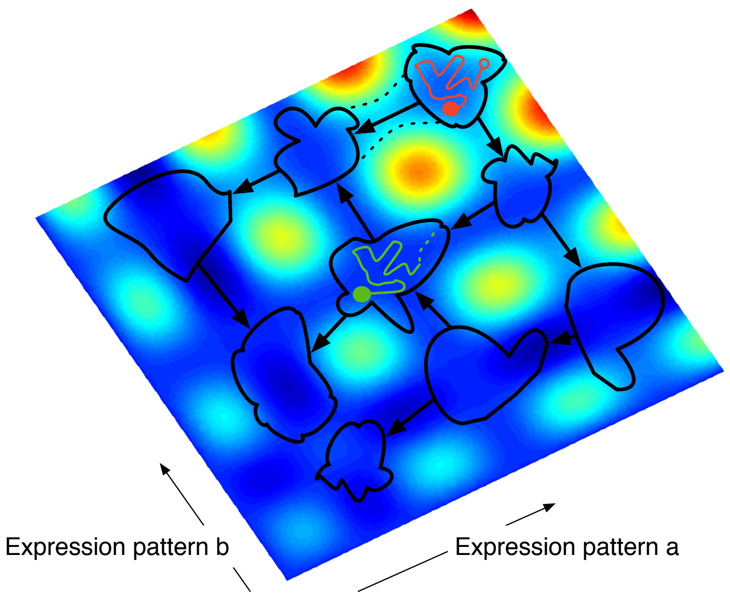


Figure 4.8. The transition diagram between different cell types. The green and the red points are cells operating in the landscape and the green and red lines are their past trajectories. The arrows show specific pathways along which cell types can be reached from other cell types. The regions that do not have any arrows leaving the region are regions of fully differentiated cell types. The cells that arrive in these regions will stay there.

Now we can imagine a trajectory of the states an initial cell must have taken during development to reach one of the final cell types (figure 4.9).

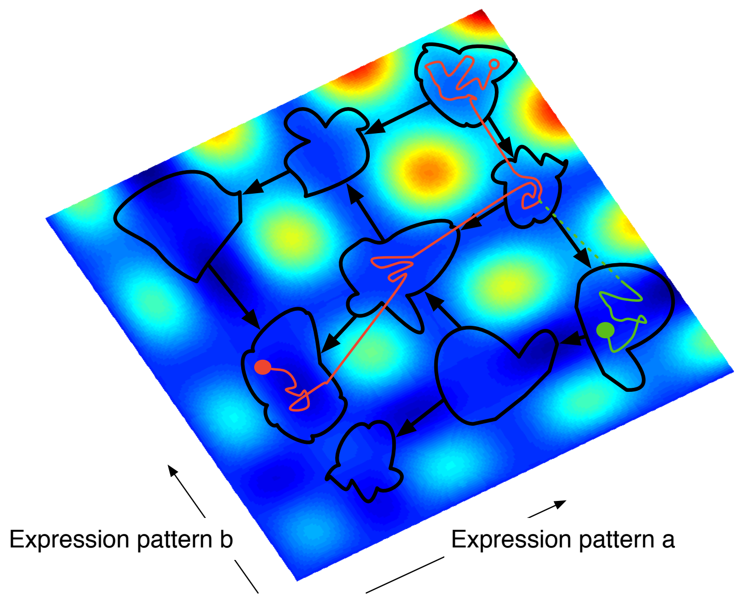


Figure 4.9. In red, the path followed by one initial stem cell after each division until it reaches a state of full differentiation. In green, an offspring cell following another path after a division. The regions represent different cell types and the arrow indicate the transitions between the cell types. Because cells can divide asymmetrically and distribute the expressed protein content asymmetrically into their offspring cells can take different paths during the development.

Long before DNA was discovered, cells have been categorized into types [155] defined by their shape, function and behavior; nowadays they would be determined by gene expression patterns [167]. These measured expression patterns are an epiphenomenon of the gene regulation. The gene regulation will activate the behaviors of the cells that will aid in the construction of the final organism and assure its survival after development. This means that at different points in the GRN landscape functional genes are activated to switch on a certain behavior of a cell through the expression of functional proteins. These behaviors can be shared between different cell types. For example the necessary genetic machinery for replication stays accessible for many kinds of cells not only for one type. There are many such examples of cell behaviors that are shared between different cell types (e.g. migration [151]). In general, the type of a cell is assessed through the sum of its behavioral characteristics. Cell types, although very well justified, are only abstract classifications that help humans understand of their functionality. We understand the cell-type in our GRN landscape as a sub manifold, a subregion that clusters the cells of a certain type within its boundaries.

Each of these behaviors needs to be activated by one or multiple genes. Assuming it is activated by a single gene, this would mean that in a dimension of our GRN-space a concentration has to reach a certain threshold for this behavior to be activated. We cannot show the activation of behavior in our landscape diagram directly but would need to switch projection planes to show the appropriate dimension (figure 4.10). Because of this, certain properties are accessible through dimensions we do not draw in our diagrams and are therefore accessible from multiple points on our surface. (Shown in figure 4.18 in section 4.3.)

It is possible to access multiple gene expressions for different behaviors for a single point in our projected landscape. For visualization purposes one cannot see those other dimensions. Due to this high dimensionality it is moreover possible that multiple points of the projected landscape have access to the same behavior.

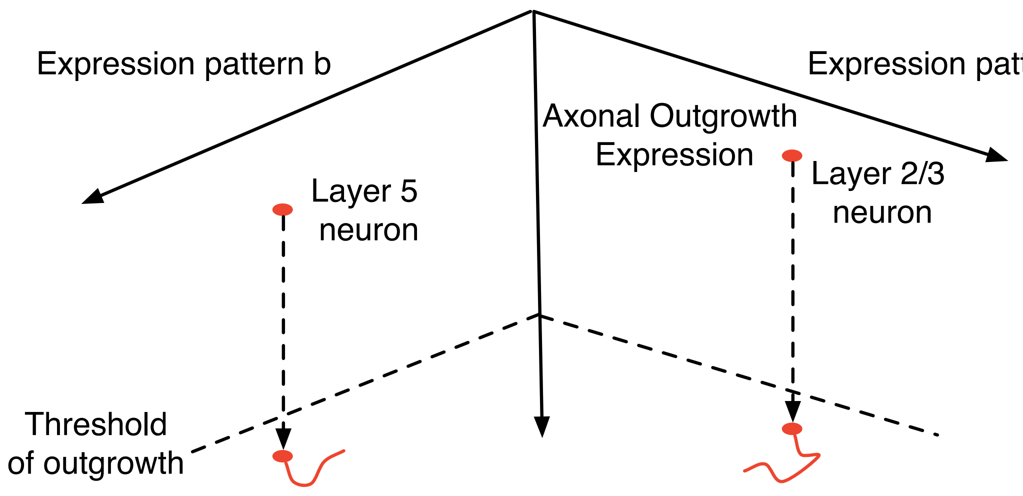


Figure 4.10. The GRN landscape turned such that the dimension of axonal outgrowth is visible. Two example cells are shown, a layer 2/3 cell and a layer 5 cell, that are beginning to express the gene that is responsible for the outgrowth of an axon. If a certain threshold of this expression is reached the cells will grow axons. This dimension is in principle accessible to any cell, given that the function that the gene regulatory network builds permits the cell to express the gene. Since the two example cells follow the gradient that the GRN function forms and the expression of the axonal outgrowth factor is part of that gradient path, the cell will inevitably extend an axon.

4.2.5 Cellular Division (Replication) in the GRN Landscape

The initial stem cell will not stay alone in the developmental process but it will divide and create many cells to build a fully functional organism. Division of a cell is a functional process, that has to be mapped to cascades of gene expressions. In order to reach a state where it is possible to divide, the cell has to reach a position in the GRN-space where the gene is activated that codes for the division machinery (figure 4.11).

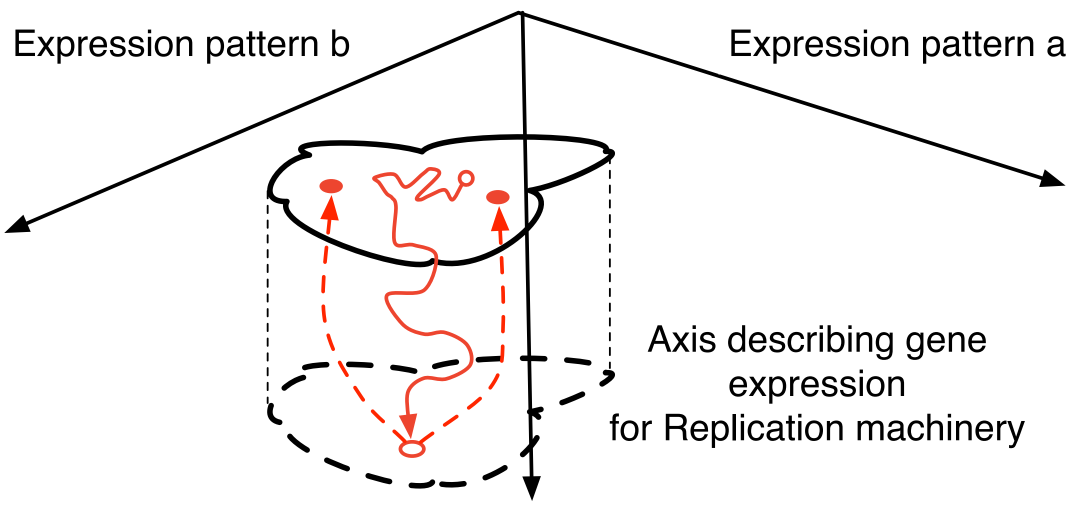


Figure 4.11. A projection in three dimensions, emphasizing the expression of the division machinery in the vertiacal axis and the GRN landscape in the horizontal axes. After a division there will be two cells which instantly have different expression patterns than their mother cell. Here the cells do not have an expression pattern that is different enough to place them into different cell types after a division.

When a cell divides it has to split its contents (chemicals, expressed proteins...). These contents are the expressed genes, this means that during a division process the mother and the daughter cell will have to divide up the expressed genes, namely the quantity of all proteins. The mother cell can define which expressed genes will be distributed in what proportions to itself and the daughter cell [157]. Assuming it would distribute the contents equally we obtain a symmetrical division, which might leave both of the cells in the same region of the landscape as the mother cell has been in (figure 4.12), or, it might cause the cells to end up at different points in the landscape changing the cell type of both cells (figure 4.13). Another possibility is that the mother cell divides the expressed genes asymmetrically between the two cells and we obtain an asymmetrical division making one daughter cell change its cell type but the other cell keeps the mother's cell type (figure 4.14). In any case the cells will make a jump in the GRN landscape, because of the sudden change of concentration of the genes expressed.

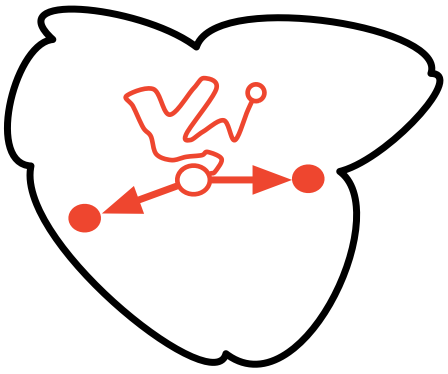


Figure 4.12. The division of a cell in a region of space where the daughter cells will remain the same cell type after the division. This type of division is called symmetric division. The content of expressed proteins is divided in same amounts to the offspring cells making them copies of each other. In this particular case they stay in the same region of the GRN- landscpae as the mother cell and are therefore not changing cell types compared to their mother cell.

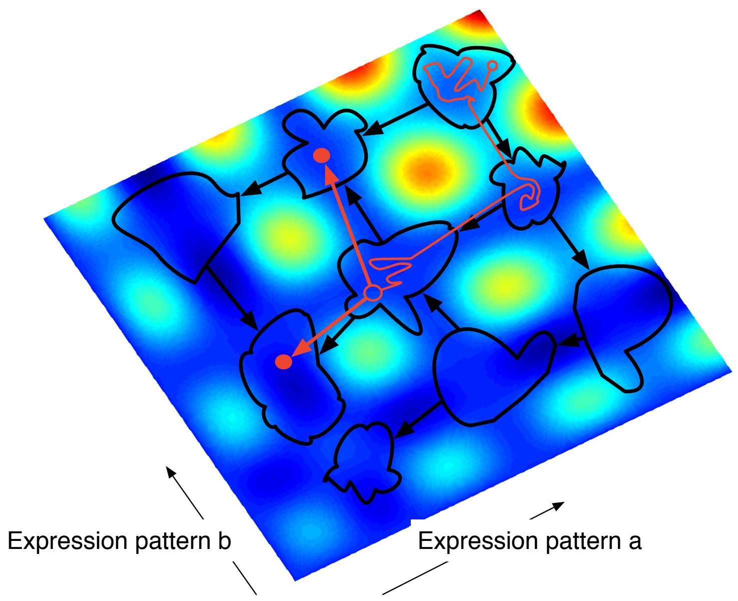


Figure 4.13. A proliferative cell dividing and giving rise to two cells in different states than the mother cell was. Each of the new cells' state lies in a completely different region of the GRN landscape therefore both cell types change from the original cell type.

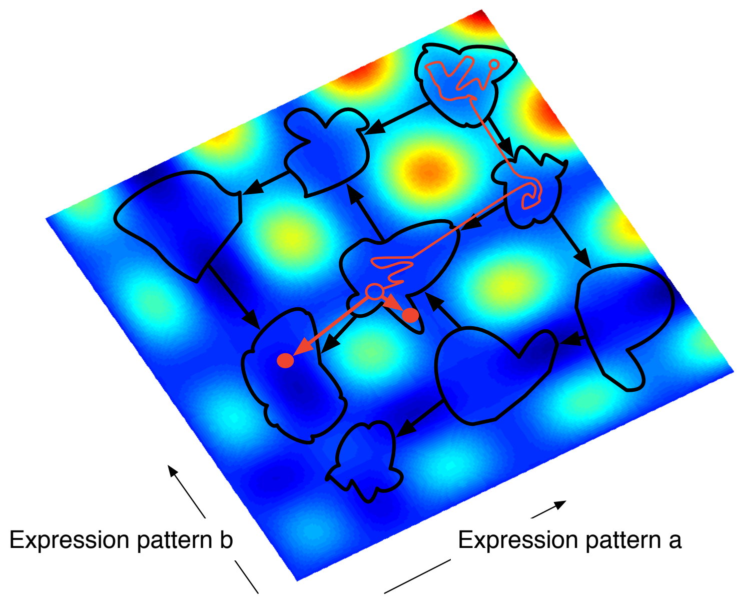


Figure 4.14. Asymmetric division where the mother cell stays in the same region of space and gives rise to a daughter cell with a different state. That state is in another region of space that lies in a confidence interval of a cell type other than that of the mother cell.

Asymmetrical divisions give the organism a powerful tool. It makes it possible to follow two different trajectories on the landscape, therefore breaking the symmetry and allowing different parts of the state space to be explored by different cells, opening up the possibility for cells to behave differently even though they have been replicated from the same source. Allowing more cell diversity and a more complex construction processes leads to the possibility of complex self constructing organisms as we observe them in biology. As soon as the first cell has divided, the construction process begins to be parallel. Two cells navigate independently in the GRN landscape. By the end of the organisms development the landscape contains billions of independently navigating cells.

In Waddington's abstraction the cell fate is the consequence of the cells following the valleys in the epigenetic landscape. When a cell reaches the end point of the valleys it is completely differentiated and therefore no longer moves through the landscape. Waddington does however not explain exactly how cells choose the valley to go down and he does not describe replication in the context of the epigenetic landscape.

4.2.6 Gene Regulation as a State Diagram

If we assume an initial cell beginning the development of a new organism, we see that the initial cell will at some point differentiate and be integrated into the final organism. That means it will change its state and go through a sequence of cell types. Hence there must be a sequence of transitions between the initial cell and all the reachable cell states in the organism otherwise certain cell types would never be expressed. There must therefore be a sequence of transitions between stem cell types and the final differentiated cell types.

Since there is not just one type of proliferative cell but multiple types and they must all have arisen during development from the initial cell, we can draw diagram of transitions between different proliferative types and to the types of their differentiated offspring. We end up with a state diagram overlaying the GRN landscape figure (4.16) in which a cell type with outgoing transitions is a proliferative type and the leaves are fully differentiated cells. The starting state of this state diagram is where the zygote starts at the very beginning of the existence of the organism. Gene-expression pattern can therefore be seen to give a state to the cell. And the gene regulatory network can be seen as a state machine that defines all possible states a cell could have in the system and defines all the transitions between the states.

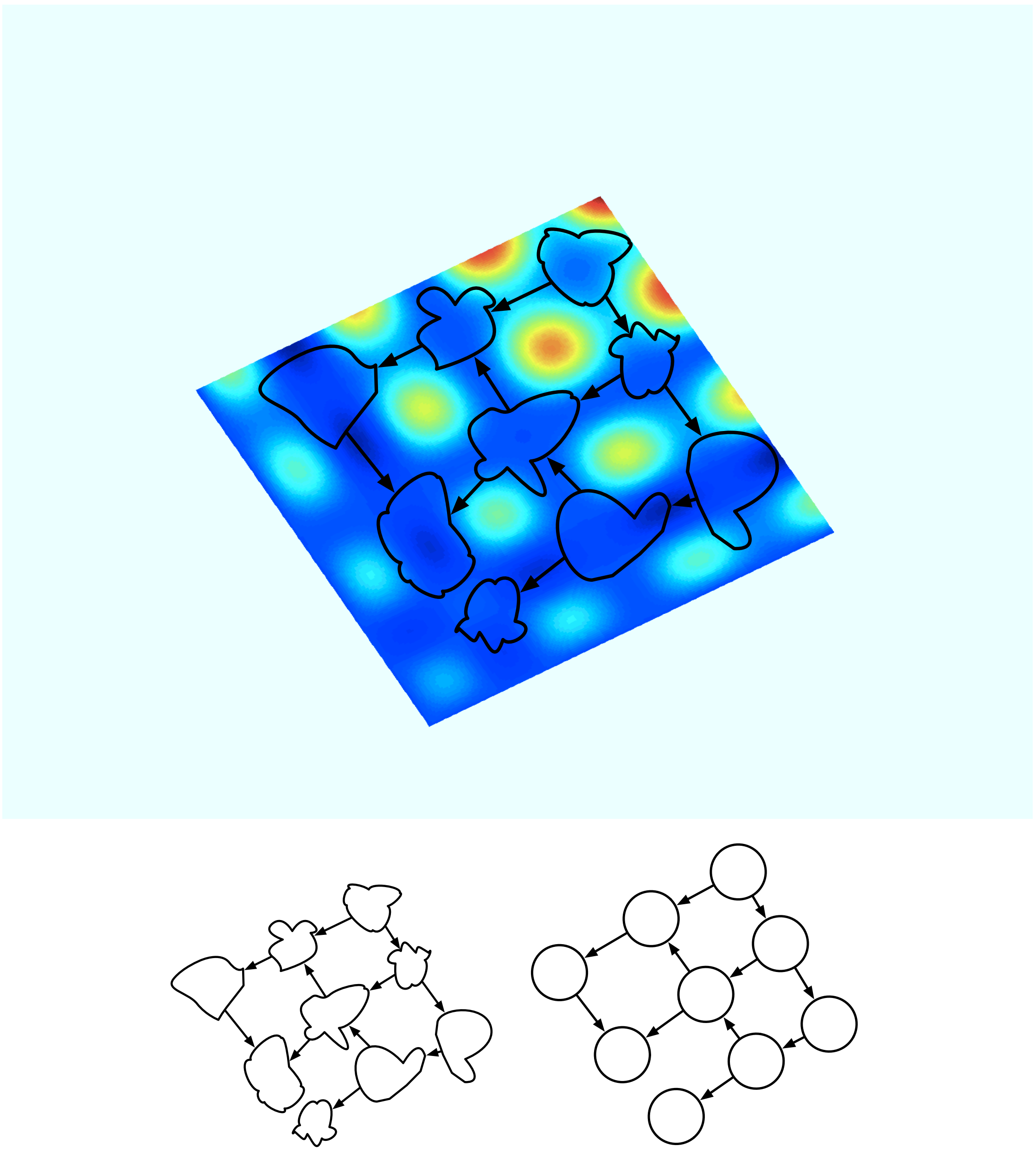


Figure 4.15. Abstraction levels of state diagrams that show the transitions between different cell types. On the top we depict the GRN- Landscape, with its cell type regions and the transitions between the different states. This Landscape can then be abstracted as only the stable regions and their transitions (bottom left) to a fully abstract state machine diagram (bottom right).

Since the regulation of genes is running separately in each cell and the cells can have different expression patterns as described above, it is the case that each cell is in a different state but operating with the same state machine, still completely controlled by the developmental process. Having all the cells controllable in different states during the development leads to a state machine that is running in the overall organism in a distributed way. With each replication of cells the system has another actor playing out the GRN state machine. This will make the system even more distributed (see figure 4.16).

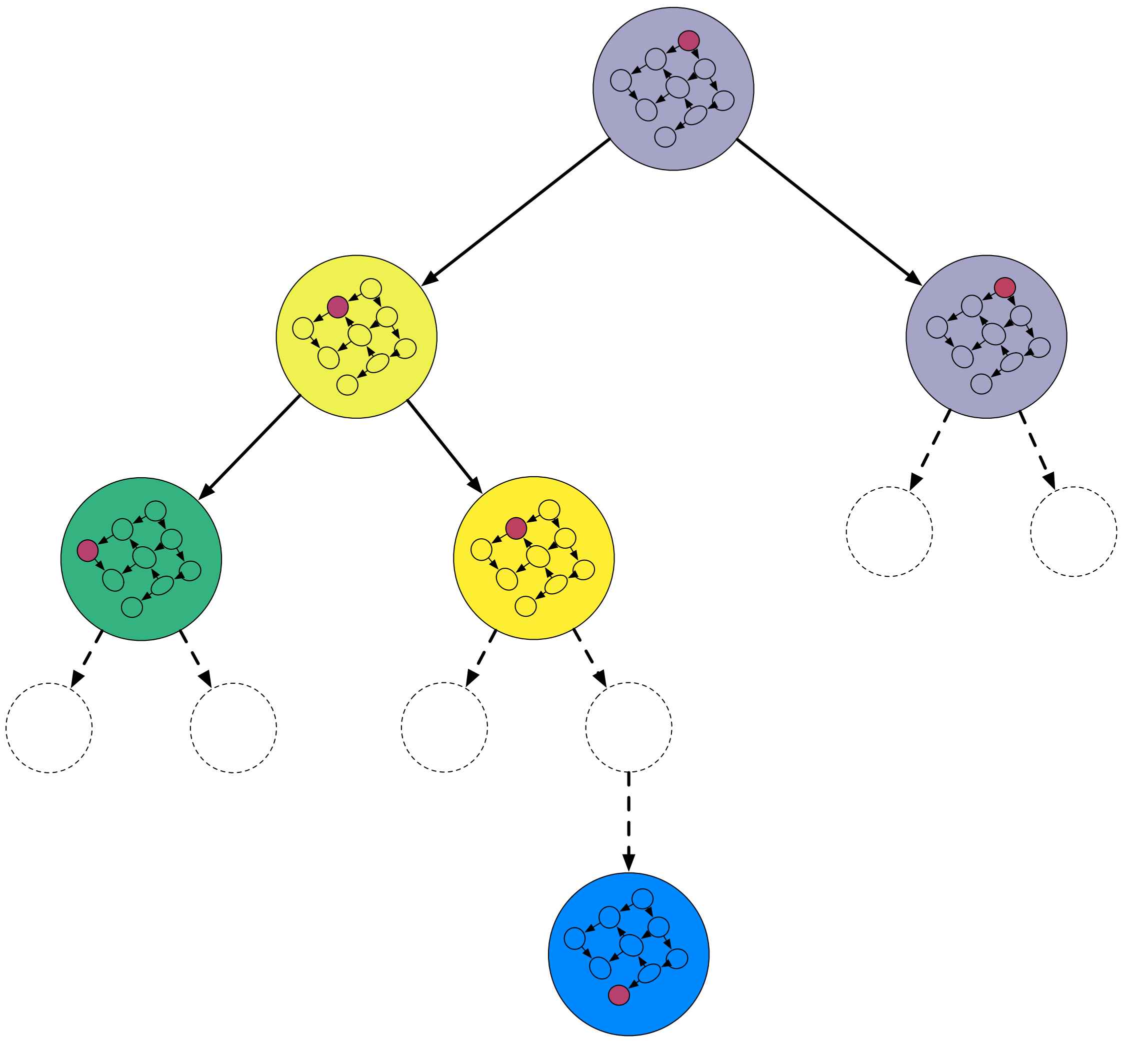


Figure 4.16. A Cell lineage tree. In each replication the state of the daughter cells can change according to the state diagram (a GRN Network). It will either stay in the same state or change state. At the leaves of the lineage tree, at the end of development, a final state is reached for each cell, and the cells are then called fully differentiated.

4.3 From Cell States to Cell Behavior

Cells that have reached a certain cell state or cell type undergo morphological or behavioral changes. These changes enable biological cells to find their target positions in the final structure of the organism to be constructed. Hence there must be a mechanism that links the states of the cells to the behavior that the cell should execute and that mechanism must seamlessly integrate into the abstraction that is given here. Indeed there exists such a mechanism, in section 4.1 it was described how there are production machineries that produce active, metabolizing proteins that we call machines. These machines can be activated in much the same way as other genes in the gene regulatory network, with the same mechanism of promotion as in the case of regulation in the gene regulatory network. Functional machines can be regulated and therefore activated in different states. In figure 4.17 this can be seen being played out in the state machine where a state activates a certain behavior of the cell that is defined for its cell fate.

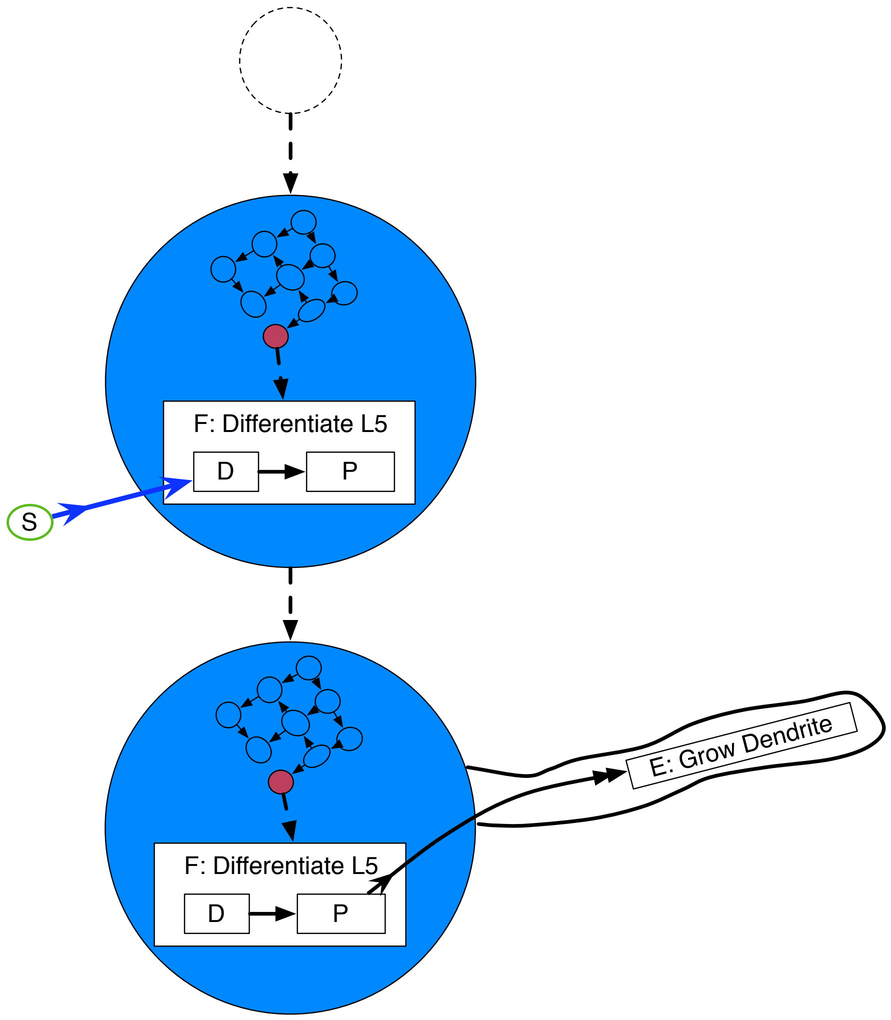


Figure 4.17. This figure is an extension of figure 4.16. We show an example of a cell which has reached the last differentiated state which happens to be the state in which a cell becomes a layer 5 neuron. Therefore the state activates a machine F: Differentiate L5 that will express the morphological transformations of the cell to a layer 5 neuron. This machine starts another machine E: Grow Dendrite that will begin to grow out a dendrite, given the right condition in form of a signal S in the environment is present. The normal arrow stands for a cooperative relationship and the double-headed arrow for a creational relationship.

Lets take a step back in the abstraction going back to the GRN landscape. As we have shown in section 4.2.4 it is possible that certain cellular behaviors such as axonal outgrowth are reachable by different types of neurons. To visualize the process of the cell starting the axonal outgrowth machinery we needed to re-project the multidimensional GRN-space to include this dimension. That certain functions are accessible to different cell types is generally true. The multi-accessibility of functions implies potential for the reuse of machinery in cells of different types. This leads to the picture of figure 4.18 in which it can be seen that multiple of these behavioral machines are potentially activated from different points in the GRN landscape. The activation would happen in dimensions of the GRN-space that are not shown in the figure; it is obvious that for each machine different dimensions are involved.

It has even been shown that proteins can be produced on demand at remote places in the cell such as in the dendrites at the synapse or the growth cone [179] [100] [180] [33] [50] [46]. This is in accordance with the idea of local machines in cellular compartments that can be assembled on demand at the needed locations. The remote production is the reason why we can safely say that machines can be produced and start in different places in the axons, the dendrites and potentially their synapses.

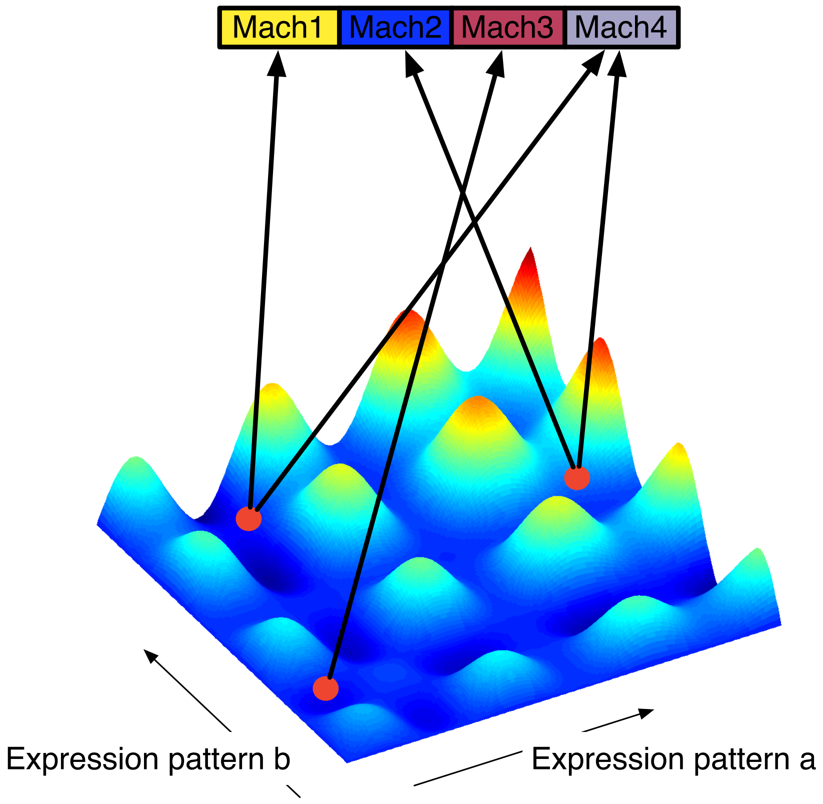


Figure 4.18. In the GRN landscape, multiple points can access a certain behavioral expression of a cell. But the functionality is only accessible via certain dimensions. We would need to project to another plane in order to see the dimension of the particular expression for a particular functionality.

Shifting the view point to the gene regulatory network as an ensemble of interacting production-machineries it can be seen that the gene regulatory network and the behavioral functional machines interact with each other through the same mechanisms (see figure 4.19). It is even possible that a behavioral machine contains a sub-GRN. This interaction implies that the sub-network is activated by the state-giving gene regulatory network, as we can see in the figure. The interaction between functionality and state neatly integrates into our framework of abstraction. It is already clear that all of the machines are assembled by primitives of functionality that are the same in the gene regulatory network and in the behavioral machineries. This fact will be expanded on in the section on the assembly of machines, section 4.3.1.

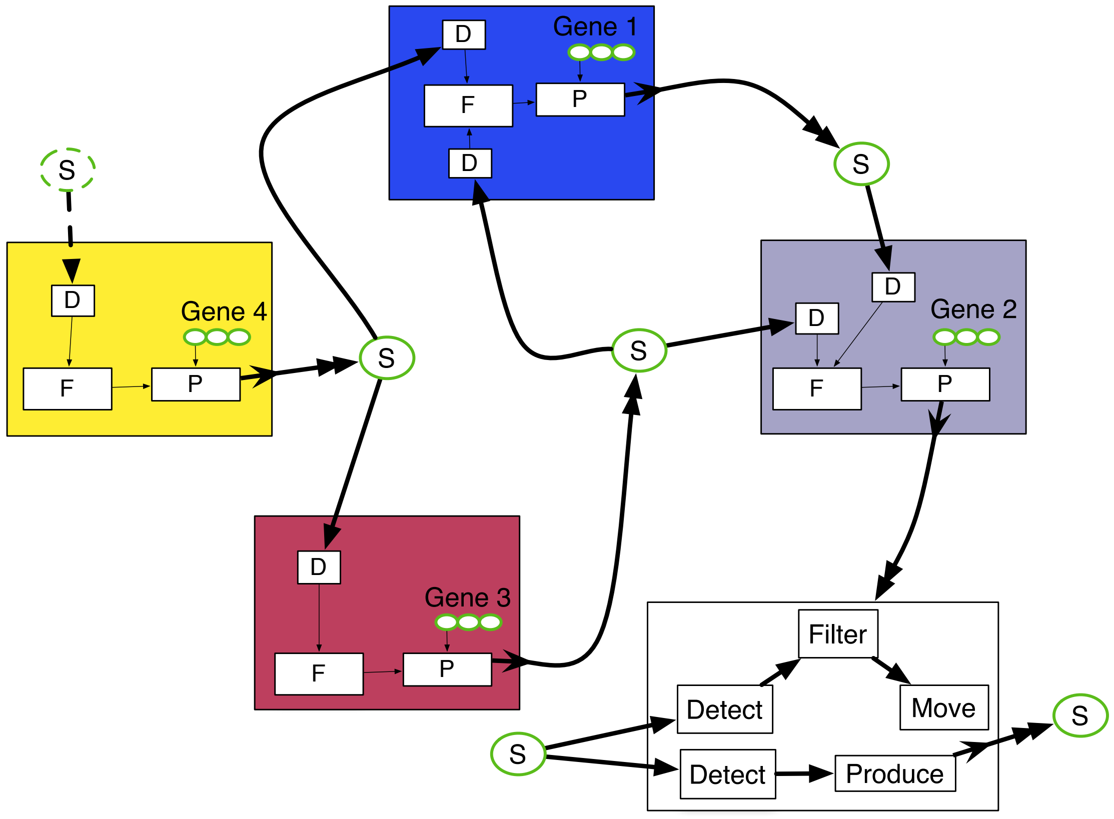


Figure 4.19. This diagram is a continuation of the diagram 4.2 on gene regulation. This diagram shows a gene regulatory network in which one of the genes, in this case Gene 2, produces on promotion a behavioural machine. The behavioural machine is composed of different primitives. This machine will detect a signal and depending on this signal it will make the cell move and produce another signal. The normal arrow stands for a cooperative relationship and the double-headed arrow for a creational relationship.

In figure 4.20 a concrete example is given in which part of the gene regulatory network from the simulation of cortical growth is shown. This shows the coordination of the machines as it happens in the simulation (see chapter 3). The gene regulatory network produces cells one after another, first layer 5 cells, then layer 4 cells and finally layer 2/3 cells. Layer 5 cells start a migration machine **C** as soon as they are born. The migration is stopped by an external signal from layer 1 cells that are already present from an earlier stages of development. The signal activates the differentiation machine **D** of the layer 5 neuron and starts the dendritic outgrowth machine **E** that targets layer 2/3. Now since layer 2/3 is not existent at this point the machine cannot yet finish its program and waits for layer 2/3 to appear. The growth cone waits therefore for an extracellular signal from other cells to continue its growth. While the other cells play out the described behavior, the progenitor cell keeps on dividing spawning layer 2/3 later in the gene regulatory network state program. The layer 2/3 behavior is activated. This is first acted out by using a copy of a migration machine **C**. The migration machine is signaled by layer 1 cells' membrane markers to stop the migration and to activate the machine **F** for the differentiation of layer 2/3. The differentiation machine in turn expresses a membrane marker that the waiting dendritic machine **E** has been waiting for. After that signal the dendritic machinery can now finally begin to act out the branching pattern that is encoded in the machine **G** Branch. This example shows that extracellular interaction between machines of different cells are also possible and desirable for shaping a final structure. We show here that it is possible to go from a gene regulatory network (states) to behavioral function of cells in a multi-cellular environment, where the cells can communicate with one another. None of the functionalities described violate our abstraction framework.

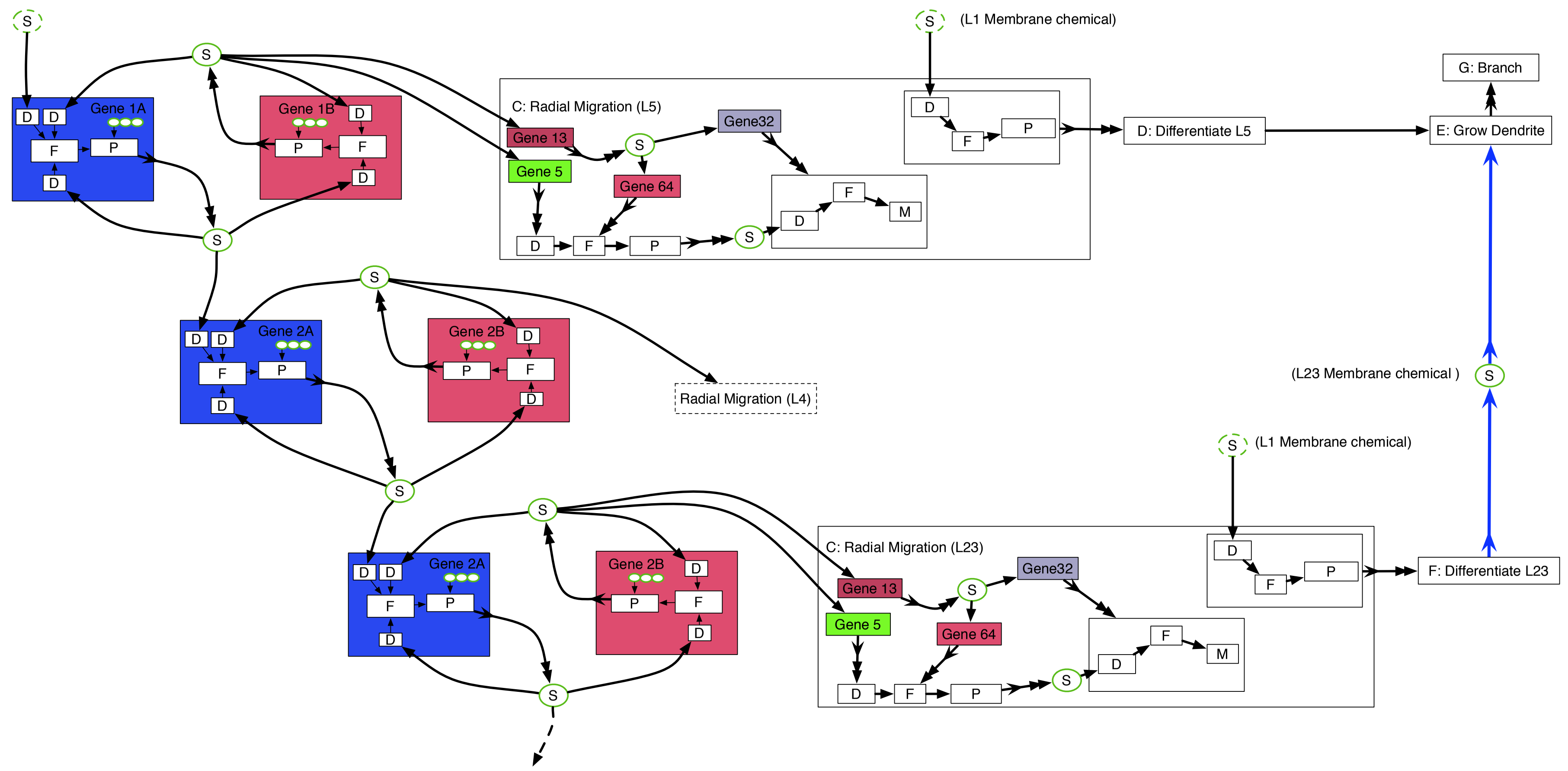


Figure 4.20. An extract from the gene regulatory network used in the simulation of cortical growth. Using bistable switches the gene regulatory network produces different states over the course of cell divisions (caused by other parts of the machinery not shown here). Three different states are shown here, in one state the cell will become a layer 5 neuron, in another a layer 4 neuron or in the last one a layer 2/3 neuron. For the layer 5 neuron and the layer 2/3 neuron the machineries are shown that make them act out their fate. This figure shows the causation and communication from the gene regulatory network to the behavioral machines that act out fate and their communication through cell-internal signals (black arrows) and external signals (blue arrows). The normal arrows stand for cooperative relationships and the double-headed arrows for creational relationships.

4.3.1 Composing Machines

A single functionality or behavior of a cell is not necessarily performed by a single protein. Often they are the result of multiple proteins binding together (an example of this multi-protein binding has been described for the DNA replication machinery [54]), or of a biochemical pathway of multiple proteins and chemicals [125]. Therefore we use a composite component system to represent a functional machinery. But a higher level machinery can always be decomposed down to the level of single protein interactions. Proteins on the other hand can participate in multiple functionalities of a cell. For example tubulin which participates in the splitting of the DNA during mitosis [170], in the migration process of cells [156] and in the growth cones in neurites [74]. This means that one functionality might rely on a specific protein. This protein might though also be used in another functionality. What behavior exactly is being called for depends on a higher level process that only recruits the protein on demand.

In order to activate the production of the high-level machine composed of multiple proteins, the production-machinery of each of these genes is activated and the machine is assembled in sequence. To have multiple proteins contributing to one single process we need to be able to let the initiating production-machinery activate the necessary protein production. Since the access to the genes is only granted for each gene by a specific production-machinery we need a system where genes can refer to other genes on demand. Hence to allow one gene to activate multiple genes in sequence to assemble a higher complex of protein transcribed from multiple genes. How the process of assimilation of multi-protein complexes exactly works is still under investigation. It is already clear that not only one but multiple processes could be responsible for the assembly of multi-protein complexes, one example is the 1D diffusion along DNA or Chromatin via weak binding [54].

Production for a higher level machine can therefore be seen as an abstract description of the whole cascade that is being initiated to assemble the machine to be produced. This production though might be more abstract and might not only involve one transcription complex but multiple transcription complexes. The term production here stands for the whole process it takes to assemble the machinery for a complex behavior a cell is exhibiting. This production machinery can therefore produce a machine that is acting on a higher abstraction level than the protein production we have shown so far but follows the same rules of activation. The possibility of higher level activation and reuse of machinery allows for a component composite scheme of recombination of partial machines to higher level ones. Functionality or behavior that is built on the basis of other already existing functionalities and/or behaviors.

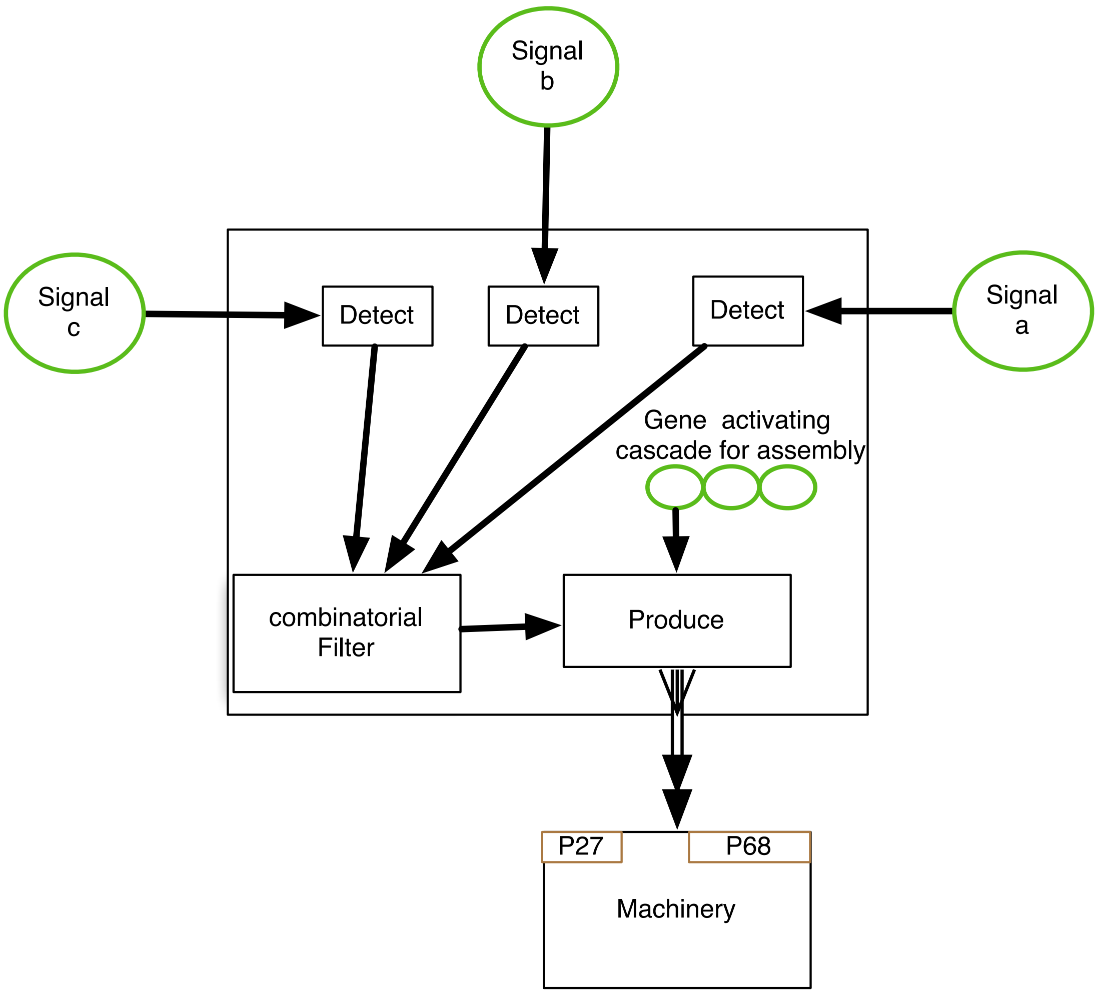


Figure 4.21. Production machinery for a higher level function that is based on multiple proteins. This figure is an extension of figure 4.1 The expression of the genome does reference the proteins. These proteins get assembled by the production process into larger protein-complexes or even biochemical pathways. Gene sequences (green) referencing genes coding for proteins (brown) in the machinery. The initiation of the production of the resulting machinery stays exclusively available to the combination transcription factors that must be available. The normal arrow stands for a cooperative relationship and the double-headed arrow for a creational relationship. The production of the whole which we indicate with a double arrow here is in fact a whole complex activation of other transcription factors and assembly processes that lead to the machinery that is to be activated.

We define a machine as an active part that describes a coherent function. It stands for proteins interacting with one another, a biochemical pathway executing a certain function as explained before in figure 4.21. An active machine is a direct result of the activation of a certain production-machinery. A machine can be another production-machinery taking care of other parts of the gene translation, or it can be of cellular behavioral nature (a behavioral machine) e.g. one that is responsible for cell migration. Machines can interact with one another through writing and reading passive signals. Since the production-machinery is also a machine it is possible to have a machine activating another machine, giving rise to another cellular behavior.

In our model we use the term primitives for something that executes a single abstract function, that is observable through the behavior of the cell e.g. movement, detection, secretion. These primitives are a combination of the interaction of one or many proteins. Every machine, production machine or machines for behavioral functions are built of these primitives; all machines have been shown to be composable from these few primitives. As described by Zubler [213], it is also clear that one can combine machines to form larger machines. All of these primitives are, as Zubler states, biologically plausible. The work of Zubler has been briefly reviewed along with the primitives he proposes in section 2.2.4. In figure 4.22 a machine is shown that is assembled from different primitives acting together to build a behavioral machine. Another more complex example of such a machine can be seen in figure 4.23. In this figure it is shown how the assembly of functionality comes about within a machine. A subnetwork of the gene regulatory network produces primitives that are then assembled into a functional behavioral machine. In figure 4.22 the subnetwork is hidden but implied as an underlying process for assembly. The machines can be recombined and reused. This recombination allows for a hierarchy of functionality and we therefore call the machines that are composed of other machines higher level machines.

The composability described abstracts away from the view of single genes and proteins but is fully compatible with the basic abstraction, allowing us to look at processes happening within cells at higher levels. The level of abstraction is chosen by the observer of the behavior and can in the extreme be broken down to single protein chemical interactions, making very complicated protein interactions humanly comprehensible.

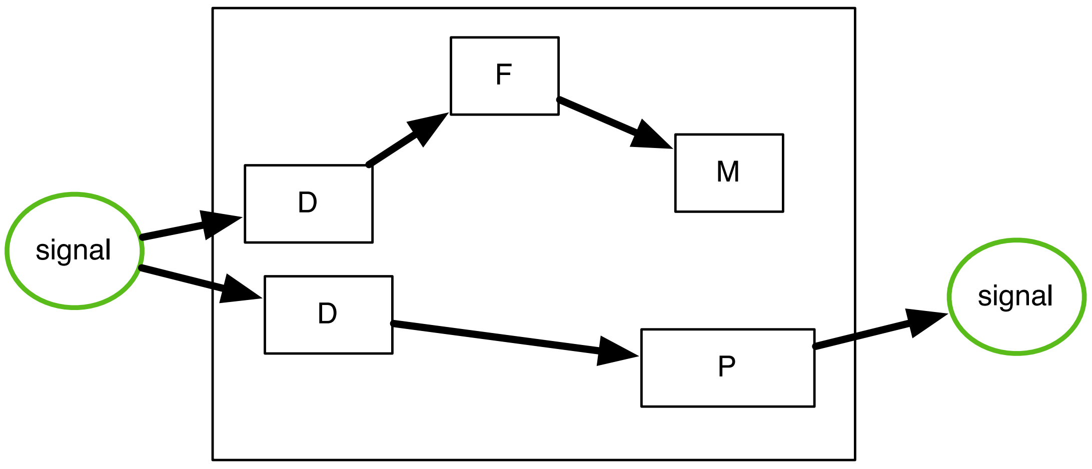


Figure 4.22. A sample machine made of some primitives, interacting with the cellular environment through passive signals The arrows stand for cooperative relationships. This diagram tries to explain the concept of assembled machines. Here we have an exemplar signal that is beeing detected by two detect primitives that feed into a filter and into a move primitive or that feed into a production machinery that then again produces another signal.

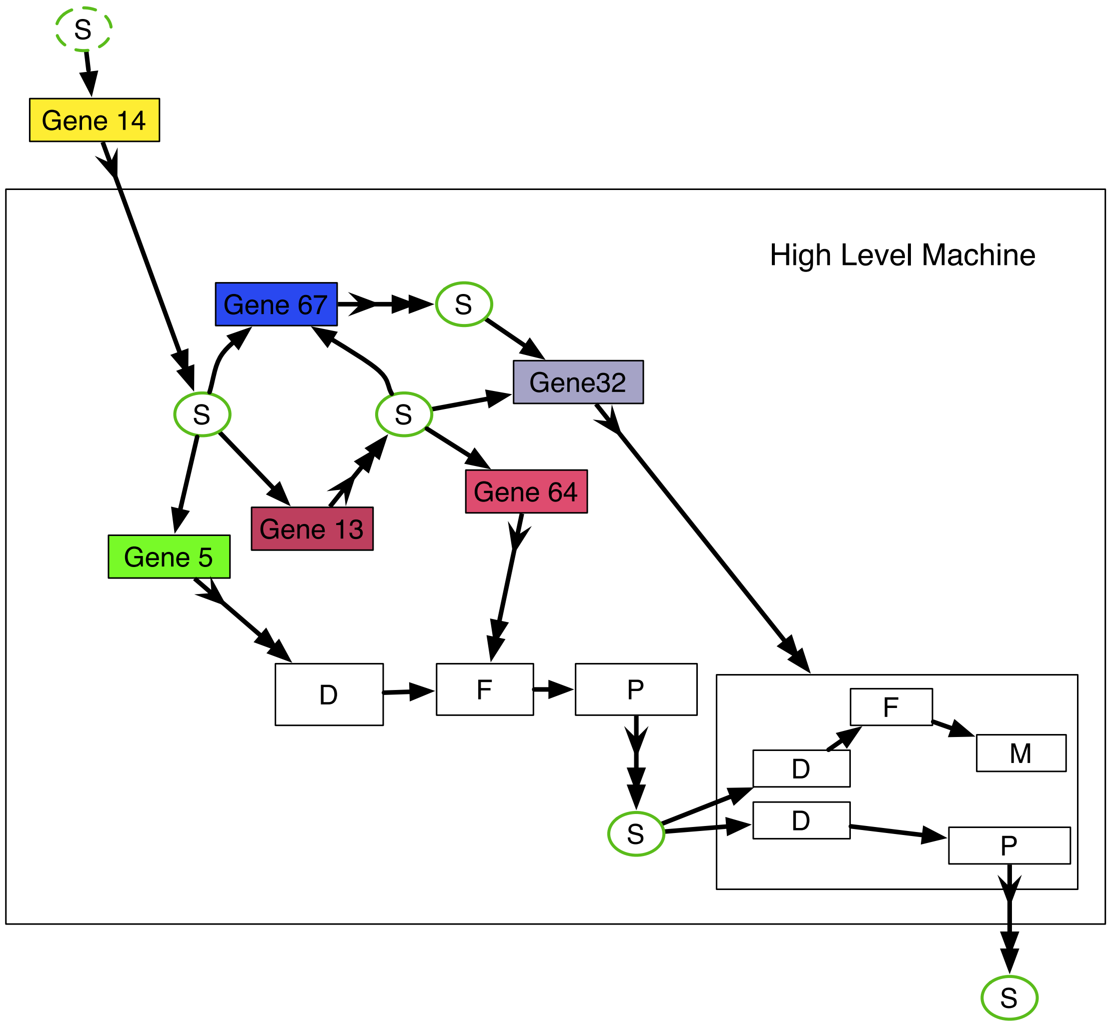


Figure 4.23. The inner workings of a high level machine. A high level machine might contain a sub-GRN in order to produce required parts from different genes that act then together to cause one behavior that is encoded by the machine. A high level machine is composed of different primitives that are assembled together, and can even be composed of other high level machines. The higher level machine shown here will produce a signal protein S. The normal arrows stands for cooperative relationships and the double-headed arrows for creational relationships.

4.3.2 Machine Interaction Causation and Communication

In the previous section it has been shown how machines can be composed of machines and primitives. Additionally, in figure 4.20 we hint at how machines can interact. Figure 4.20 shows that cells can activate behavior dependent on their state and how machines can interact in a multicellular environment. This interaction between machines needs further investigation.

There are two different modes of interactions between machines; one machine causes another one to come into existence or two already existing machines working in parallel communicate with each other. If we focus on the causal relationship between machines we see a call-tree like structure. The causal relation tree indicates an unfolding over time: a machine that causes another machine to come into existence must exist first before the second machine can come into existence. Examples of this causal relationship are given in figure 4.24 indicated by the black arrows. In this view, the machine history can be understood as a causal tree of machines. It indicates a sequence of events but not the accurate timing. The same holds true for a family tree. In an un-annotated family tree we get the sense that the children must come after the parents but there is no information given on the date of birth of each child. In the case of the causal tree the 'causing' machine must be there before the 'caused' machine but when these machines exist in time is not given. This unfolding in time is dependent in the case of the machine on the gene regulatory state and on some environmental (intracellular or extracellular) factors. Figure 4.24 shows the development of a single precursor cell and its offspring. The causal tree is shown at the bottom and the unfolding in time at the top. One can see that different machines cause one another with a certain timing. It can be seen in the example that the L1 differentiation machine **B** produces a signal as a membrane marker that will inform a layer 5 cell born later that runs the radial migration machine **C** to stop its migration and to differentiate. This communication between the two machines is inter-cellular via environmental factors (indicated with the blue arrows for communication through the environment). The machines **B**, **C** and **D** in this example are deactivated as soon as they have fulfilled their purpose. The communication between the two cells would not have been possible if the layer 1 cell would not have already been there to interact with the layer 5 cell. It is important to notice that cells build an environment for later cells and their machines in which the machines can operate on a scaffold that has been established by earlier-born cells. Hence it is important that the right cells of the right type are born with the right timing. But luckily through control mechanisms like the distributed GRN state machine and the causal tree of machines this timing is easy to achieve and to modify. Of course the interaction between two machines need not only be intercellular but can also be intracellular, with the same effects.

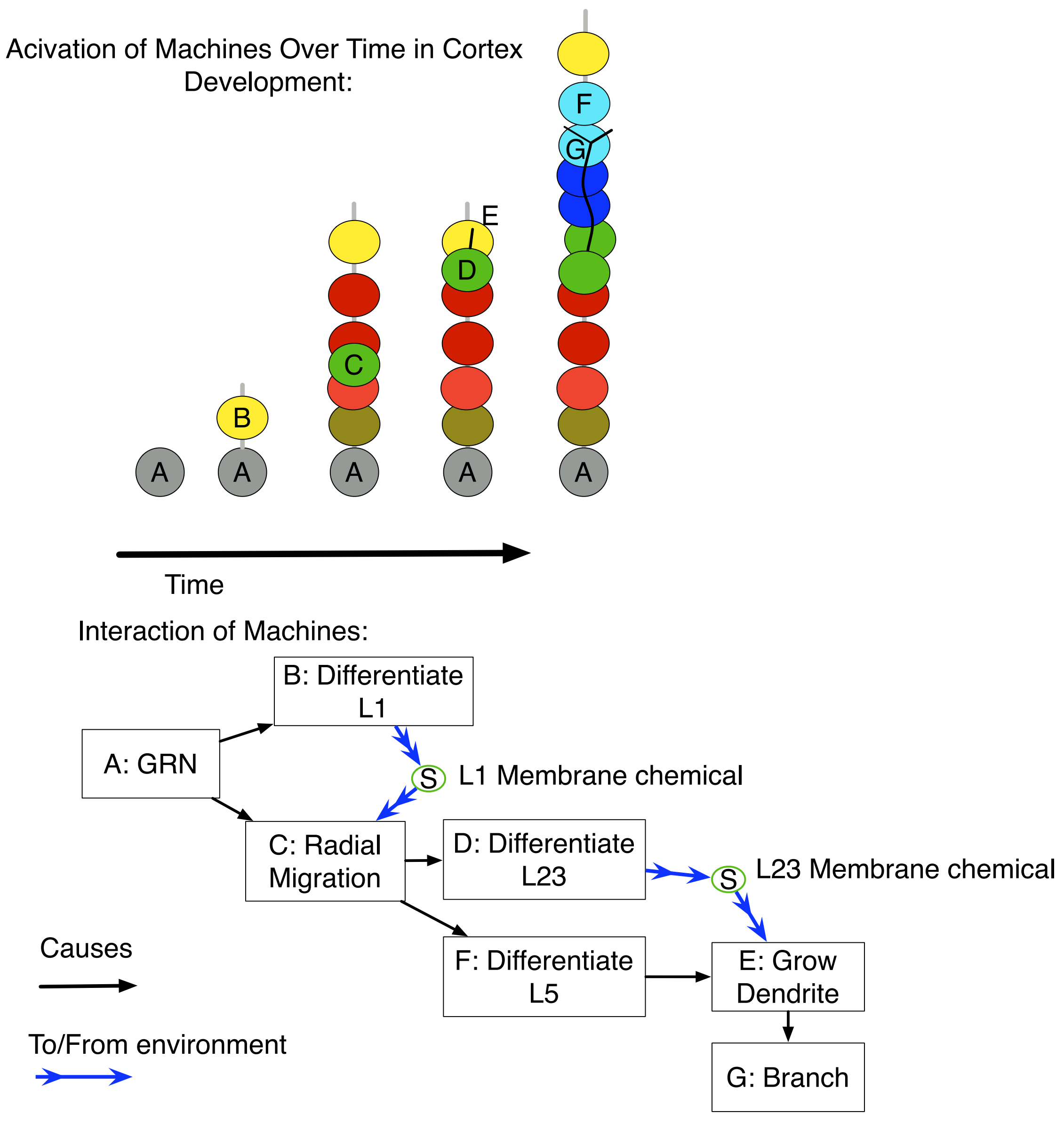


Figure 4.24. Top: the time development of the offspring of a single precursor cell (gray) where the letters inside the spheres indicate the machines responsible for the behavior of the cells. These machines are shown in the lower part in a causation (black arrows) and communication (blue arrows) diagram. This figure shows how sophisticated global behavior can be achieved through local rules encoded in these machines. The machines **B**, **C** and **D** are deactivated as soon as they have fulfilled their purpose.

It can be seen how this communication with and through the environment works from the point of view of a single cell. Machines in cells can interact with each other intracellularly, they can be produced by a production machine and the machines can interact with the environment and the surrounding cells by deploying signals. Since all the signals and machines must have a physical representation, their physical properties can be used to aid the communication process. For example cells can use a morphogen's diffusibility property to communicate with other cells. The passive signals used for communication can in addition have a physical meaning such as adhesion molecules attaching cells together [76]. In this case we would see the adhesion molecule still as a passive signal since it does not take part in the metabolism but acts as a passive physical bond. (See figure 4.25)

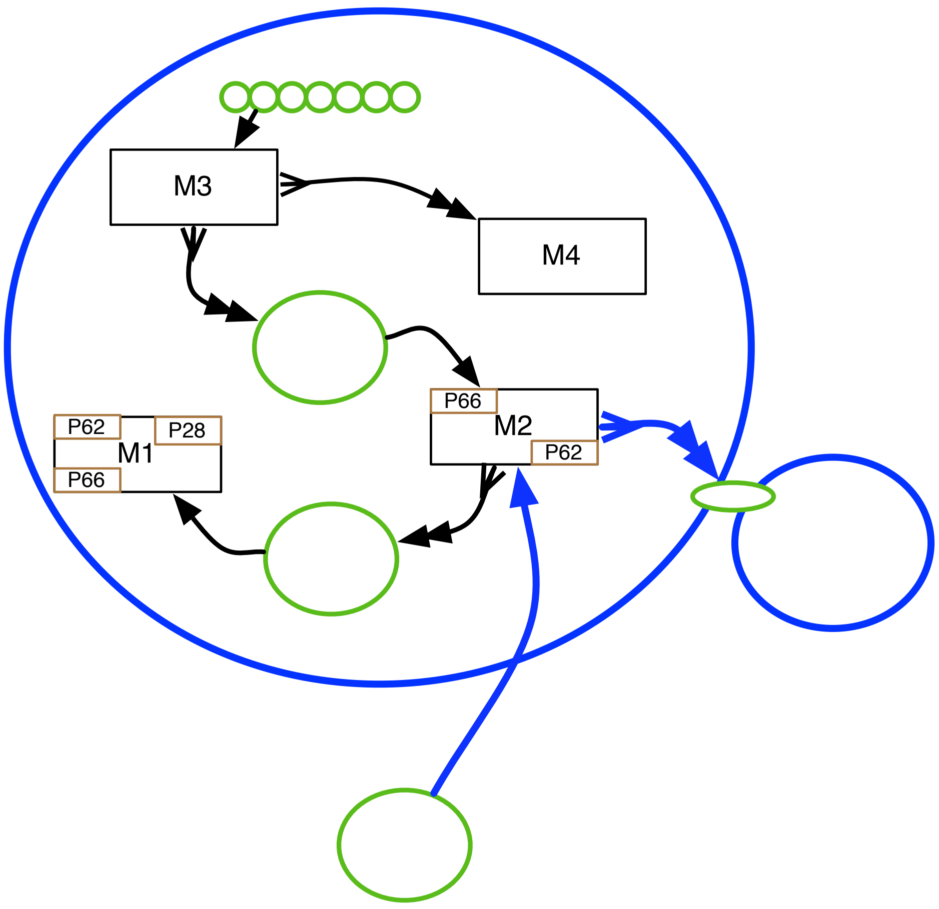


Figure 4.25. The interaction of the machines in a single cell with the environment. The cell's machines can produce other machines that then produce signals that go out into the environment. These environmental signals might then be read by other cells. The figure also shows that passive signals can be used in their physical instantiation capacity as proteins for structure, as is shown here as a physical bond (adhesion) to another cell. The normal arrows stand for cooperative relationships and the double-headed arrows for creational relationships.

We see now that ensembles of cells steered by behavioral machines timed by the distributed GRN state machine can cause highly complex behaviors as we observe in biology. Of course everything depends on the actual genetic program that defines how these machines and cells play out. It is a logical next step to see how the many emerging cells interact with each other and how the information encoded in the DNA is unfolded into the environment to build a desired structure.

4.3.3 Multicellular Organization

We have now explained how it is possible to generate different cell types at different stages of development. We have potentially a huge diversity of cells and many of each type, but now the cells must arrange themselves in the environment in order to build a functional organism that can survive in this environment. One single cell is a microenvironment, confined in a membrane. It has a position in space and occupies a part of the space. The interaction radius of a cell is bound to the cell's location and its spread in space. It can only interact locally with the environment and other cells. The need for controlled interaction leads to the second part: the coordinated behaviors of the cells. Cells must be able to arrange themselves in the environment relative to other cells to build a functional organism.

The necessity for communication between cells, the movement of cells, the shaping of cell morphology and their functional properties arises. Cells can secrete morphogens [12] into the extracellular space, or express membrane proteins [47] [208] to communicate with other cells. Now these signals must be read and there must be an active machinery in the cells that is converting these external signals coming from other cells into internal signals that influence the gene regulatory network in the cell and change their positions in the GRN landscape. Cells are therefore able to influence the behavior of their neighbouring cells (delta-notch signaling) [9] or even to broadcast a signal using extracellular morphogens. These signals though will only be read by cells that are expecting to read them (figure 4.26). They have to have the machinery for it otherwise the signal is meaningless to them. The reading machinery for external signals further allows that the cells can adjust their path in the GRN landscape according to external signals that are around. Cells can therefore remain in a waiting position until they receive external signals that will activate certain parts of the regulatory network and that might push the cells expression pattern out of a local minima in the GRN landscape into another one. Through these mechanisms cells can be synchronized, for example cell type one will remain in a waiting position until cell type two has generated a certain signal that lets cell type one go on toward its fate.

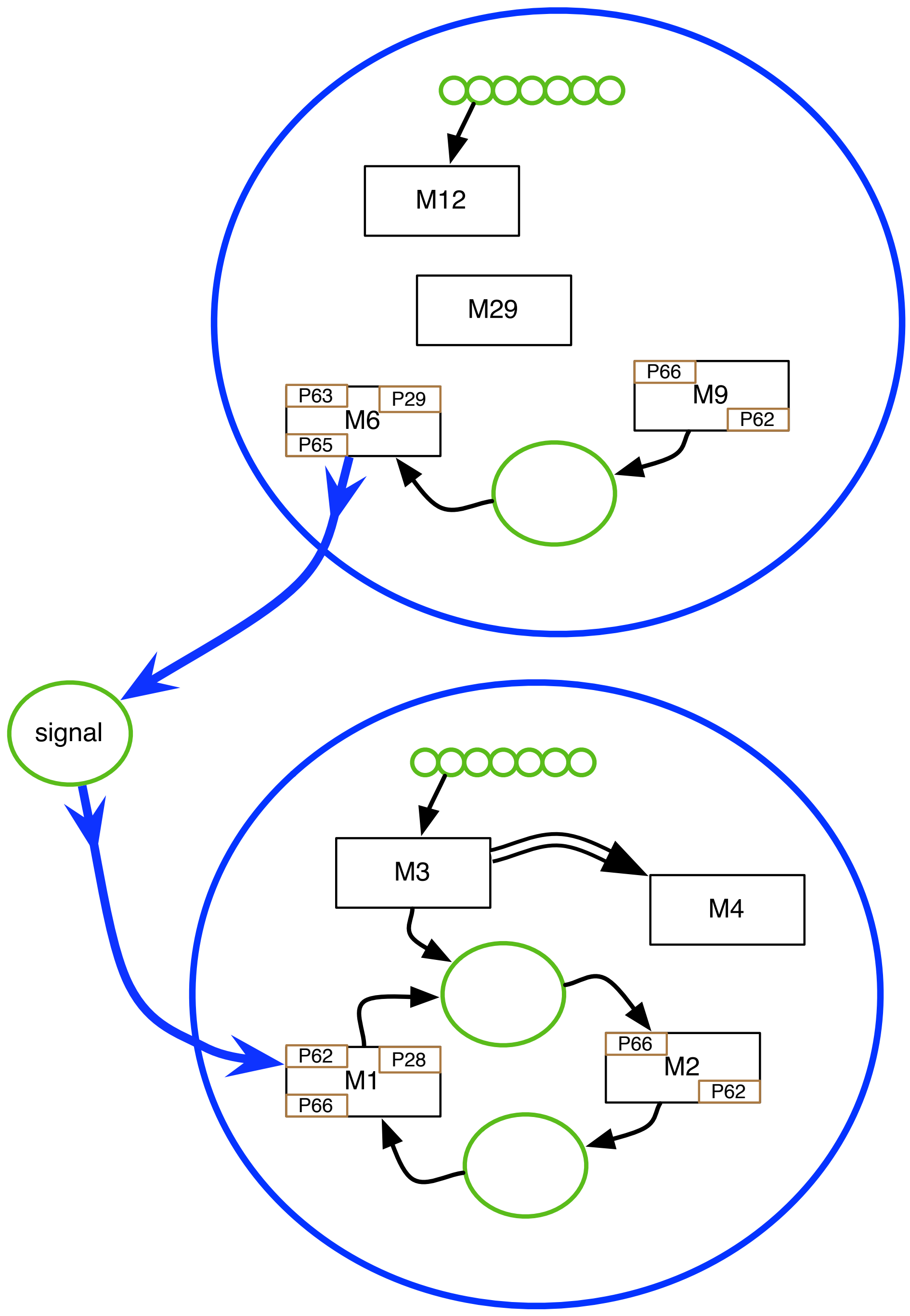


Figure 4.26. Machines in two autonomous cells exchanging information through the means of an extracellular signal in the environment. Both the sending mechanism and the receiving mechanism have to be actively present as machines in order to exchange information. The single arrows $\longrightarrow$ stand for cooperative relationships and the double headed arrow $\Longrightarrow$ for a creational relationship.

The same mechanism can be used by cells to label the environment, where other cells that migrate through the environment can then integrate all the information and react to the signals they need (e.g. permissive cues, chemoattraction, chemorepulsion, and start/stop signals are possible and used [133] [156] [201]). Passive physical (non biological) processes are exploited by the cells to perform self-construction. These processes include diffusion for long range communication or forces that do not permit cells be in the same place at the same time.

The position of the cell in the abstract GRN landscape defines which reading or writing mechanisms are active in which time point in a cell. The gene regulatory network navigates through the state space but also integrates information the cell picks up from the environment. It is important to note that the position of the cell in the environment and the position of the cell in the abstract GRN landscape have two completely different meanings. The position in the GRN landscape gives information about the state the cell is in, whereas the position in the environment describes the physical location of the cell in relation to 3D space.

With the mechanisms explained in this section it is possible to self-construct complex looking structures without a global controller.

4.3.4 Complexity and Information Flow

Complexity is a difficult issue for human perception; we can create systems perceived as having complex behavior with relatively simple programs. For example, with simple local rules we can generate many different patterns with local computation, such as the famous Turing patterns [188] or fractal patterns such as the Mandelbrot set [115]. Although these patterns are perceived to be very complex, the rules to generate them are relatively simple. We must be careful not to confuse the human perceived complexity with the measurable complexity of a system that is defined by Kolmogorov [105] as the length of the shortest program to create a certain pattern using the universal Turing machine. Roth already described this issue of complexity in his thesis [159]. The Kolmogorov complexity [105] is a measure of complexity for an object, typically an object represented as a string. This complexity is described by the minimal length of a program to generate the string. Or in other words how short the maximally compressed description of an object can become.

In our abstraction the flow of information is critical for the construction of the final structure. The idea is to extract information from various sources and to use that information to build a structure in the environment. In order to understand the flow of information it is important to identify the sources of information that the system has at its disposal.

In our case the closed system would be the environment in which the organism develops. The sources of information are therefore enclosed in this environment. We identify three basic sources of information. One source is the string of genome, that describes the program to be executed during the development (DNA). Another other source of information are all the signals that are potentially read by the cell ($E_{\text{inf}}(t)$) at a given time t. And the last would be the rules of the environment such as gravity or force interactions between objects (E_{rules}). If we investigate the complexity of the program versus the complexity of the final structure in a closed system we can look at it under the Kolmogorov's constraints [105]. The application of the Kolmogorov complexity constraints implies that structures in our system cannot be of higher complexity than the shortest program that generates them. If we keep the initial conditions E_{inf} in our environment and the rules of our environment E_{rules} constant we can therefore expect that the complexity of the final structure is not more complex than the program that describes it. In our case this program is the DNA. Now the concept of Kolmogorov complexity holds true. If we take a global view of the system the information content will not change globally, but what will change is how the information is distributed in the system. In our case the program of the DNA gradually unfolds itself in the environment and builds a structure.

On the other hand, if we change our perspective and do not look at the system as a global observer but from the perspective of a single cell, the information content of the environment can change. If we follow the initial cell's course of development we can observe the following changes in the environment. At the very beginning of development, the initial cell contains one copy of the DNA code. At this point the distribution of information in the the environment is obvious (figure 4.27). The cell A is in state a_0 and has all the information available in the environment:

$$\Pr(a_0|I_{\text{DNA}}, E_{\text{rules}}, E_{\text{inf}}(t_0))$$

This means that the state a_0 of cell A depends on the environmental rules E_{rules}, on the initial conditions $E_{\text{inf}}(t_0)$ and the information I_{DNA} available in the DNA (figure 4.27). During the development of the system the initial cell A will split into two and create cells B and C in the states b_0 and c_0 respectively (see figure 4.28). At the moment of cell-division the question arises what information each cell has available for changing to the subsequent states b_i and c_i? (Figure 4.29.) Compared to the starting point of the development the information begins to be distributed in the environmental space, therefore no single cell has the complete information in the system. This makes the environment a potential source for information to be integrated into each cell. What state b_i the cell B will choose next is dependent on the environment and the DNA that it has received from cell A. From the viewpoint of cell B, the information from the DNA stays constant but the information in the environment changes over time. This gives the choices of the next state for cell B:

$$\Pr(b_i | I_{\text{DNA}}, E_{\text{rules}}, E_{\text{inf}}(t), b_0)$$

where b_0 is the previous state of the cell and b_i the future state of the cell. From the viewpoint of cell B we consider cell C and its state c_0 part of the the environmental information $E_{\text{inf}}(t)$.

In the case of the cell C the view point is different. The cell C has to decide which state c_i to take next. As for the next state b_i of cell B, the next state c_i of cell C is dependent on the environment and the DNA that it has received from cell A. But for cell C, cell B and its state b_0 are part of the environmental information $E_{\text{inf}}(t)$. The following formula describes the choice of cell C for its next state c_i, where c_0 is the previous state of the cell and c_i the future state of the cell.

$$\Pr(c_i | I_{\text{DNA}}, E_{\text{rules}}, E_{\text{inf}}(t), c_0)$$

The information content of the environment will be determined as a function of time and the cells acting in it, $E_{\text{inf}}(t) = f(E_{\text{rules}}, \text{CELLS}, E_{\text{inf}}(t-1))$ where $A, B, C \in \text{CELLS}$, t is time, and each of the cells has a state. Looking at individual cells emphasizes that depending on the cell, the view of the information sources changes and gives the cell the opportunity to learn about its environment. This means that from the cell's point of view the environment and its information content apparently changes, over time there might be more and more cues providing information to this single cell.

But for the total system the sources of information have not changed. It is still the $E_{\text{inf}}(t)$, E_{rules} and the DNA that act as sources of information. We can also see that the information that is coming from the DNA and that describes the program that the cells follow, transforms the structure of the environment. This process is transferring information locked away in the DNA into the environment manifesting itself in global structure in the environment.

If we think about a system that is not closed, in which the environment is influenced by many processes that have initially nothing to do with the development of the organism, we can see that the organism can potentially make use of the information available and integrate it at any developmental step into the construction, but still it would only react to signals it expects (e.g. rods and cones [206] that are receptive to light). Thus the information that can be potentially integrated grows with the potential for absorbing this information. This potential for information integration grows on the other hand with the number of cells in the system. Further it implies that the information for the construction of an organism comes more and more from the environment rather than from the DNA during its development. This is because the available information from the DNA is constant and millions of orders of magnitude less than the information available in a natural environment. Hence the organism can exceed the

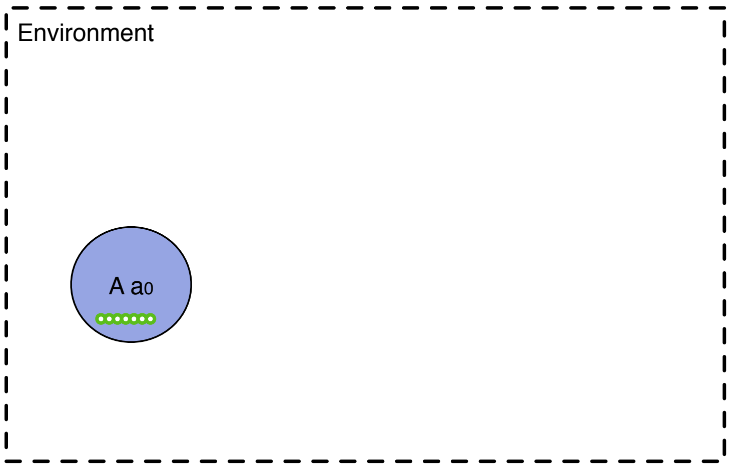


Figure 4.27. The initial situation of the environment and the initial cell in its initial state. The small green dots in the cell represent the DNA. The initial cell contains all the information that is then to be expanded into an organism. The cell will start to divide and copy its DNA to it's offspring cells. The information in the DNA will be transformed into a structure built by the cells.

complexity of its DNA. This was not the possible in the closed system case explained before. This influx of external environmental information has two implications. First, the system can exploit the information in the environment for its construction. And second, if we have a system that is constantly reshaping its structure dependent on the environment it is not finished with constructing, but under constant reconstruction and it is becoming potentially more and more complex in its structure. An example of this behavior would be a constructed brain that rearranges or reconstructs itself and the connectivity within it depending on the information available from the environment, where the DNA plays only an assisting role in providing the machinery to extract this information (synaptic learning [119]). In this case the processes that define how to react to the environment are still encoded in the DNA. The DNA sets up and maintains a scaffold that is able to integrate information coming from a vast environment. This implies that the scaffold changes (integrates information) from the surrounding environment and changes it structure accordingly; it adapts to and/or learns from the environment. Even with such a learning process in place the information theoretical ground rules of Kolmogorov complexity are not violated because the information that flows into the developmental process is provided by a vast natural environment. This means that the developing organism can no longer be seen as a closed system but the overall system must be seen as the organism and the huge environment it is in, including all the information present in this huge environment.

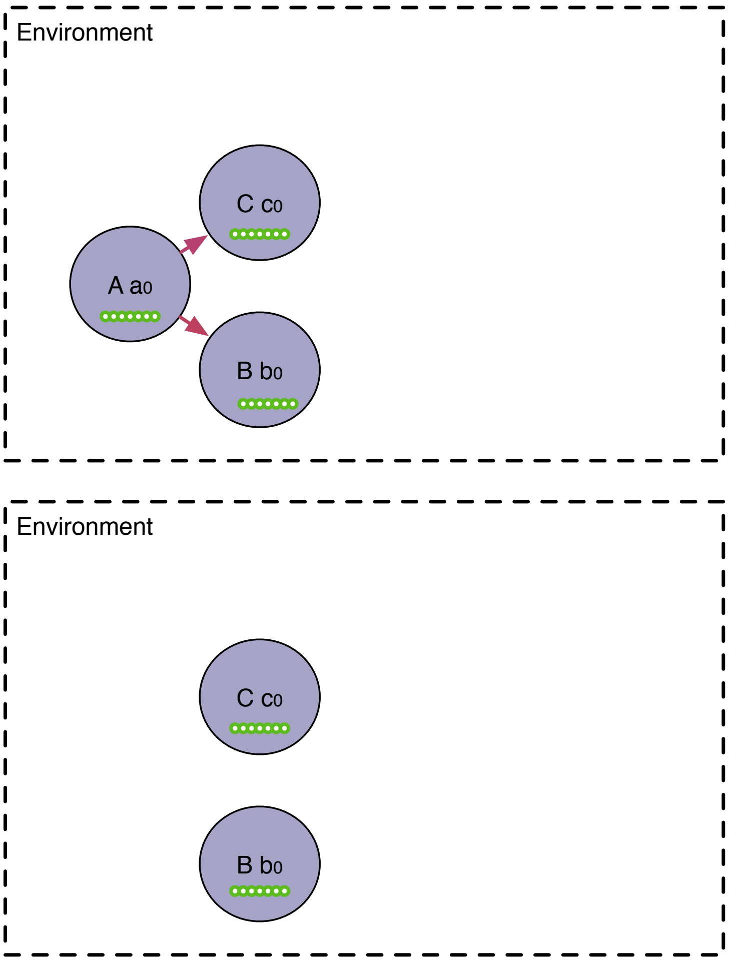


Figure 4.28. The situation in which two cells are introduced. Top: the moment when A splits into B and C, and bottom: after splitting, each cell carries the information of the DNA, but each cell is in a different state. The new offspring cells B and C contain the same DNA as the initial cell A had. From the viewpoint of the complete system the information content is still the same. But the individual cells do have different view points of the system and do not have all the systems information directly available.

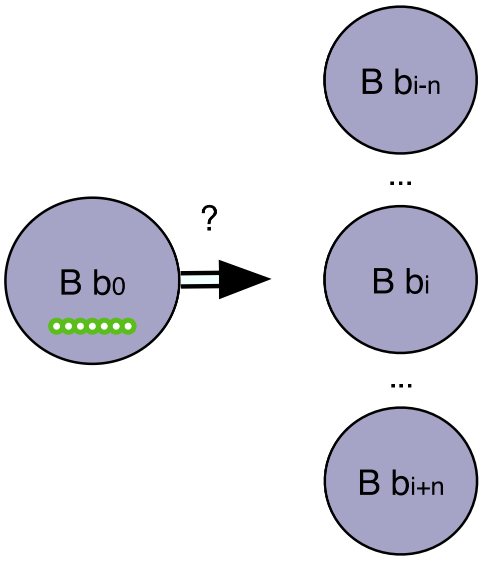


Figure 4.29. The question of which state a cell shall take next is dependent on the environment and on its previous state. From a global view point the information content of the system does not change on cell B choosing a state. Since single cells can not take the global perspective the cells surrounding B might still rely on the information that the cell B and its state provide.

4.4 Discussion

In this chapter we have presented a differentiated view of biological development. We present a model of the biological system that ranges from a specific gene to the behavior of ensembles of cells. We do not make assumptions about single genes or single cell types but about how in general processes in a cell can lead to cellular behavior.

We began by describing a single gene and how it is transforms into proteins that have an active metabolizing functional aspect or a passive informational aspect. Then we showed how genes can interact and build in ensembles to build a gene regulatory network (GRN) based on the work of Pfister [146], with genes inhibiting and exciting each other. This network spans the whole genome and can be seen as a function with a high dimensionality where each dimension corresponds to the expression level of one gene. A specific genetic expression pattern forms one point in this function, and the function defines through its gradient how over time the expression will change. We show how cells let the activities in the network evolve and therefore navigate the functional landscape that we call the GRN landscape, whereby the points in the landscape need not only be seen as an expression pattern but also as the state of the cell. We showed how the GRN landscape of gene expressions can be interpreted as a state machine where the gradient defines the transitions, and that in this state space confidence interval-like regions can be defined that represent cell types. For the example of an initial cell of an organism, the path through this state space was shown. But also the effects of replication are shown, namely that the cells jump in the GRN landscape, since at division the cell has to distribute the available expression pattern. This replication process is a key point for how millions of different cells can be created with thousands of different cell types in a coordinated way through symmetrical and asymmetrical division. We explain how a specific function in a cell can arise through assemblies of proteins that work together and through the reuse of proteins for different purposes. These higher level functionalities that influence the cell's behavior are named here *machines*. We show that it follows naturally from looking at the landscape how such machines can be activated by the different cells, allowing for many different cell types to use the same types of machines or express the same proteins for different kinds of machines.

Considering Zubler's work [213] on G-Code, we show how G-Code fits into the notions of assembled machines that we describe in our abstraction. This allows cells to have explicit behavior such as movement or secretion of morphogens. The coordination of cells is subsequently addressed and it is shown how cells can organize themselves to build a whole organism. We support the viewpoint that studying single genes will not increase the understanding of the cellular behaviour because there is always a large pathway involved composed of multiple genes that can participate in potentially multiple behaviours. The expressed proteins would have to be investigated in their participating behavioural machine. The complete machine is the important aspect to investigate. Looking at the single gene is like looking at each type of nut and bolt in an airplane to see what it does.

Having shown the whole process, we analyze the flow of information, and show that the information is partly coming from the DNA and partly from the environment. The environment that defines the physics can potentially contain some preexisting morphogens and it provides a space for the assembly of the organism. That means that all the information of the whole developmental process is present from the beginning but is just transferred, or computed, into structure. The system never violates the principles of Kolmogorov complexity since the entire expansion of the organism is computed using the DNA and the rules of the environment. This means that the final organism's information content can at least be compressed back to the DNA code and the rules of the environment. The complexity of the whole organism cannot be greater than the complexity of the DNA and the information supplied to the system by the environment. The environment includes the starting conditions, physical rules and the information presented to the organism during development. Roth [159] observes the complexity of the organism and realizes correctly that complexity has a high perceptual component, if not analyzed under the strict scope of Kolmogorov complexity. If humans perceive something as complex it does not mean it requires a complex description to generate it. One popular example is the fractal Mandelbrot set [115] that looks extremely complex but can be defined with just a few lines of code, this would not qualify as immensely complex under the scope of Kolmogorov complexity.

Waddington's epigenetic landscape [196] has gained a lot of attention. Waddington shows a cell that develops and figuratively follows a downhill path in a landscape. This landscape is shaped by underlying genes that are analogous to poles with ropes attached that pull on the landscape, sometimes more than one gene pulls on one rope and sometimes one gene pulls on multiple ropes. The landscape is imaginary, defined by the genes to influence the cells' paths during development. The ropes and poles can be interpreted as the functional dependance of the landscape on these ropes and therefore on the interaction of the genes as a gene network. It has been a very influential work and must therefore be compared to our view of the genetic network, in order to avoid confusion. Waddington's epigenetic landscape bears a certain resemblance to the GRN landscape we present here. A GRN landscape is in a high dimensional space, and what we show is just a projection of this space. Waddington also imagined a high dimensional space but instead of projections of this space he works with the analogy of the two dimensional landscape with the underlying genes that pull on the landscape. The GRN landscape is completely fixed, defined by the genome or gene regulatory network of the organism and does not change over the course of development. The GRN landscape would only change if the genome would. It is not entirely clear whether Waddington imagines the epigenetic landscape to change during development or whether it stays fixed. Waddington speaks of cells navigating the epigenetic-landscape but never specifically mentions how the cells decide which route to take; he stops his explanations there. We take it from there and begin to explain how cells can navigate the landscape though divisions and therefore jumps between regions in space. We additionally explain how to get from these states in the GRN landscape to functional behaving cells and from these cells to multicellular organization.

In terms of the validity of our present model it is clear that it does not describe the biology of development in every detail, but it is very helpful to think in these terms in order to get a grasp at the problem of understanding biological development. The model presented will break down at some level of detail but it helps to a certain extent to describe observable processes. One should view our work as a first attempt to bring a general model of development to biology. With this theory we want to provide a tool for how to think about biological development, in the spirit of Newton's model (Newton's laws of motion [137]) in physics and the Bohr model (Rutherford-Bohr model [27]) in chemistry: both will break down at some point but are often sufficient for practical problems.

Chapter 5

Developmental Programming Paradigm

In this chapter it is shown how the theory of biological development from chapter 4 is directly relevant for a new style of programming. The main goal of the thesis is to derive a potential engineering technique from biology to be able to program systems that have the ability to self-construct as we observe it in biology. First it is explained why the need for a new paradigm arises. Then the relevance of biological development is explained in the context of a programming paradigm — how the biology contains directly applicable lessons for the programming of self-constructing systems. Having explained the paradigm, it is compared with functional and object-oriented programming that are currently state of the art to indicate the differences.

5.1 Standard Paradigms are Inadequate to Describe Highly Parallel Processes

In the middle of the last century the first programmable computers were created [25]. In the beginning the programs for these early computers were written on punching cards that were fed one after the other to the computer [91]. It was extremely time consuming to write programs and extremely difficult to find errors, only one hole too few or too many would destroy the result of the intended calculation. With the invention of assembler code it became possible to tell the computer with human comprehensible commands what it should do. Assembler improved the usability for the programmer compared to earlier methods, but it is still hard to write programs that are more than a few hundred lines. Assembler programming can lead to confusing jump sequences that are very hard to keep track of (the famous spaghetti code) [53]. The structured programming paradigm [26] has a remedy for exactly this problem: it only allows for jumps in the flow of control where no unconditional jump statements are permitted but only control loops and if-then-else constructs, resulting in much more understandable code without hampering the computational power of the code. Control loops were implemented in programming languages such as Algol [14] and Pascal [202]. In programming it often occurs that the same piece of code has to be executed at different locations. If the code is just copied each time the maintenance of the code becomes a nightmare. The introduction of the procedure call in the procedural programming paradigm [92], in which blocks of code are associated in a logical module (the subroutine or procedure) that is then invoked when needed increased maintainability greatly. Typical programming languages of this type are C or BASIC. This paradigm used variables and data structures as its abstractions, allowing for collections of associated values for accessing the data and, the routines would operate on these collections. [203] Even though many successful programs are written in these languages, it is still a hard task to create large programs.

Built on the idea of mathematical functions and the lambda calculus [41] another programming paradigm surfaced: functional programming [89]. This paradigm is based on functions that are allowed to take arguments, run using those arguments, and return calculated values. There is no other way to pass values between different parts of the program than to use arguments and return values. This scheme of value passing leads to a system that is very nice to make proofs about because the flow of data is very strict and clear. But as a programming paradigm for the masses it never had a huge success: most programs are still written in object-oriented languages, we suspect because humans tend to get lost in recursions. Programming languages such as Haskell [90] or R [64] are instances of the functional programming paradigm.

Object-oriented programming, in which the central point of programming is the object, integrates the idea of the data structures very well. Each object has associated data and functions, allowing the programmer to use the objects' functions without knowing exactly how they are implemented. The object-oriented view led to a revolution in computer science that allowed for much bigger programs that were much easier to maintain, to be developed in parallel by multiple programmers and to be reused [108]. The success of this paradigm is not that it is easier for a computer to work with such code, but that it is a very natural way to think about a problem for a human. Collections of objects and what they can do is very much how we see the world, such as cars that can drive, doors that can open and close, or many other objects with which we associate functions and properties. It is very human to reason about a problem with objects. The object-oriented programming paradigm is implemented in languages that are the current state of the art, such as Java, C++ and C# [61]. There are many more paradigms that shed light on different aspects of thinking about a problem but the most influential ones have been covered.

At the beginning of this century the first multi-core / parallel processors started to become commonly available [198]. Now it was possible to do the work of one program with two processors in theory in half the time. But it turned out to be not so simple. Even though all the major programming languages are able to make use of multi-processor environments and have control structures to steer these, it is not a trivial task to parallelize a program to efficiently make use of all the available computing power. It is hard for humans to reason about parallel programming in the world of object-oriented programming. It requires a lot of training and experience to be able to write efficient parallel code, and often there is just one task or very few tasks that are parallelized where all the parallel processes do the same thing at the same time. A good paradigm that captures parallel programming efficiently is still missing.

Currently many interesting developments are being made in different fields of science. In biology the first artificial cells are being made [71] by creating artificial genomes for single cell organisms. In nano-technology we are getting closer and closer to creating machines at the nano-scale that are execute the tasks they are constructed for. In computer science we are getting more and more cores in one processor. If any time soon now it will be possible that any of these technologies begins to create self-replicating behavior we are at the beginning of a technological revolution. Multicellular, agent systems will be possible that develop out of an initially placed cell or agent, that have the potential to construct organisms that are made of active partial automatons. Already now we see the first systems that are potentially able to have replication behavior, if not in the physical sense then in the informational sense. Such systems include grid computers, clouds, chips such as the SpiNNaker chip [102] or sensor networks where programs are entered at one point and have to be spread through the whole system through the means of information replication. Let's say we have a new system that is able to exhibit self-replicating behavior. How would we think about programming such a system? How would we go about it and let it construct complex 'organisms' out of an initial agent or cell? Biology is an instance of such a system that already has self-replicating parts. Our bodies are based on billions of cells that have been constructed out of one single individual cell at the beginning of our biological development. The processes of construction and maintenance are heavily parallel and performs many completely different tasks at once. Programming such a system in an object-oriented way would clearly be unsuited to the problem. We need a better way to reason about programs that are executed by such systems.

We found a way to abstract biologically developing systems and implemented G-Code for simulated simplified versions of them. But if we were to move to another system we would have to re-implement a language equivalent to G-Code [213]. Our goal is to capture the basic principles of how such a language is implemented. What would an engineer have to do in order to obtain a system that is as easy to code in as G-Code is. I want to define a 'developmental programming' paradigm that captures the basic ideas of G-Code and generalizes it for other systems. G-Code is to the ‘developmental programming’ paradigm as C++ is to object-oriented programming. An instance of a language following the principles of the paradigm. ’Developmental programing’ should be considered as a style of thinking about a problem.

Object-oriented programming has become very influential in computer science for software development, because it allows us to abstract physical world objects into a computer executable model that is humanly comprehensible even if the number of lines of code in the system increases drastically. It allows for a structuring and encapsulation of data that is humanly understandable. The objects in a program in this paradigm work hand in hand to build very complex systems. Object-oriented programming supports standard software methods and patterns that help engineers to design correct code. But as soon as we have systems that are massively parallel in a non-trivial way, object-oriented programming breaks down. It might be because the object abstraction is no longer the right abstraction for thinking about the problem the software is trying to solve. For systems that undergo self-construction, we can think about any program in the ‘developmental programming’ way. Where we have the object, in our system we will use the notion of machine that acts through a cell on an environment. The machine executes commands through a container (cell) in the environment with only local awareness. In these systems there is no single global controller. Languages following the ‘developmental programming’ paradigm are inherently parallel. These languages would be not require active consideration of thread synchronization.

The ‘developmental programming’ paradigm will become very influential as soon as a programmable cheap self-replicating systems exist, such as nano-robots or programmable synthetic cells. Or even for configuration and setup of grid computers or other computational platforms that have to be set up and configured (organization of computational resources). How to program nano-robots to perform sophisticated behaviors while having a limited amount of memory available to execute a program is still an open question. Engineers will have to implement algorithms for these systems that are massively parallel and hard to analyze if not seen from the viewpoint of the ‘developmental programming’ paradigm.

We show in this chapter that if a system implements a language according to the ‘developmental programming’ paradigm it will be universal Turing complete, allowing a system that implements such a language to create the ‘Any Shape’, meaning that for each computable shape there exists a program in this language that will generate it. And that these shapes can be constructed in reasonable time. What are the necessary operations such a system must support in order to be able to exhibit self-construction? We will explain the Developmental Programming paradigm using the example of the G-Code language.

5.2 Biological Implications on a Programming Paradigm for Self-Construction

In the chapter about the developmental theory, chapter 4, we have discussed how biology can create a complete organism from one cell through the process of development which is a self-constructing process. It was explained how single genes interacting in networks can direct cellular behavior, how these behaving cells can interact with one another, and where the information for this process comes from.
Now what general principle can be extracted from this biological theory?

There are prerequisites for a system that can self-construct. The first is to define the substrate that consists of an environment and one or more containers. In biology the environment is given by the rules of nature i.e. physics and chemistry. In our simulation of the developmental process, this environment is given by Cx3Dp defining the physical interaction of cells and the extracellular diffusion. The container in biology is the cell. Proteins mainly define the behavior of the cell they are produced in, not other ones. In our simulation of cortical development there are two types of containers, the soma and the neurites of the cell. The containers of the system are the components the self-constructed object is made of. Another prerequisite is that the instruction code for the system needs to be available in all the containers, in a textual form. In biology the textual instruction code would correspond to the DNA and in the simulation the instruction code would be our artificial DNA-code defining the G-Machines. This code is selectively translated on demand by each container individually. Typically, not all the code is used at once in all containers. The container needs to be capable of maintaining a status over time; the container needs to be stateful. In biology this state is given by the expression pattern of the genes. In the simulation this process is mimicked with intracellular substances and active machines. This statefullness of the container leads to the need to define state transitions. The state transitions are defined in biology by the gene regulatory network that defines the state transitions that the cell can undergo. In the simulation these state transitions are mimicked by an artificial gene regulatory network and by the successive activation from one machine to the next. The ability to have a state is only useful if the behavior of the container can be changed. Therefore the container needs certain abilities. For example it must be able to move, to interact with the environment, and to read and write information. In biology these abilities would correspond to the abilities a cell has i.e. migration, and the secretion and detection of chemicals in the extracellular or intracellular environment.

Besides these abilities the container needs to be able to run instructional machines. These machines are the actual agents that define the behavior of the container. Multiple agents can be active at a time in one container. As an example in biology, cells have many different proteins active at once that define the behavior of the cell. The agent in biology can be seen as one biochemical pathway or protein complex that leads to a certain behavior e.g. migration or apoptosis. In our simulation these agents are called G-Machines, defining the actions of the simulated cells and growth cones. These instructional machines are fully encoded in the textual form in the artificial DNA and can only be produced inside the containers. The instructional machines on the other hand cannot leave the container and can only make use of the abilities given by the container. A container can only be controlled by its instructional machines and cannot be directly controlled from the outside. The relationship between instructional machines and containers can be seen as equivalent to the relationship between software and hardware. Software can only make use of the abilities the hardware provides. The interaction of software with the environment is only possible through the hardware. In biology this distinction is not as sharp, since proteins are used also as structural entities for giving the cell shape and structure. Viewing proteins as signalling, structural or as behavior-defining entities leads back to the dual view of signaling proteins versus active proteins from chapter 4.

A system that shall be capable of self-construction must therefore define what its environment, container and machines are. The container has to have a state and the machines are activated depending on that state. The abilities of a container must be made available to the machines in oder to execute the behavior. All the code for the machines must be available in all containers in textual from. This interplay of environment, container and instruction machine is captured in figure 5.1.

5.3 The Machine Concept

Machines are autonomous agents that steer the behavior of the container. This machines exist only inside containers and do never leave them. The machines are compositions of the containers abilities/primitives in cascade with one another. Machines can contain functionalities that are defined by other machines thus creating a hierarchy of composable machines. Machines are defined in the code in textual form and can be translated into an active form as agents that influence the behavior of the container. In the textual form it is defined how the available primitives or abilities of a container are combined in order to define the behavior that the machine should elicit on the container. These machines correspond to the biological machineries described in chapter 4 that represent from single proteins up to complete biochemical pathways. In G-Code these machines are called G-Machines.

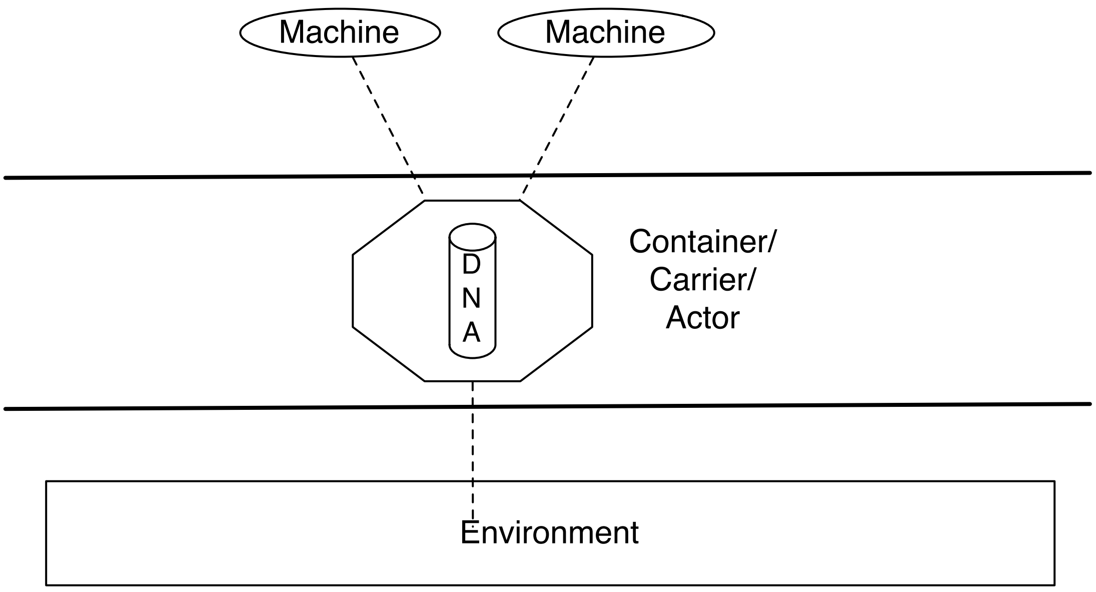


Figure 5.1. An abstract view of a stereotypical system that is capable of self-construction. The container (octagon, center) acts in the environment (bottom) steered by the behavioral machines (top)

5.4 Abilities of the Container

We have described how the interaction between environment, container and machine occurs. We have shown how we got to this abstraction by the analysis of the biological theory. It is described that these primitives are composed into machines that they reside inside the container to steer its behavior. But what are these abilities of the container exactly? What must it be capable of? These abilities corresponds strongly to the primitives found in G-Code [213]. However, G-Code is only valid for Cx3Dp but there must be a way to define categories of primitives/Abilities of the container that allows for the directing the container behavior via Machines. We will now give a list of these categories and explain why these categories of abilities are important for a functioning system.

5.4.1 The Central Role of Replication

We described replication in biological systems in chapter 4. A replication event generates a copy of the source cell and brings the new cell and the old cell into a specific state. Both cells are active afterwards and contribute computationally to the system.

In the more abstract case, if we do not allow replication in our system, theoretically we can still generate a structure in the environment, but the whole structure is passive except for one active container, the initial container, that computes the whole structure. The drawback of a system that has only one constructor is that all of the computation has to be done from this single constructor.

Let's now imagine this single constructor container has the ability to generate container-like structures that are not active and therefore cannot contribute further to the generation of the structure once they are created. These pseudo-containers have all the physical parameters of a real container but no active components. Placing one of these pseudo-containers from the active constructor container would be like writing to the environment.

If we imagine this we see that one single constructor container is able to generate the same structure as a system that can replicate the constructor since it is able to move, write and read in the environment. Now in biology a single cell takes about eight hours to replicate in the fastest case, which would mean that creating one of these pseudo-cells in our thought experiment would take the same amount of time. If this single constructing cell now had to generate a whole human body with its trillions of cells we would end up with a time for construction longer than the lifetime of the universe so far. Hence in a reasonable time it is not possible for a single constructor container to generate all the containers of a single organism. If we now allow replication of the constructor container, this means that a replication process really does create another active constructor, making it in theory possible to cut the construction time in half after each doubling of constructors. This is of course not counting the mutually exclusive construction parts that can only be done in sequence, one after the other. Following this argument, if we can double the constructors we will get an exponential speed up in the execution of the construction. Allowing systems of huge dimensions to be constructed in reasonable time. In theory we can construct a system that has twice the size of another system in only one more step. That further means the construction will not be slowed down by the availability of constructors but rather by the availability of the resources that are needed for the computation of the individual constructors. The ability to replicate is important for being able to construct faster, and theoretically exponentially faster. In figure 5.2 the execution of a 'replicate' primitive is shown in a stereotypical environment that is be able to exhibit self-constructing behavior. That replication is one of the most important features of a paradigm concerning self-construction can be seen in the attention that Roth's thesis [159] devotes to it. He gives a nice summary of the work of von Neumann who invented a theoretical universal self-replicating machine with 29 states, giving essentially a description of what a container has to implement in order to be able to replicate [194] [31].

It is important to note that multiple cell types in biology are only possible because the new-born cell's state can be changed through asymmetrical division. If asymmetrical division was

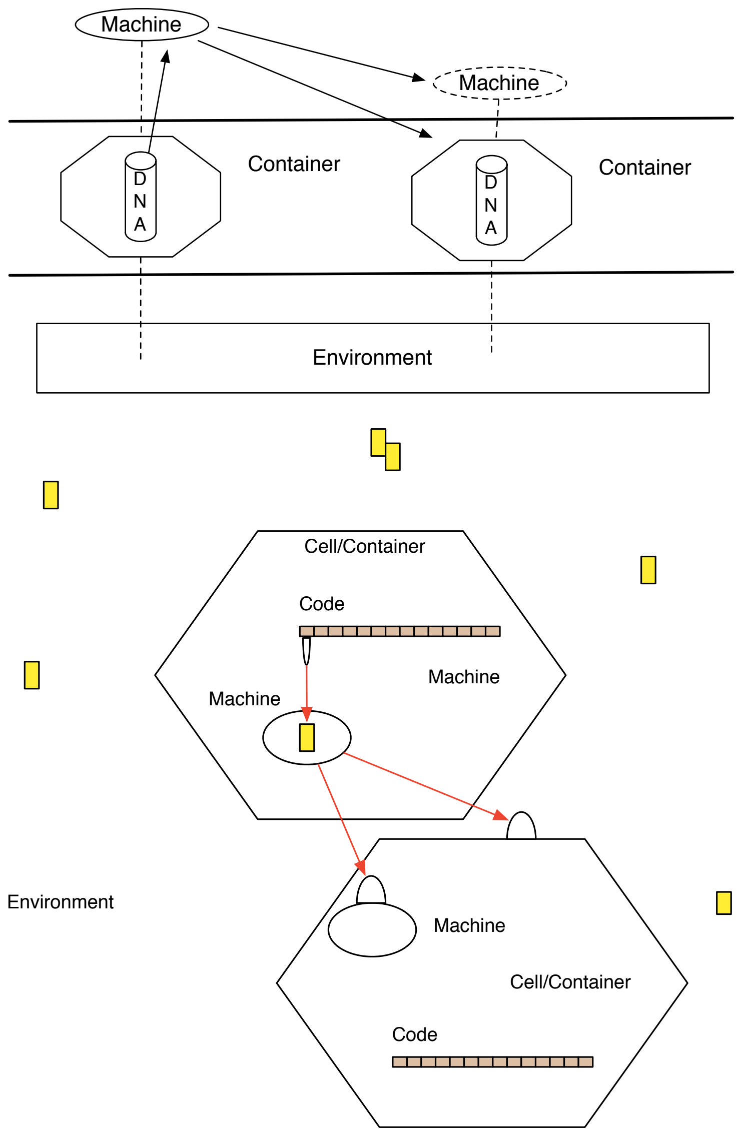


Figure 5.2. Top: the effect of the replicate for the environment, the container and the machine level. A new container is created and in it a new machine is started. After the replication process, both of them are independent of the creating container. Bottom: an example of a replicating container. Replicate introduces a new container into the system and starts a new machine in the container created. (The yellow rectangles represent information.)

not possible, then the whole system would only produce, exponentially fast, cells of the same type for ever. Having all cells in the same state is not desirable; containers need obviously to behave differently otherwise nothing sophisticated could be constructed. Therefore any 'replicate' primitive must allow for state changes in the newly born cell.

The paradigm states that there must be a replicate primitive; it does not define how the replication should happen. The replication therefore does not have to be a physical replication but can also be a purely informational replication. In fact information replication is what happens in G-Code, G-Code defines replication as a purely informational transfer. It creates a new object in the memory that is a copy of the source object with the ability to start a different G-Machine in the new object on replication and with the possibility to have intracellular substance that are divided potentially asymmetrically to change the state of the simulated cell. The possibility of asymmetrical replication is one of the key points that a system must have to exhibit self-constructing behavior.

5.4.2 Eliminate

‘Eliminate’ is the ability to remove a container from the system. During the development of the system, replicate generates additional instances of containers in the system. Some of these containers are only created to act as guidance for other containers and do not later contribe to the final constructed structure or organism. The containers can therefore act as a transient scaffold only for the orientation of other containers. Furthermore it can be that through a mishap a container is placed at the wrong point, or is damaged. In that case it must be possible to remove it. In biology, eliminate is quite common and is named apoptosis. If a cell is in the wrong place during development it can undergo programmed cell death (apoptosis). Apoptosis is used to remove virus-infected cells [60], and it is also used to allow for the separation of the fingers in the hand during development. These cells are function purely as a scaffold for the construction of the fingers and toes [16]. Any system that wants to allow for this scaffolding effect or for the correction of mistakes must have the ability to remove one container from the system, even if the actual implementation only moves the container physically out of the way of the rest of the construction process. Cell death is actually a process executed autonomously by the cell, but the cell has special receptors that allow external signals to trigger its internal cell death program. It is again the case that in biology a cell apoptoses itself and it is not *done* from the outside. In Cx3Dp’s G-Code the process that does eliminate is called die. Figure 5.3 shows an eliminate process happening in a stereotypical environment.

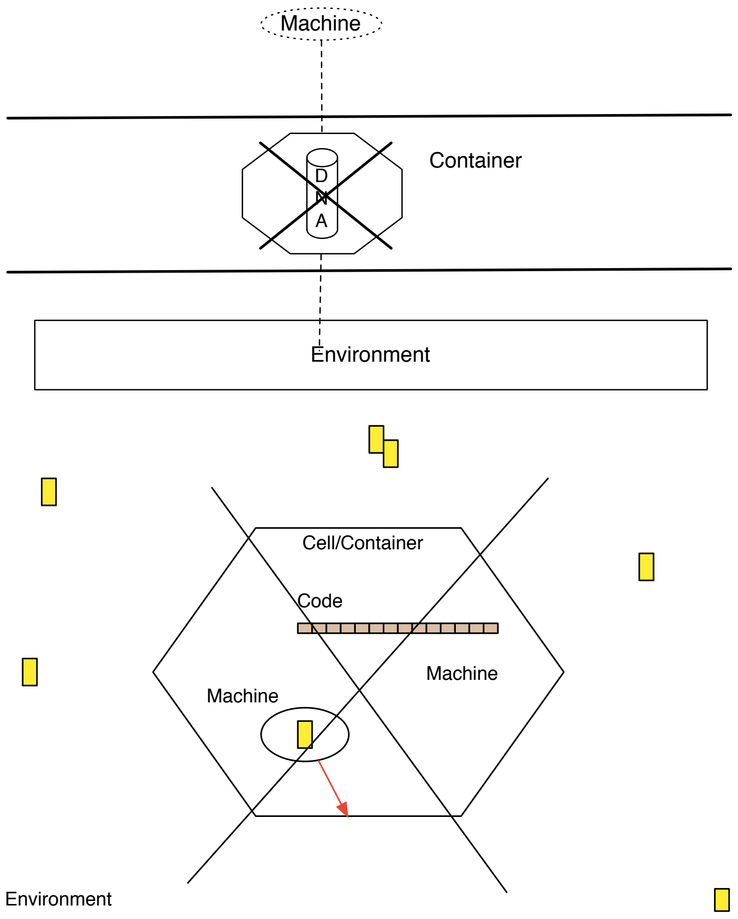


Figure 5.3. Top: the effect of Eliminate in each of the elements involved (machine level / container / environment). Bottom: the effect in a stereotypical self-constructing system. An eliminate primitive removes a container from the system. (The yellow rectangles represent information.)

5.4.3 Activate

Activate abilities are responsible for the translation of a machine from its textual form into an active agent that then begins to influence the behavior of the container. Machines can only be activated within the container. No outside influence can activate a machine in the container. It is possible through a cascade of read and activate to activate a machine within the container as a reaction to outside signals. But this mechanisms of 'listening' to such a signal must be present in the form of a machine. Activate primitives are needed such that the passive textual from can be translated. If activate were missing, the container could never change its behavior and would constantly do the same thing. It is very much wanted that containers can change their behavior, otherwise the orchestration of the cells would not be possible, and the container would remain in the same state. How the actual implementation is handled, i.e. whether the machine is really translated on the fly or just activated when needed does not matter from a theoretical point of view. All that matters is that the functionality of the machines can be called into action on demand. The biological cell activation would correspond to a translation from the genome into a functional protein/protein-complex/biochemical pathway. In G-Code activate corresponds to the primitive instantiate that creates a new machine within the container. Figure 5.4 shows an activate process in a stereotypical environment that allows for self-construction.

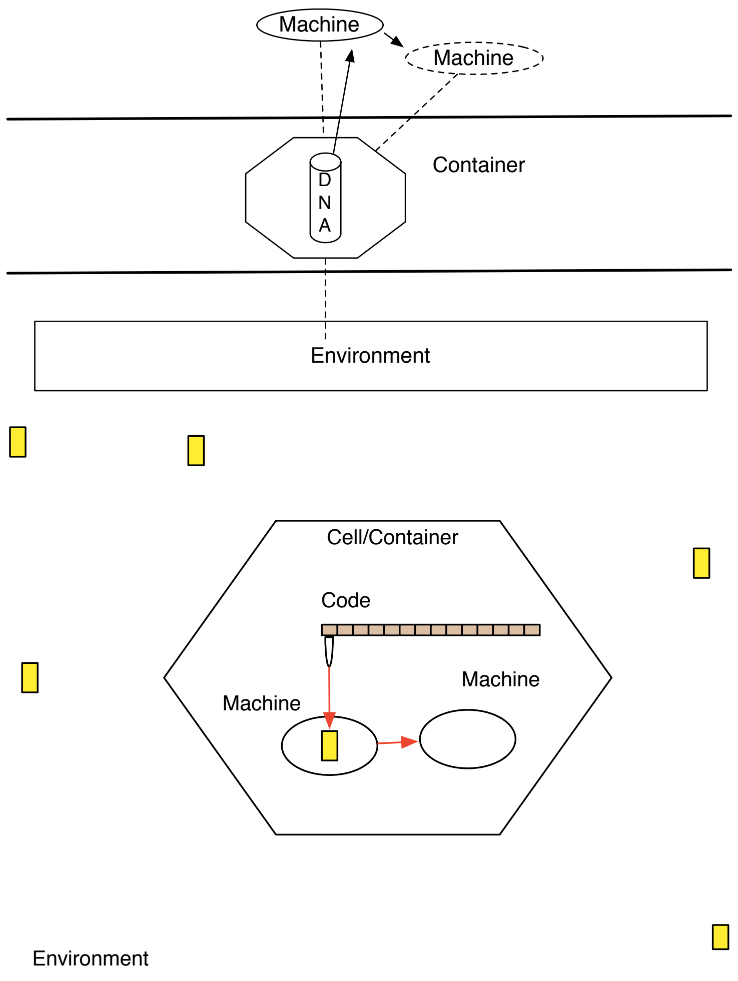


Figure 5.4. Top: the effect of activate in each of the elements involved (machine level / container / environment). Bottom: the effect in a stereotypical self-constructing system. An activate primitive translates a machine from code into active form within the the same container as the activating machine. (The yellow rectangles represent information.)

5.4.4 Terminate

The primitives of the terminate category are the opposite of the ones in the activate category. Terminate will make a machine stop. Note the difference to elimination where the complete container is removed from the environment. The terminate category of primitives or abilities is necessary for the same reasons as the activate is necessary, without it it would be hard to change the cells' behavior on demand. At some point a machine has reached its goal and finished its task within a container. That is when the machine should be terminated in the container. Terminate is only possible within a container and can not be controlled directly from outside without a signaling cascade machine involving a read and a terminate. In biology, terminate corresponds to the degradation of a protein/protein-complex/biochemical-pathway to ineffective levels. In G-Code the only primitive that does termination is 'kill' that will remove a G-Machine from the cell. The effects on the system are shown in figure 5.5.

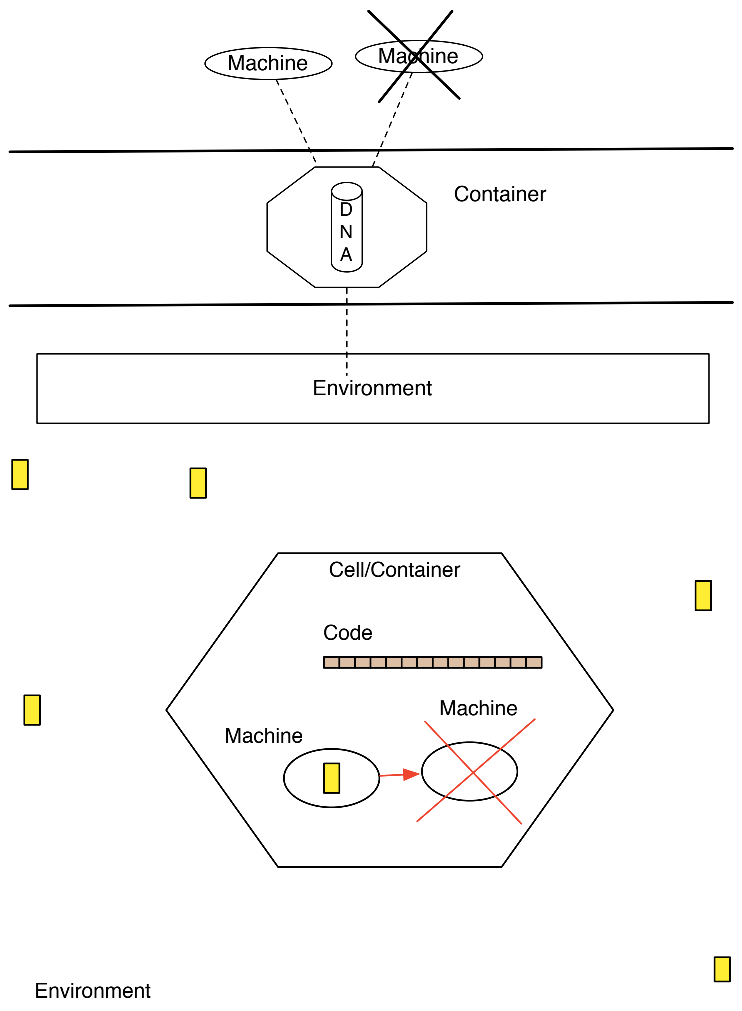


Figure 5.5. Top: the effect of terminate in each of the elements involved (machine level / container / environment). Bottom: the effect in a stereotypical self-constructing system. A terminate primitive removes a machine within the container. (The yellow rectangles represent information.)

5.4.5 Read

In order to be able to communicate between two machines that are not in the same container it is necessary to have a way to read from the environment. Even for two independent machines within a container it is important to be able to communicate, to coordinate behavior. Communication always refers to information transfer. This information can come in different forms and/or encodings, but always serves the purpose of communication. Communication between containers is essential in order to allow for coordination of the containers. Read and write abilities of the container allow for such communication. Another aspect of a read ability is that if a machine that contains such a primitive is not active in a container, the container will not notice even if there are potentially readable chemicals in the the vicinity. Read is a way to selectively choose which signals should be integrated into the cell. Roth calls this effect of only selectively reading signals into the container a 'symbolizer' [159]. Having such symbolization in place, there can be many signals around for multiple orchestration purposes and only the cells that should read a signal do read it. Read is designed to be an active process and not just something passive that happens to a container. In biology this read mechanism is implemented through sensing the membrane proteins [47] [208] of neighboring cells or through the ability to sense morphogens [12], and other possibilities. In G-Code typical read actions include the sensing of neighboring cells, reading of gradients and concentrations of extracellular chemicals, and also reading the physical parameters of the cell such as its size or the tension it is under.

Figure 5.6 shows a stereotypical read primitive in action.

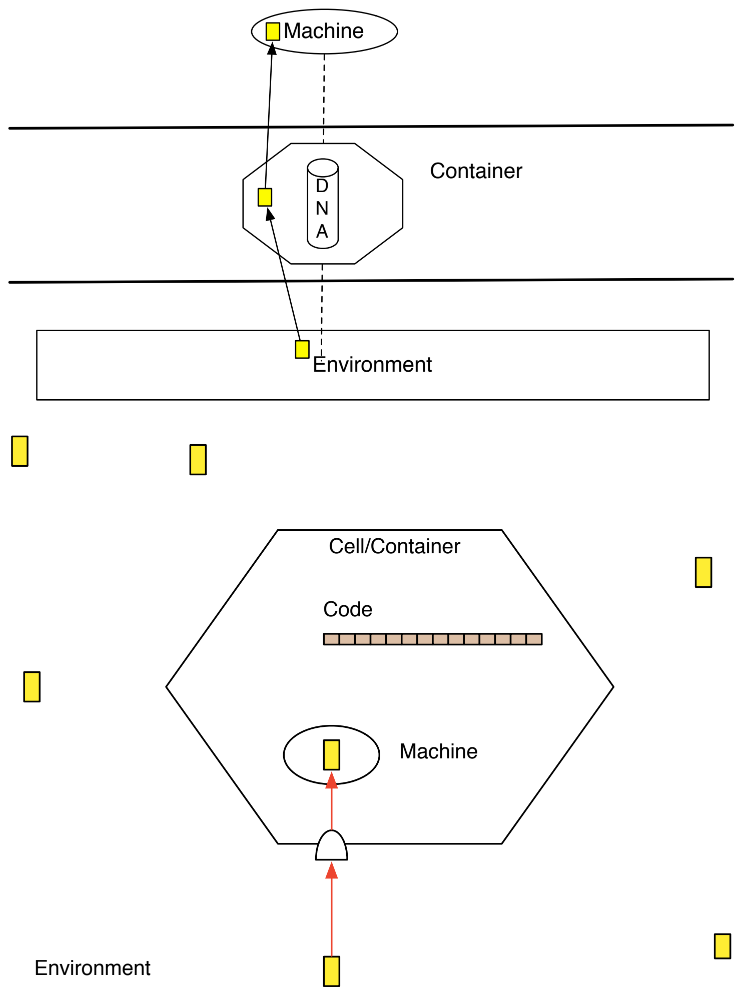


Figure 5.6. Top: the effect of read in each of the elements involved (machine level / container / environment). Bottom: the effect in a stereotypical self-constructing system. An read primitive transfers information from the 'physical' environmental world into the reading machine. (The yellow rectangles represent information.)

5.4.6 Write

The write abilities are the counterpart of the read abilities. Write primitives are needed to communicate between machines of different containers or within the same container. It is important to note that all the written signals must be readable or have a passive effect on the environment. Otherwise the information is not able to reach other containers. It is obvious that matching read and write abilities are needed so that the cells are able to read what has been written. Typical write abilities of the cell in biology are secretion of morphogens or expression of membrane chemicals. In G-Code typical write primitives of the artificial cells include secretion of intracellular chemicals or the change of cell properties through morph, or even attachment to other cells via physical bonds (which has a passive influence on the physics of the environment and can be read by the other cell). Figure 5.7 shows a stereotypical write.

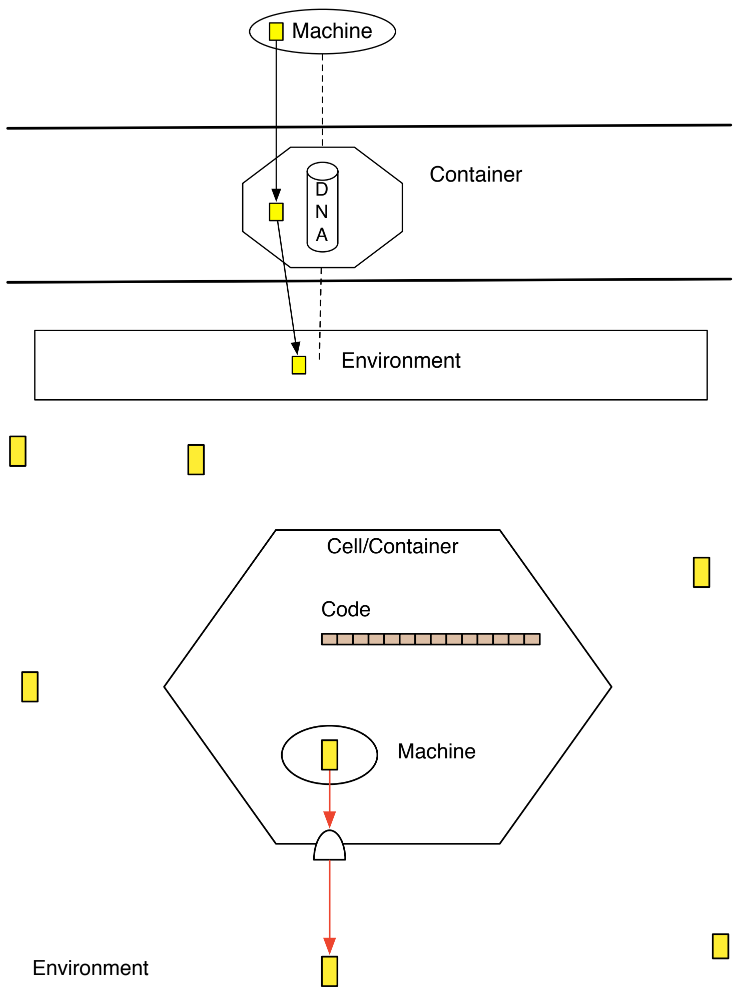


Figure 5.7. Top: the effect of write in each of the elements involved (machine level / container / environment). Bottom: the effect in a stereotypical self-constructing system. A write primitive transfers information from the machine to the 'physical' environmental world. (The yellow rectangles represent information.)

5.4.7 Filter

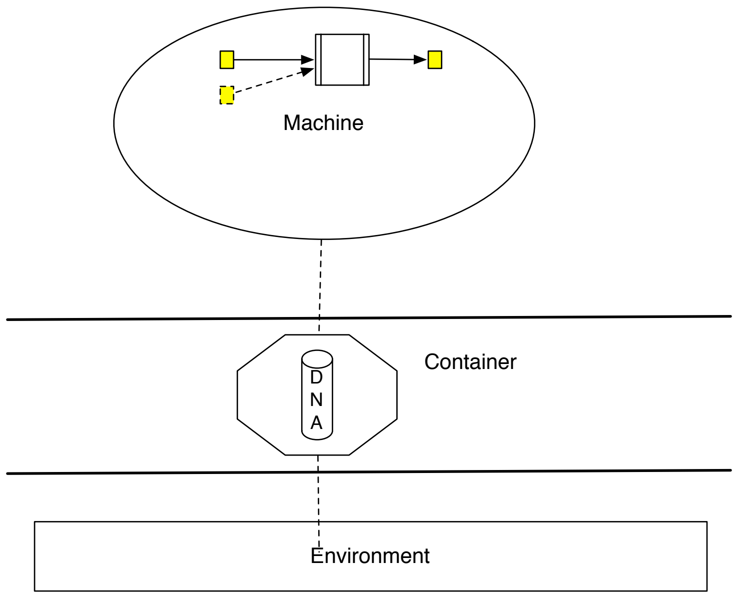
The ability to filter signals by applying either mathematical or logical functions is essential for signal transformation between different primitives. In biological cells such abilities include dampening or amplifying of signals detected in the vicinity. Even very small changes in gradient can have a huge effect on the direction of growth of growth cones [158], for example. Even decisions to change behavior based on the detection of a signal such as the stopping mechanism for the radial migration of cells upon arrival at the marginal zone [152] are executed by the cell which is essentially a logical if operation. In G-Code, mathematical and logical abilities and filters such as addition, multiplication, AND, OR and IF are used to transform signals. Signal filtering is a machine-internal action and does not interact in any way with the world outside of the machine (see Figure 5.8).

5.4.8 Move

The ability of the container to move in the environment adds to the coordination possibilities of the whole system. The container can move in the dimensions of the environment and translocates the information and the function encapsulated within it. In addition, move allows the container to gather more information about the local environment. Movement of cells can be observed in biological development in many places. Move helps to coordinate cell fates and to make sure that cells end up at the right place. One example would be the radial migration of pyramidal neurons to reach the cortical plate [151]. Movement of cells is often coupled with the sensing of a gradient of chemicals that will give the move a direction. In Cx3Dp's G-Code movement is allowed in all three dimensions. The primitive is called move. Instead of moving containers, one could imaging just replicating in a certain direction with later elimination of the original cells. However, movement through replication is an extreme waste of resources if it is done for each translocation action. Move and movement through replication are possibly interchangeable. This was already noted by von Neumann who distinguished between cellular and kinematic self-replication [31]. The latter implements real movement whereas the former moves through replicate and eliminate. Information transport is essentially the target of move: bringing information and/or functionality to the right place in the structure.

In some abstract environments the dimensionality of move is not necessarily three: one could also imagine higher dimensional spaces or even varying dimensions for the translocation of the containers.

A stereotypical move is shown in figure 5.9.



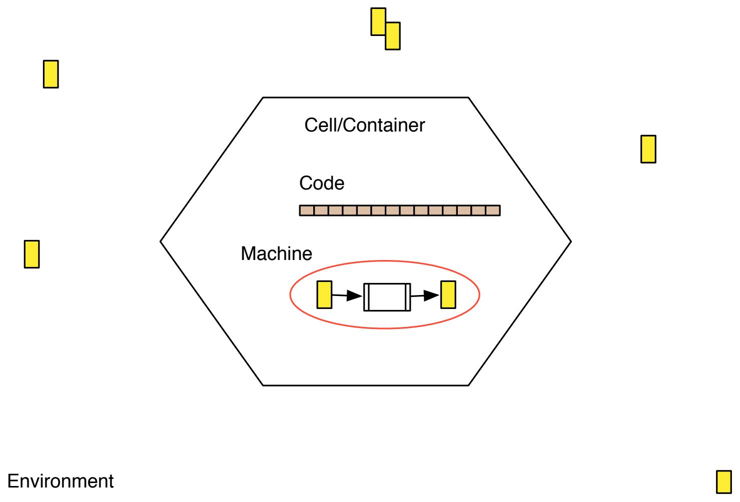


Figure 5.8. Top: the effect of a filter (the box with the double-line sides) in each of the elements involved (machine level / container / environment). Bottom: the effect in a stereotypical self-constructing system. A filter primitive transforms and transfers information within the machine between primitives. It can transform the information according to the mathematical or logical rules it defines. The filter has no state and always executes the same transformation. (The yellow rectangles represent information.)

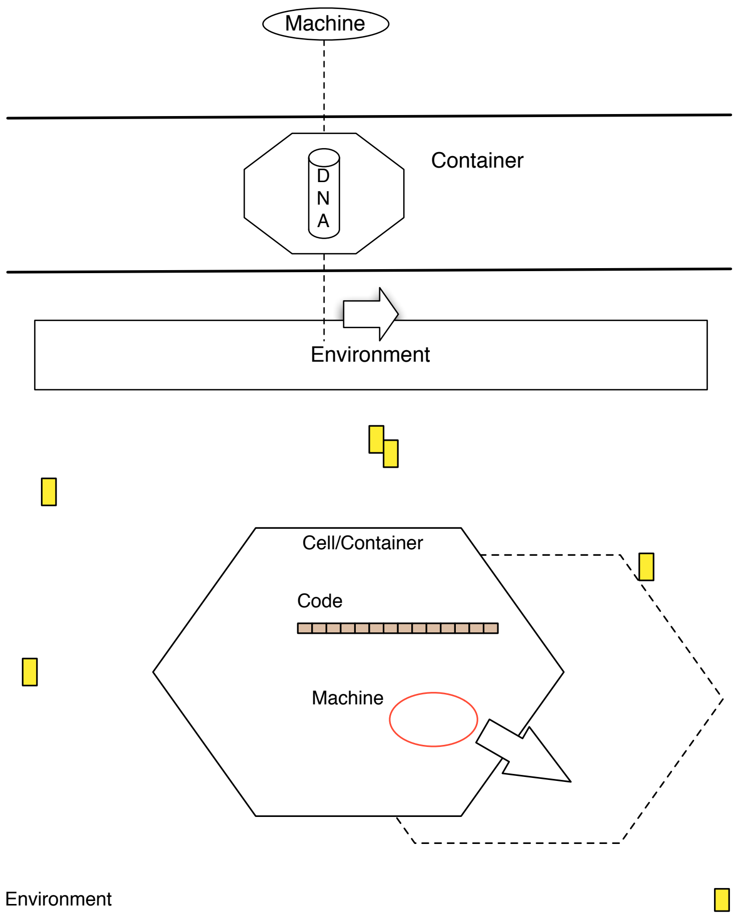


Figure 5.9. Top: the effect of move in each of the elements involved (machine level / container / environment). Bottom: the effect in a stereotypical self-constructing system. A move primitive translocates the container (and all its associated information and machines) in the environment. (The yellow rectangles represent information.)

5.5 Principle of Locality, Limitations and Computational Power of Developmental Programming

The implementation of a system that permits the developmental style of programming requires the definition of an environment (dimensionality, physics etc.); the definition of one or more containers that live in the environment (e.g. soma, growth-cone); the definition of the abilities or primitives that the containers can execute according to the categories given in section 5.4 and the ability to assemble these primitives into machines. Of course a DNA like structure needs to be available to store the machines that are programmed. If a system follows these criteria, it offers future programmers a framework in which they can take full advantage of the Developmental Programming paradigm. Note that there is a distinction between the programmer of the system and the provider of the system. The programmer does not need to worry about how to comply with the rules of the Developmental Programming paradigm since the programmer is forced to only use the available language implementation. This language is necessarily compatible with the paradigm; the provider of the system has to ensure that. For example in Java, an object-oriented language, the provider of the system is the one who defines for the Java language what is valid and what is invalid to use where and the programmer must obey these rules to be able to compile his code in Java. The same analogy goes for the Developmental Programming paradigm.

In a system that supports the Developmental Programming paradigm each container is independent of the others. This independence allows for a high degree of parallelism. This means that each container can potentially be computed separately due to the locality principle which says that the containers can only be at a certain place in the environment and interact only at this place. This implies that a concurrency protocol is only needed in these local places in the environment and only when reads and writes to the environment occur. The programmer of the system never needs to think about the concurrency aspects since the concurrency protocol is inherently given by a system that supports the Developmental Programming paradigm. The containers can replicate themselves, implying that the number of processing units and therefore the computational power of the system can grow potentially exponentially.

Although a system implementing this Developmental Programming paradigm limits the programmer to use just the given primitives for the containers, it automatically enables the programmer to write code that runs inherently in parallel and is therefore scalable. The paradigm limits the programmer to only use the developmental style of programming, these limits however are not computational. A system that follows this paradigm is computationally Turing complete. Hence any computable shape can be computed in such a system. The Turing completeness follows from that fact that the container (a writing head) can read and write on the environment (tape like structure), it can move in the environment and the container has a state through the execution of the machine [187]. All the necessary functionality is given in order to have a universal Turing complete system. Though having the program encoded in a DNA-like text form that cannot be changed at runtime makes the system closer to a von Neumann architecture than to a Turing machine directly [195].

5.6 G-Code: A Language Based on the Developmental Programming Paradigm

The new version of G-Code (Appendix A) is a concrete example of a language implementing the Developmental Programming paradigm. The new version of G-Code is based on Zubler's original implementation [213] (see section section 2.2.4 for a description). The new version of G-Code was written for Cx3Dp (see section 6).[1] Cx3Dp provides the ability to direct cellular growth. The virtual cells in Cx3Dp are approximated with a spherical soma and a cylindrical neurite element. Cx3Dp provides an implementation of physical interactions between cellular compartments and a diffusion grid for diffusing extracellular and intracellular substances. In this section it is shown that G-Code follows all the important criteria that are needed for an implementation of a self-constructing system. In the following we describe each of them.

Looking at the system from the view point of the Developmental Programming paradigm, the system provides an environment (the physics implementation) and two types of containers: a container for the cell body (soma) and a container for the neurities (neurite element). In order to provide the possibility to control the soma or the neurite element with behavioral programs, a set of primitives have been developed. These primitives allow the programmer to use all the possible abilities that somas or neurites can exhibit. In table 5.1, all the primitives are shown categorized into the eight categories: Replicate, Eliminate, Activate, Terminate, Read, Write, Filter and Move. Since G-Code has two different types of containers, two different implementation of the base machine were needed since the primitives have slightly different effects. For example the Move in the soma would dislocate the soma where as the move in an element at the tip of a neurite would elongate the neurite. But still both implementations would relocate the executing container. Because the abilities and primitives of both containers are very similar both have been integrated into the same table, table 5.1.

The Replicate category is interesting because one can see that there is more than one primitive namely the replicate and the fork primitives. The primitive replicate is simply the functionality of creating a new cell when executed on a soma; the cell will divide. Executing replicate

[1] During the implementation of G-Code, more primitives were added to the system but they all belong to the categories of Read, Write and Filter, improving the communication between the containers in the system. The other categories remained nearly identical.

introduces a new container (another soma) and activates a machine in the soma. A very fitting primitive for the replicate category. Fork on the other hand does not look suitable for the category of replicate at first glance. Fork has the effect in a soma of growing out a new neurite, and in the neurite will add a new branch (bifurcation or side-branching). But if we look more closely at the behavior of fork, it introduces in both cases a new container, a neurite with a new machine in it (a growth cone machine). The effect of fork fits perfectly into the replicate category according to our definition in section 5.4.1.

Activate	Terminate	Move	Eliminate	Replicate	Read	Write	Filter
instantiate	kill	move	die	fork	AttachHasPhysicalBondWith	AttachPhysicalBond	Madd
		moveAlong		replicate		AttachRemovePhysicalBondCellElement	Msub
		moveBackOn					Mdec
					MoprhProperty	MorphAdherence	Minc
					MorphAsymmetryConstant	MorphAsymmetryConstant	Mdiv
					MorphAdherence	MorphColorColor	Mmult
					MorphDiameter	MorphDiameter	
					MorphIgnoreBoundaries	MorphIgnoreBoundaries	Mnorm
					MorphInterObjectForceCoefficient	MorphInterObjectForceCoefficient	MscalarMult
					MorphMass	MorphMass	
					MorphVolume	MorphProperty	MinRangeBase
					MorphColor	MorphVolumeVolume	
							MthrowDice
					DetectExtracellularConcentration	SecreteExtracellularConcentration	Mrand
					DetectExtracellularGradient	SecreteExtracellularQuantity	MHill
					DetectIntracellularConcentration	SecreteIntracellularConcentration	
						SecreteIntracellularQuantity	
					TasteDistanceFrom		Land
					TasteForceFrom		Lor
					TasteMaxInteractionDistance		Lnot
					TastePhysicalBondedObjects		
					TastePropertyEquals		LisSet
					TasteSurroundingNeurites		LSmaller
					TasteSurroundingNeuritesCellType		LBigger
					TasteSurroundingSomas		LEqual
					TasteSurroundingSomasCellType		
							if
							else if
							else

Table 5.1. Table showing the primitives of the new implementation of G-Code sorted into categories.

In simulation we show that the interplay of Cx3Dp and G-Code is working and mimics the behaviors observed in biology. That this claim holds true can be seen in chapter 3. Since the containers are virtual and do not create new computational power in the hardware sense, it means that as the system grows, the developmental process gets slower. The hardware does not replicate with the software. But it can be seen that the whole system is very well suited for parallelization. Each cell could in theory run on an independent core. It can also be seen that the available computational power is quickly used up by the replicating cells. Even though the program runs in parallel the programmer of the containers never has to worry about how to parallelize the code written as it is automatically parallel.

5.7 State of the Art Programming Paradigms vs. Developmental Oriented Programming

The two most popular programming patterns that have arisen over the last few years are object-oriented programming and functional programming [73] [176]. Object orientation is used in industry standard languages such as Java, C++ and C#. It is often used for large software projects. Functional programming is more of a niche paradigm. It is ideal for being able to prove the correctness of a program [15]. It is has a high degree of similarity to lambda calculus [41]. Functional programming is often used in safety-critical applications where software bugs can have fatal consequences. [82]. In this section the Developmental Programming paradigm is compared to the object-oriented and functional programming paradigms to see the similarities and differences and to show that they are related but not the same.

In object-orientation, the central abstraction is the object, or rather the class. Classes have certain functionalities and data associated with them. In functional programming the function is the main abstraction. The Function takes information as input and transforms it to an output; no other information may enter the computation. By comparison the Developmental Programming paradigm takes the machine as the main abstraction. The machine defines the behavior of a container in the environment. The container can be seen as an object, but in the Developmental Programming paradigm the container is given and does not have to be modeled by the programmer of the system whereas in object-orientation the goal of the programmer is to define the object. In developmental-oriented programming there are no objects that can be programmed. In developmental-oriented programming there are only emerging structures that appear during the execution of the program. These structures are made entirely of the containers. But these ‘structural-objects’ are not under the direct control of the programmer but must be controlled by their building blocks, the containers.

Communication in these three paradigms is very different in each case. The communication in object-oriented programming is through the access to objects' data either through their functions or through fields provided by the objects. Objects are writeable and/or readable by other objects but can be restricted to be completely unwritable and only readable from the outside of the object. In functional programming [89] the communication is handled via the input values to the function and its return value. The machines in Developmental Programming are by definition completely independent of other machines, the only method of communication is through the environment via the functionality provided by the container. A machine can read and write through the given primitives of the container that are defined by the system to be programmed.

Concurrent programming in these paradigms is directly related to the communication, because wherever there is communication potential problems with synchronization can occur. In functional programming it is possible on function calls to introduce parallelism because each function is completely independent of all the others as soon as the input is provided [145]. In object orientation, concurrency is one of the major problems of the paradigm. It is very hard to write correct concurrent code in object-oriented languages, because the interdependence of the program is often non-trivial; one cannot define each object as a parallel process. There are no fixed pattern solutions to this problem, but many attempts to solve it [4] [109] [75] [10]. In Developmental Programming, through the independence of the machines each machine is in theory executable in parallel. Concurrent access to data via read and write has to be ensured by the environment that is given. The only interaction of machines is through containers that have a location in the environment. Therefore no global communication is possible, which ensures only local concurrent access in the environment. Each machine can be seen as an infinite loop that is only stopped on calling the kill primitive. Parallelism in this paradigm is inherent. Another way to put the difference in concurrency between object-oriented and developmental-oriented programming programming is that in object-oriented programming, parallel processes operate on the object structure whereas in the Developmental Programming paradigm the machines are the parallel processes. This difference shows how the data is encapsulated in the three different paradigms. In object-orientation, data is stored in the objects and potentially hidden in the object. In functional programming, the data is stored during execution in the input and output values of the functions. In the Developmental Programming paradigm the data is stored in the environment, the containers 'physical' parameters and the potential states of the machine. The data in the machine is only accessible via the machine and the data in the container and environment is only locally accessible by machines that are in containers located in the vicinity. In the Developmental Programming paradigm the data is where the processing happens or if the processing container wants to access other data it has to move to the data. Location is the main encapsulation of data.

Code reuse is another important aspect of programming. In object-orientation, polymorphism encourages the reuse of classes and the class encapsulation the reuse of their functions. In functional programming it is the subroutines that allow for the reuse of code. In Developmental Programming it is the machine that can be integrated into other machines. Code reuse is in all cases possible and desirable.

The program itself in each of these paradigms in stored as definitions either of objects (classes), functions (function definitions) or machines (artificial DNA). It has to be stressed that in Developmental Programming the code has to be accessible explicitly in each container, otherwise new machines cannot be produced. The execution of the program starts in object-orientation usually with a main method that acts as the root of the program. At the end of the program the end of the main method is reached and the program usually terminates (there are exceptions to that are not described here). In functional programming it is guaranteed that at the end of the initially-called function the program is terminated. In the Developmental Programming paradigm, the initial machine in the initial container starts the process of development. When the machine creates another machine via activate or replicate it has no further control over the initiated machines. The machines live independently and have no knowledge of who created them and will not report back to their parent machine on termination via kill. It can even be the case that the machines never cease to exist during the lifetime of the developed organism. For example, this effect allows for the constant monitoring needed for self-repair or constant adjustment to the environment of the organism. (A biological example would be the learning rules in synapses implemented via proteins [119].) Following the developmental scheme of execution, the system is inherently distributed with no global control. Interestingly enough, Developmental Programming in contrast to object-oriented programming does not have explicit loops. One can do ‘for each’ loops in order to check all neighboring objects but not a logical loop. That there are no loops needed arises from the fact that each machine is in principle a never ending loop encapsulating the machines program and is executed constantly. Functional programming also does not have explicit loops but uses recursion to emulate it [89].

All of these paradigms deal with a model of computation that is Turing complete. All of these paradigms are just an abstraction for helping the programmer to think about a problem. It would be possible to implement all of the algorithms from one paradigm in another one, but it might be easier for humans to think about the algorithm in one paradigm then in the other. Where Developmental Programming is really strong is in creating controllable parallel executable developing systems that potentially grow in computational power exponentially during construction. Parallelism is where the object-oriented approach struggles but technology is heading exactly towards highly parallel systems. The Developmental Programming paradigm helps the programmer with its limitations to program the behavior of a given system that allows for self-construction. The system ensures that all the programming primitives are given, just like a compiler for an object-oriented programming language ensures that all the right assembler instructions for the given CPU's are used.

5.8 Discussion

We have made a bridge between what can be learned about the construction process in biology, the biological development, and self-construction as an engineering principle. We have attempted to transfer this insight into an engineering method, namely the Developmental Programming paradigm.

We have explained how a system has to be designed in order to support the Developmental Programming paradigm. Such as system consists of an environment, one or more physical container types and an instruction language to program these containers with the help of machines. These machines are stored in a textual from in a single string and can be activated on demand by the containers. An active machine guides the behavior of the container. G-Code is an implementation of the Developmental Programming paradigm [209]. We show that such systems are computationally powerful enough to compute any computable problem by being Turing complete.

Systems that follow the Developmental Programming paradigm have inherently many advantages. Among the strongest are: the capability to self-repair, the potentially exponential acquisition of computing power, and the lack in general of a single point of failure. They are suited to the iterative improvement of structure during construction, which means that intermediate goals can be defined in the design process that act later on as a scaffold for future structures. This idea of iteratively improving scaffolds makes the systems following the Developmental Programming approach also suitable for evolutionary improvement, where the structure of the system (namely the number of primitives and their parameters) already puts boundaries on the search space, and intermediate results can be checked for their 'fitness'. In addition, after the system is designed the construction process is fully automatic and requires no human input. Such a system will therefore computationally not be more powerful than any other given paradigm such as object-orientation, but the abstraction is more suited for instructing container behavior than object-oriented programming is.

Object-orientation seems not to be the right tool to think about the behavior of entities operating in parallel. That this approach fails can be seen in that it is thought to be a very hard process only for experts to do concurrent programming with objects and in that there are no parallel programming patterns that always work [10]. In the Developmental Programming paradigm, all the objects are already given, or they are constructed as part of the organism to be developed. The abstraction needed for a developing system does not follow the classical view of entire objects but rather the view of how a nearly unimaginable number of small behaving entities can organize themselves. This organization involves a huge amount of parallelism that is hard to express in an object-oriented way, but follows naturally from the Developmental Programming paradigm.

Functional programming is also not the right fit for a developing system. The replication of a container resembles the call to a new function, since in both cases a new functional element is invoked. But in the case of functional programming, the function ends, which it does not do in the case of the newly created container. The function returns to its caller function whereas in developmental oriented programming the new container is completely autonomous and does not know about its ancestry nor can it be controlled by it.

A programming language for a system that follows the Developmental Programming paradigm will only have very limited functionality that is offered to the programmer through the abilities of the container. The central point for a programmer of such a system will be the development of machines and hence the behavior of the container. The challenge for the programmer is when to activate what behavior in order to achieve global coordination. The programmer in such a language is forced to use local rules bound to single containers. Through their locality property the containers are automatically executable in parallel. We enforce a programmer to think in a way that is drastically different from the object-oriented standard way of programming.

Replicate is one key aspect of the Developmental Programming paradigm. Through replication it is possible to exponentially enhance or consume the computational power of the given system. With the possibility of having state in each container and that the state is decidable on replication, a symmetry-breaking effect is achieved that is very controllable. Replication with state change permits a vast number of containers to behave in different ways and solve complex tasks. Potentially, each container can execute different programs or machines and therefore behave differently. Through reading and writing, communication is assured between containers and allows for coordination of assembly. An noteworthy contribution on replication was written by von Neumann who gave a theoretical example of a self-replicating universal machine [194] [31]. But he did not consider how the automata finally constructed would interact with each other to work towards a common goal.

There are many similar concepts such as self-assembly and self-organization that should not be confused with self-construction. In self-assembly, we often talk about agents that all behave the same and assemble in a particular way that is usually uniform. Therefore self-assembly is often used in the context of materials science where certain materials passively, by moving to a state of minimal energy, assemble large sheets of structures that exhibit ordered properties [199]. Self-organization is a process in which local identical agents interact locally to give rise to a global structure, behavior or organization. There are examples of self-organization in many fields including physics [6] and biology[34]. As in self-assembly and self-organization, self-construction is agent based. In all three, the agents act towards an emergent global behavior that cannot be predicted purely from the single element; they have emergent behavior. But in the case of self-construction, the agents do not necessarily need to be of the same kind; not all agents need to exhibit the same behavior. It is important though that in self-construction, the agents can be produced on demand, and traced back to a single source. The Developmental Programming paradigm is an instance of self-construction.

Roth [159] already identified that the interaction between cell, environment and the DNA is a crucial one for understanding self-construction. We followed in his footsteps in the direction of creating design principles for self-construction. Compared to Roth though, we left aside completely the notion of learning even though we did not exclude it from our paradigm. Learning could be realized with reconfiguration of the structure through machines that are running locally in the containers, where reads are able to pick up signals that are relayed through a classical electrophysiological type of interaction between the containers. But this reconfigurability would need to be investigated further and more clearly and is not directly part of this thesis. Interestingly enough Roth identifies that it is necessary to have design principles that form a language, even though he is thinking of motifs whereas we rather have the idea of a programming paradigm that describes the necessary properties of a system to allow programmable self-construction and how a programmer can be guided to use the right abstraction. Roth proposes that the task for the engineer will become easier with self-construction. We cannot agree with this proposition since the whole process still has to be designed and programmed. The design step is not automated with this paradigm, but it is strongly guided in that it cannot violate the self-construction capability. Another interesting aspect is Roth's ideas on factories of reactors and competences which go in the direction of defining machines made of primitives. These ideas have been pursued by Zubler who gave a first implementation of a language [213] and an environment [211] that basically follows the rules of the self-construction paradigm. We have gone one step further and have given an abstraction of how such a language has to be designed to program self-constructing systems.

The concept of having a repertoire of software design patterns as described by the gang of four [68] is also an interesting concept which could be developed for the Developmental Programming paradigm. Such concepts might include:

- State space definition: For having one central machine acting as a gene regulatory machine defining the states of proliferating containers.

- Scaffolding: The idea of developing scaffold after scaffold.

- Encapsulation of function: Using many machines as high level reusable functions.
- Communication scheme: A Delta-Notch [9] handshake protocol for communication between containers.
- Homeostasis: For self-regulating local conditions and their restoration or reacting to a change in the standard conditions.
- Turing patterns: Turing shows interesting pattern formation for computable structures that could be used for generating global patterns [188].
- Self-repair: A general Pattern for the regeneration of damaged parts in the system.

Design patterns for the Developmental Programming paradigm are not part of this thesis but should be investigated in further work. That design patterns in self-construction are important has already been identified by Roth [159]. He even implicitly or explicitly identifies some of them. But he asks for off-the-shelf components — finished machines — that can just be reused in any situation. Off-the-shelf components exist for object-oriented programming like libraries or whole frameworks. Often though these off-the-shelf software components need a lot of rework before they can be used in situations other than originally intended [67]. Each programming task is always a tradeoff between the flexibility of the code and the direct solution to approach just the current problem at hand. For each organism to be designed the design differs, therefore also the code will differ and it would be hard to anticipate every single machine that will ever be needed for all the imaginable organisms. Sure, as we encountered in chapter 3 there will be certain machines that are nearly identical and could be made into off the shelf machine, but the subtle differences make a machine work in the needed situation or not. One can not anticipate in all the situations where these subtle changes come have to be made.

We expect that in the near future many systems will be created that are either not globally controllable or that are massively parallel in their nature or are even able to have replicating structures. Prominent candidates are nano-robotic systems. Nano-robots have been an idea for a long time, and it is beginning to be possible to create minute designed structures [143]. The trouble is that even if we had existing nano-robots we would not yet be able to make them behave in a such a way that they follow a coherent behavior. Algorithms exist that let robots behave in an organized way [135] [8] but these algorithms are hard to design and are not generalizable. If it was possible to design nano-robots such that they follow the Developmental Programming paradigm, then the engineering of swarms of nano-robots would become much easier. Robotic structures are currently being made that would act as smart dust, such as the robotic pebble. The robotic pebble cube-like robots are able to transform from one shape into another but they are not in the classical sense programmable [72]. These robotic pebbles do not have the capability to self-replicate but are designed to be cheap to produce, as in general smart dust systems are designed to be. Of course for a nano-robotic system one of the problems is that the available resources are very limited. These limitations make the developmental paradigm an interesting choice because the machines are only active and only need memory when they are currently serving a function. Hence most of the code in a container is in a dormant form that does not use resources except for storage. The translation process into active form and therefore the access speed to the storage need not be fast but could be rather slow and the system would still work. Our simulation of cortical growth has shown that in general only one or two machines are active at a time in one cell, limiting the need for resources even further. In general the single container must not be extremely fast. The advantage comes in the massive parallelism and the self-replication capability. This fact implies for nano-robots that there is no need for a huge memory and fast processing. Some of these nano-robots in these massive numbers are bound to fail. Using the Developmental Programming paradigm, self-repair mechanisms can easily be implemented through local means. The problem is though that a self-replicating nano-robotic structure has not yet been created. Even though investigations have been carried out in the 1970's, no self-replicating machinery could so far be produced. Von Neumann worked on a self-replicating machine [31] but never found a practical implementation. NASA's 'grey goo' was also an attempt to create self-replicating machines for a moon base but their efforts remained fruitless [57]. This was inspired by Feynman's idea of ever smaller factories that can produce smaller factories that then produce smaller factories for materials and so on until the nano-scale would be reached [62]. In recent times the search for self-replicating structures has gained new traction with the new nano-technology. This new research has produced some interesting and promising techniques where the creation of controlled self-replicating shapes is possible. DNA origami is one of these techniques to create self-replicating systems [174]. Recently even structures that can carry varying information and that can self-replicate have been created [197].

The only systems that truly self-replicate are biological systems, with their cells as the containers, proteins as the machines and the DNA code as the program. In synthetic biology, efforts are already being made to create artificial living cells. In 2010, Venter [71] and colleagues created an organism based on a genome that was completely artificially created on a computer. The single cell organism was able to live and replicate. The approach was not yet real programming but rather a stripping down of an existing genome with certain modifications. At some point it should be possible to identify the genome that lets a cell exhibit abilities as described in section 5.4. A synthetic cell would then be completely controllable in the sense that we described in this chapter, and one could program a multicellular organism that develops out of a single cell. A complete organism engineered or programmed and not evolved. Fully controlled engineering of multicellular organisms would open possibilities of designed organisms for food, as machines, for their function, or for building large structures out of biological materials — the cell as a construction worker guided by a programmer. Genetic modification would become more like programming than the bio-engineering it is now. The possibilities and opportunities would be enormous.

Biological or artificial self-constructing programmable systems would change the way objects are currently constructed by a global controller to being constructed by a distributed process. It will change the way to reason about construction. Physical self-replication in combination with programmability is not yet possible in an artificial system, but virtually it is possible. The current trend in processor design is moving away from ever faster single-cores, to multi-core architectures. If the trend continues these numbers will reach the thousands, millions or even billions of cores. But somehow these processors have to be instructed or programmed. If this programming is done sequentially, the process could take a long time. The idea would be to create a virtual container, that can exist in the memory close to the processor. A virtual container would allow for the Developmental Programming paradigm and hence for the implementation of a virtual assembly of containers. These virtual containers could then cooperate in an cell-like way to compute the structure of a virtual computational 'organism'.

One system that is in principle designed to have millions of cores is the SpiNNaker system [102]. Through its completely decentralized design, it is an ideal candidate for the implementation of a programming language following the Developmental Programming paradigm. The SpiNNaker chip contains small local memory for the execution of programs (or potentially machines) and a memory structure shared between multiple cores that is a bit slower that can store a larger amount of data (or the artificial DNA code). The system is designed to be locally connected, each processor is connected to just a few other processors. To deliver a program to the whole chip, only one of the cores needs to be communicated with. This is ideal for the seeding of a single virtual container that would then develop and spread the virtual organism over the whole of the spinnaker chip. The system is currently scalable up to approx. one million cores, a scale which should soon be reached. The same idea could be used to harvest the potential parallel computing power of LANs, clusters, and grid or cloud computers, or even better for larger scale networks such as the internet with millions of nodes. The network acts as a hybrid virtual/real environment with virtual containers spreading over it. Roth already proposes a similar network in his thesis [159]. It could be thought of as a piece of software that acts as a virtual environment for containers. The containers would populate the environment as a virtual organism to serve a computational function. For each new purpose of the virtual system a new artificial DNA code would be created and a new organism could be developed inside the virtual substrate.

It is clear that the potential fields of application for the Developmental Programming paradigm are tremendous.

Chapter 6

Cx3Dp: A Framework for Large Scale Tissue Growth Simulation

This chapter presents the simulation tool Cx3Dp. This tool has been developed over the course of this thesis in order to allow for the simulation of cortical growth simulation. Cx3Dp makes the the scale of the simulation presented in chapter 3 possible. Cx3D would not have sufficed for that. It is shown how the principles of the simulation tool work and it is compared with state of the art simulation tools for neural growth and other large scale simulation tools from different fields that face similar challenges for simulation.

6.1 Introduction

The development of an organism is a complex process of self-construction. Beginning with a single precursor cell, a series of mitotic divisions gives rise to a lineage of more specialized cells. As the lineage progresses, the successive mother-cells progressively distribute more specific construction information to their offspring through inheritance of particular configurations of gene regulatory networks, as well as by local environmental labeling. This combination of information affects division, migration, and differentiation of the individual in the context of the larger scale collective organization of its cousins. The process is all the more fascinating for neurons, whose final differentiation involves the elaboration of huge dendritic and axonal arborizations often on a spatial scale that is many orders of magnitude larger than the diameter of the soma. Understanding the principles of this extraordinary biological self-construction is a major intellectual challenge, the solution to which will have significant implications for biology, health, and technology.

In [211] we introduced a software framework, Cx3D, that permits the simulation of cell growth and behavior in a 3D space that respects physical processes such as cell division, cell-cell interactions, movement and chemical diffusion. The behavior of these cells is determined by intracellular molecular-gene-like codes that are expressed according to intracellular or extracellular conditions [213]. Cx3D is the first general purpose simulation tool for exploring neural development in a physical 3D environment, and is now being used successfully for productive research [1] [20]. The significance of the Cx3D approach is that individual cells as well as their expressed intracellular mechanisms behave as independent agents, able to sense only local 'physical' information, and able to act only through local 'physical' actions. Consequently, for the self-construction process to play out correctly in this simulation, the genetic configuration of the precursor cell must be correctly specified. The entire developmental process is implicitly encoded in this first cell. Furthermore, since there can be no appeal to a pre-labelled metric 3D space, this initial code must also contain arrangements whereby early generations of cells actively label the developmental space for use by their offspring. We will not discuss here the methods and results of these developmental simulations. These have been partially described in [213, 20, 212], and will be dealt with more fully in [78]. Our purpose in this chapter is to describe the technical advance which makes Cx3D scalable across multiple processors and machines.

The previous implementation of Cx3D has been very successful as a research tool. However, this prototype was designed for only single-threaded / single-computer implementation and so practical simulations were restricted to a few thousand cells. The restriction arises because the complex neuronal morphology is compartmentalized, and these large structures require many calculations to resolve the physical interactions occurring during the growth process [1]. Research questions on the scale of cortical development require simulation methods for cell populations a few orders of magnitude larger. There are no other software tools available for simulating such detailed growth models, and so we report here on a parallelized version of Cx3D that we denote Cx3Dp, with which it is possible to simulate systems of millions of growing cells on a network of only a few inexpensive commodity computers. The performance of Cx3Dp scales nearly linearly with the number of processors and machines over the range that we have been able to test so far.

In this chapter we describe and discuss briefly the interesting software design challenges that arise in cellular growth simulations of this kind, and our approach to their solution. We demonstrate the power of the parallel version of Cx3Dp using a number of characteristic cases. The first case demonstrates simple growth of a very large cellular sheet. The cells have a simple divide and grow rule, and simulation of the expanding sheet spreads over three computers. The second case tests the growth of axonal structures over multiple computers. Two opposing sheets of neurons grow axons towards each other. This case shows demonstrates that not only complete cells but also partial cells can cross computer boundaries. The third case demonstrates the integrity of physical diffusion processes across the distributed simulation using a simple diffusion scheme with several release sites and a volume distributed uptake. We demonstrate that Cx3Dp and the original version Cx3D are capable of running the same models and simulations. We compare the accuracy of execution by comparison with the original Cx3D using very nearly the same code as in the original. The final case simulates exponential growth to explore how Cx3Dp performance scales with computers and cores.

6.2 Biophysical Modeling

In Cx3D and Cx3Dp we have aimed to create tools that permit the simulation of huge developmental models in a similar manner to existing software tools for large electrophysiological simulations on morhoplogically specified neurons [126] [144] [28], except that in our case the focus is on simulating the physical growth processes of these neurons in a manner suitable for investigating corticogenesis at the level of cellular mechanisms, such as those responsible for cell morphology and the formation of inter- and intra-areal connectivity. In these tissue simulations each cell (and its expressed mechanisms) is an autonomous agent using only locally available information, and so development is inherently a densely distributed process that lends itself to parallel computation. The continual changes in the morphology and connection topology of neurons during development, as well as the continued synaptic changes of adult learning, place severe demands on methods of simulating distributed models.

The original version Cx3D contains a three layered model aimed at simulating cellular growth. In the top layer of the simulation, the biological parameters of the cell are modelled, i.e. how the cell behaves and its biological parameters. This biological layer heavily relies on the physical layer of the model. In this layer, all the physical properties are modelled such as the forces between somas and neurites, forces within the cell such as the tension within the neurites, and intracellular and extracellular diffusion of chemicals. In order to allow the physical layer to compute interactions between different cells, Cx3D contains a third and bottom layer that models the spatial relations between the cellular elements of the simulation. This spatial model is implemented by a 3D Delaunay triangulation in order to compute the neighbours of each cellular element, so that the forces between the cell elements can be computed. Cx3D does not need a large amount of machinery to organise the computation within the model, because it is single threaded and one computer only. It consists mainly of the model itself and a scheduler that computes one cellular component after the other.

The fundamental computational organisation of Cx3D has been retained in Cx3Dp (figure 6.1). The cell / environmental model has three layers of abstraction: spatial location, physical implementation, and biological implementation. The lowest 'spatial' layer of Cx3Dp models the spatial locations and spatial relationships between physical objects. The middle 'physical' layer represents physical properties such as mass and size and models physical processes such as interactive forces and diffusion. The top 'biological' layer assigns biological properties to physical components and wraps collections of these components as individual cells.

The layers interact with one another: each object in the biological abstraction is associated with a shaped object in the physical world that grants that part of a cell its physical properties. Thus, the biological layer represents somas and parts of neurites (neurite elements), and these biological parts are associated with spheres and cylinders in physical layer, while on the lowest layer the spatial organization of these physical layer objects are associated with their unique space-nodes, each representing the position of a physical object in the overall 3D model space.

The interaction between these layers can be understood through the example of a biological mechanism that senses the concentration of a signal molecule in the extracellular space. This biological 'molecular' mechanism is localized to a particular physical compartment of a neurite. The compartment is localized by its internal spatial node. This spatial node is linked to neighboring internal nodes (in adjacent neurite segments), and also to nodes in the neighboring extracellular space. The biological mechanism senses the required signal by examining the concentration of that signal in the spatial nodes that are external neighbors of the compartment's own node. When a signal is above a certain threshold the neurite may shorten by contraction, in which case the biological mechanism might increase the internal tension of its physical compartment. The distributed shortening of affected compartments will cause the overall neurite to shorten, and this shortening will move the spatial nodes of the affected compartments. As in the single-threaded version of Cx3D we assume that cells are embedded in an extracellular matrix to which they can stick. We have also retained the original force model. This model does not use standard Newtonian forces but rather an over-damping force model that evaluates the resistance of the extracellular matrix to acceleration [211].

The three model layers are supported by a technical simulation layer that provides simulation services such as schedulers and data-loggers, and does not form part of the theoretical model being simulated. This simulation layer was very simple in Cx3D. For example, the entire simulation was run on a single processing thread. This situation has changed radically in Cx3Dp.

The original Cx3D simulation framework was not naturally parallelizable, even though the model that it simulates is. The original simulation framework contained many dependencies between between the data of different abstraction layers, meaning that the interfaces between different classes were not defined sharply enough. These dependencies had the effect that changes in one part of the simulation could lead to global effects on the whole simulation directly, and not only through propagation of a well defined signalling process. The three layers are now more clearly distinguished to remove dependencies and provide better code encapsulation, so that the layer can perform distinct and concise tasks.

In Cx3D it was possible to control the whole simulation from a single main program, a central global control over the whole simulation. In Cx3Dp this is still possible though only if it is run on one single computer and it does not take advantage then of the multi-threading capabilities. Cx3Dp is optimised for an agent-based approach where each cellular part in the system is controlled locally by that cell part itself, by a small local program that is running in the soma or the neurite to mimic biological behavior.

6.3 Model and Simulation Services

Figure 6.1 shows the model of Cx3Dp; this model has has been kept from the Cx3D version. It defines the physical and biological processes of objects and the spatial relationships between these objects.

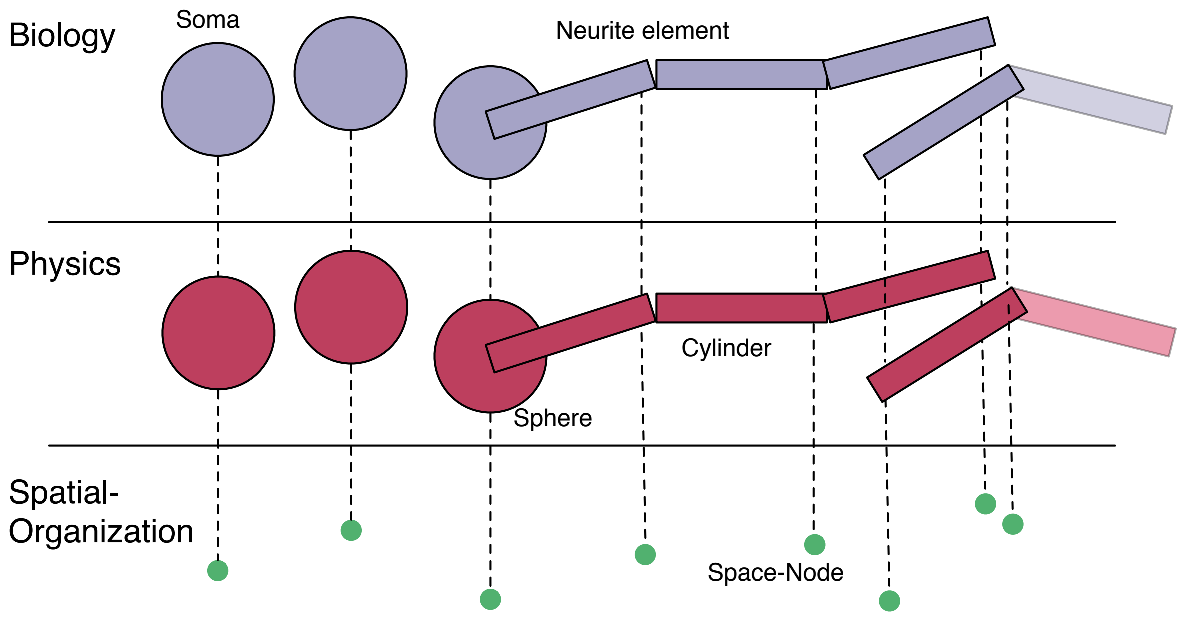


Figure 6.1. Computational layers of Cx3Dp: From top to bottom the different layers of abstraction: Biology, Physics and Spatial Organization. Cells are represented by compartments. There are two types of compartment (lilac), neurite elements for modeling axons and dendrites, and somas. Each element has a representation in each layer of abstraction. The top layer's somas are represented as spheres in the physics layer and as space-nodes (position indicators; green) in the spatial organization layer. The neurite elements are represented in the physics layer as cylinders that are each associated with a space-node.

To run a successful simulation the model has to be embedded in a simulation framework that takes care of keeping the model consistent. In Cx3Dp this administrative part is clearly separated from the model aspect of the system. The interaction scheme on the physical and biological levels remains the same as in Cx3D allowing us to offer the same user interface to users of Cx3Dp. The administrative part of Cx3Dp though has changed drastically, to prepare the simulation for use on multi-core/multi-computer architectures. In figure 6.2, in the very center of Cx3Dp is the model.

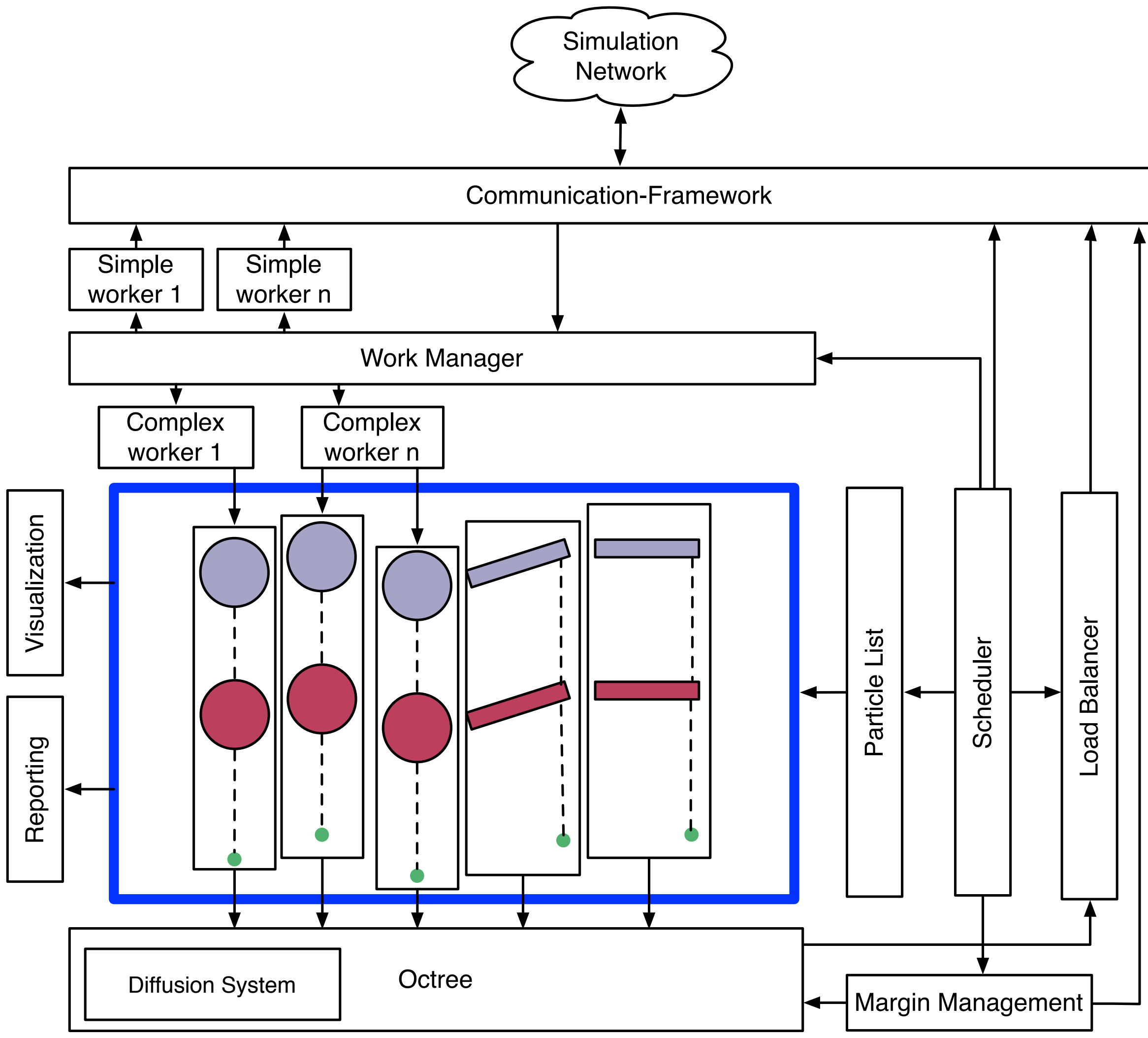


Figure 6.2. Model and Simulation Services: At the center of Cx3Dp is the model (outlined in blue), the biological and physical model that the system simulates. Outside of the model, administrative services are provided to ensure the consistency of the simulation, and the accurate computation of the model in each time step. Each box outside of the model represents a simulation service that has a clear task in the system. These tasks will be explained in the following sections. The arrows indicate the communication between the different services.

It is surrounded with components that ensure that the model remains consistent over the course of simulation in a multi-core/multi-computer environment. Each subcomponent of Cx3Dp has clear responsibilities. At the centre of Cx3Dp is the model, this model has been broken into smaller parts we call particles (see section 6.3.1). These particles are entities that cannot be further subdivided computationally and must stay together in order to assure the consistency of the computed results. We have three types of particles: particles representing somas, particles representing neurite-elements and particle and diffusion particles that represent diffusion space, see figure 6.3.

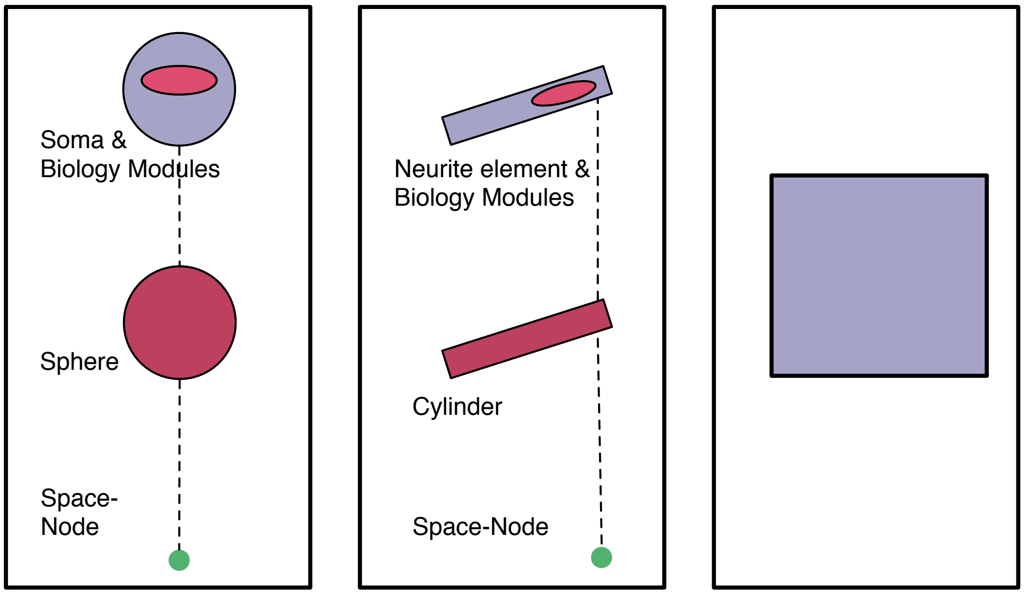


Figure 6.3. The three particle types in Cx3Dp. Left: the soma with its biological modules, sphere and space-node. Center: the neurite element with its biology modules, cylinder and space-node. Right: the diffusion compartment of the extracellular space. The black border around the particle indicates the inseparability of computation of the structure. The particle is all that is contained in the black box. For the soma particle this is the soma, its physical sphere and its space-node, for the neurite-element particle it is the neurite-element, its cylinder and its space node, and for the diffusion particle it is the part of the space of the octree it represents.

These particles have spatial relationships to one another, and if they are close enough they can interact. These spatial relationships are managed by the octree (see section 6.3.2). Whenever a particle needs to find out who its neighbours are it must ask the octree to find them. The particle itself does not possess the information as to which are the neighbouring particles (this has changed compared to Cx3D). The octree additionally manages all the diffusion of extracellular chemical substances since it possesses information about the spatial relationships in the system.

Additionally all the particles on a computer are registered in a global list (see section 6.3.3). Each simulation step in Cx3Dp calculates the changes of the system within a small time step. The organisation of the simulation is performed by the scheduler (see section 6.3.6). The scheduler engages the simulation process. One call to the scheduler leads the simulation to go through one simulation step. The scheduler has access to the global list of particles and assembles them into work packages because the simulation's task is to calculate the properties of each particle one time step after the other. The work packages are then passed on to the Work Manager that distributes the work to be executed by a pool of threads called the complex workers and simple workers (see section 6.3.3). To allow for a multi-computer system a communication framework (see section 6.3.4) has been devised that takes messages and/or tasks from other computers and executes them on the local computer. It also sends messages from this computer back to the neighbouring computers in the system. The scheduler organises when in the simulation process these messages are sent and when they are expected to have arrived. To allow multiple computers to participate in the simulation, the simulation work has to be distributed. Since the entities to be simulated are the particles, these particles are distributed among the computers of the system. In Cx3Dp, the space and the particles it contains are chosen as a way to partition the simulation. This requires a load balancing strategy (see section 6.3.8) to distribute the computation as efficiently as possible among the computers but also a strategy to exchange the data at the borders of space between computers, because the particles need to know about their neighbours in order to compute the next simulation step even though they might be located on an other computer. This is handled by the margin management (see section 6.3.5).

The model's state can be reported by the system in two ways: via the visualisation and the reporting that both have direct read only access to the model and can inspect all of its aspects (see section 6.3.7).

6.3.1 Simulation Particles

In the original CX3D, the model was mixed with the simulation framework and was processed on a single computer by a single thread. Now, to allow Cx3Dp to operate across multiple threads, processors and machines, we parcelate the model into multiple (very large numbers) simulation units that we call 'particles'. Each particle is an independent container embedded in the simulation layer that extends across the entire simulation system. Each particle is responsible for performing only local computation on its properties and will not calculate any properties of other particles. This means that all the data represented by a particle is immutable to all other particles; only the particle itself can change its data. In the case of a soma particle, this data includes its position, the concentrations of intracellular chemicals, the size of its soma and so on.

Each particle provides a range of simulation services for the model compartment that it contains. Typically this model compartment contains at least one space-node, and possibly also physical, and biological layers. The simulation particles are distributed across the available computational resources (machines, processors, and processing threads). Overall, the particles mask from the model its distribution across the simulation system.

The line between the different aspects and abstraction layers of the simulation, physical, biological and spatial has been more sharply drawn.

To be able to do simulations of the magnitude of one million cellular particles and more, it is important that the simulation is parallelizable. This parallelization is on two levels. The first is that Cx3Dp can use all of the processors available in the system on a single computer. Secondly, Cx3Dp can also make use of a multi-computer system in order to use not only the computational power but also the memory available. At each simulation step for each particle independently, all the necessary information for the next computational step is gathered. Then in the second step the computation is executed on all the particles, and in the third step all the computational results are applied to again reach a coherent state in which the time step has advanced by one. In this way, all of the particles could in theory be computed at the same time. This allows us to make a pool of work where each thread can just pick any as yet un-computed particles and execute the computation necessary. The theoretical limit of how parallel the system can be is given by the number of particles the whole simulation has. There are typically many thousands of particles in a simulation implying that even large arrays of processors could be made use of. This allows the full use of the processors available in state of the art systems.

6.3.2 3D Space and Diffusion

The overall simulation container tracks and maintains the organization of the model space. Tracking the relationships between neighbouring objects in the model space is crucial, because neighbouring particles interact via forces and chemical signals, as well as via simulation level information. In each simulation step the position and the size of nearly all the particles changes. Maintaining the spatial relationships between all the particles is computationally very expensive because the position of particles is relative. In order to perform an accurate simulation, all the particles need to know what the neighbouring particles are to compute the correct next steps of the simulation.

Cx3D uses a three dimensional Delaunay Triangulation to define spatial relationships between the space-nodes (not particles, which are a Cx3Dp concept) of the model. The explicit triangulation has the advantage that a node can immediately access its neighbors across existing edges, and does not need to perform a search. On the other hand, maintaining the triangulation has an overhead: the Delaunay criteria must be checked continually, and the triangulation adapted accordingly. And unfortunately, maintaining this triangulation violates the requirement that local changes in one part of the space should not influence distant parts. If the position of one node changes it might affect more than just its local vicinity. In some cases the effects of a single move might propagate widely, and sometimes involve the whole triangulation. These long range effects become extremely complex to manage when the triangulation is distributed across multiple computers, and the update problem is particularly intense for growth simulations where a large fraction of the space-nodes are changing position at each time-step. Therefore the space-model of Cx3Dp has been completely recast. The new version uses a compartmentalisation of space based on octrees [122]. An octree is a tree like data-structure where each node has exactly eight or zero children. It is used to compartmentalise 3D-space because if one cuts a cuboid in half along each dimension, exactly eight pieces of equal size result.

If we were to check whether each pair of particles in the simulation are neighbors, the computational demand would increase quadratically with the number of particles. Therefore we require a data structure that allows us to perform a nearest neighbor search as quickly as possible. Particles of the simulation must be addressable, therefore they must be managed by a data structure. We chose a data structure based on the 3D simulation space: an octree.

Octrees were used by Meagher [122] to represent 3D objects, whereas we use them to discretize space. In an octree, each box can be segmented into eight other boxes giving eight new boxes by cutting each box in half in all three dimensions. Each of these new boxes has the same size. In this way it is possible to create different spatial resolutions in different regions of space. This has the effect that each of the octree boxes manages approximately the same amount of particles. If the object count of one box becomes too high, that box can be divided into eight equally sized boxes. Each space node knows its spatial location, and can therefore quickly determine which octree box it is in. Therefore, when a given node must probe for neighbors, it need only search those boxes in its interaction radius.

Simulating the diffusion of multiple substances in the 3D model space is a difficult problem, and computationally expensive. To keep diffusion computationally tractable, the diffusion space must be quantized at a resolution that somehow matches the precision required by the cellular detection mechanisms, cellular density, etc. Previously, in Cx3D we chose to compute diffusion on the tetrahedron of the Delaunay triangulation, in which each node was a space-node. Now that tetrahedral quantization is no longer available, and the extracellular space has to be represented by a different data structure. Here again we can make use of the octree. In the diffusion process a single compartment is a leaf in the octree that handles the information and computation of the extracellular substances in a small part of the space (Figure 6.4 shows an illustration of the compartments). Each of these diffusional compartments is contained within a simulation particle. Using octrees, the resolution of the diffusion grid can be made dependent on the concentration of cellular compartments in space or can even be chosen to be completely regular if needed. It can be tuned for the needs of the modeler.

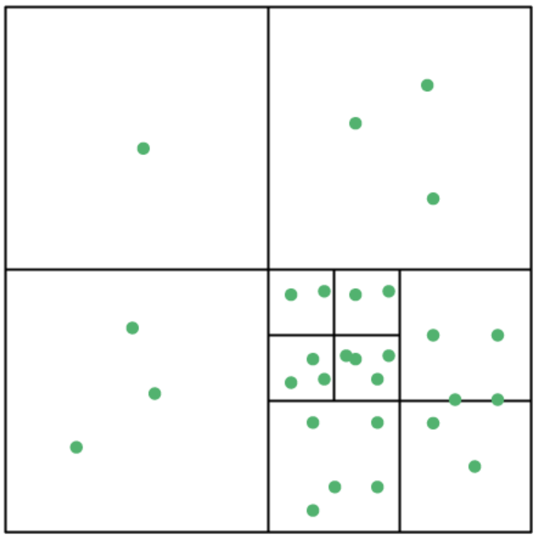


Figure 6.4. Space Model of Cx3Dp: In Cx3Dp, space is divided by an octree into spatial boxes. In this diagram, the 3D space is projected to two dimensions, for better visualization. The resolution of the spatial division is adjusted depending on the density of space-nodes in the region.

6.3.3 Threading Framework for Parallelization

Our scheme for distributing computational load across the available processors available on a single computer uses a modified producer-consumer principle. In our scenario, the producer is the Cx3Dp Scheduler (section 6.3.6). It generates tasks to be processed and adds them to a queue. The consumers are worker threads (workers). Workers dispatch the tasks from the queue and compute them. A typical task is computing the position advancement of a particle's space node.

In Cx3Dp there are two kinds of tasks to be performed, first the general number crunching that advances the model simulation (computing forces, biological models, positions etc.); and secondly system-maintenance tasks required to maintain the organization of the simulation level itself. Such tasks include the load-balancing of the system or work flow signals to keep the computation synchronized. System-maintenance work needs to be handled faster than the number crunching so as to maintain the flow of the simulation.

We created a dedicated worker type called 'simple worker' to provide this fast meta-processing. There can be many of these simple workers. The work that is fed to them typically consists of short tasks that are quickly executed and do not depend on other workers finishing where short response times are important. Typically these are the receiver threads that are responsible for receiving information from other connected computers. Each simple worker has its own work queue and any producer can fill the queue with simple work tasks to be processed. This work is distributed across the queues in a round robin fashion.

All the other work is done by complex workers. That is the heavy computational work such as calculating forces or calculating the influence of biological modules on the cells. There can be many complex workers in the system, typically as many as there are processors available. Each complex worker has its own queue where a work-producing thread places work packages that the worker consumes. Typically the producer-thread is the scheduler for complex work tasks. The distribution of work to the different complex and simple workers is performed by the 'work manager', which distributes work to the appropriate queues. The work manager can invoke a barrier at which the producer threads must wait until the current work is all finished. This barrier allows us to impose discrete computational steps in the simulation. In our simulations we use one simple worker and as many complex workers as cores are available on the computer.

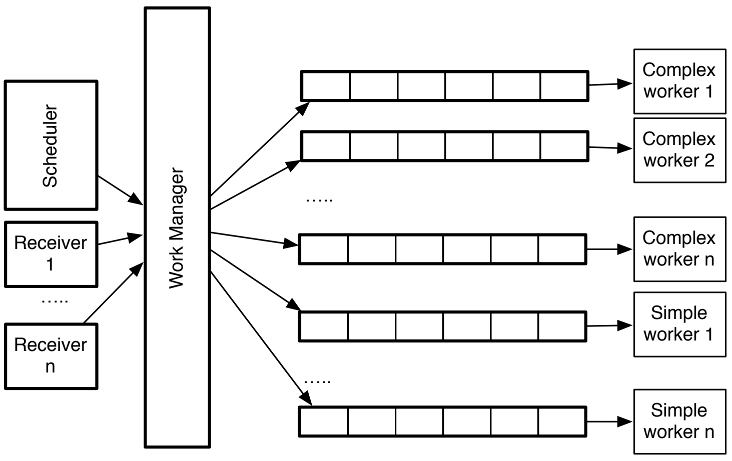


Figure 6.5. Work-Queue: The scheduler and receivers produce work that is then distributed by the work manager onto queues to be handled by the various worker threads. Work with low latency such as packages received by other computers in the network will typically be handled by the simple workers (low latency and very short duration work), whereas other tasks such as the heavy computation will be executed by complex workers.

6.3.4 Communication Framework

To allow communication between computers we require a framework that permits control of the complete communication flow between the different computers of the simulation system. In the spirit of our threading framework we used worker tasks as messages for communication between the computers. The protocol of the communication is as follows. The sending computer generates a new task of a certain type, the sender collects the necessary supporting data (and compresses it when large amounts of data are to be transmitted). The task object and its data are then serialized (converted into a stream of bits) and transmitted over the network to the receiver, which deserializes the data back into the task object, decompresses its data (if necessary), and posts the task to the appropriate local execution queue for the workers. These tasks are often a mixture of data and functions to be executed on the other machine. Compression is a crucial point to reduce the amount of data that must be communicated, and saves communication time. Compression is worthwhile because fast CPUs and high compression rates provide an advantage over the transmission of uncompressed data over the relatively slow communication networks.

Due to the overhead inherent in sending a package of data, it is preferable to send one big package rather than many small ones. Send and forget communication is preferable to handshake, because it does not hold up the process on the sending computer waiting for an acknowledgement. This means waiting for data only when it is absolutely unavoidable, such as when return data from the destination computers is required to compute the next time step.

The lack of handshake does not entail that data loss would lead to a corrupt simulation. All the data transfer is still monitored through the means of the communication protocols that the operating system provides to Java; this provides all the data loss recovery and data correction protocols needed.

There are simple and complex communication tasks. Simple tasks include mostly status messages such as the transfer of the current system status for reporting progress on the simulation and iteration of the time step, short tasks that indicate some control behavior such as save commands, pause commands or reporting commands. Complex communication tasks typically involve migration of parts of space to another computer, transfer of moving single particles that have moved to a space handled by another computer and transmission of stub information for generating marginal zones between computers (see section 6.3.5).

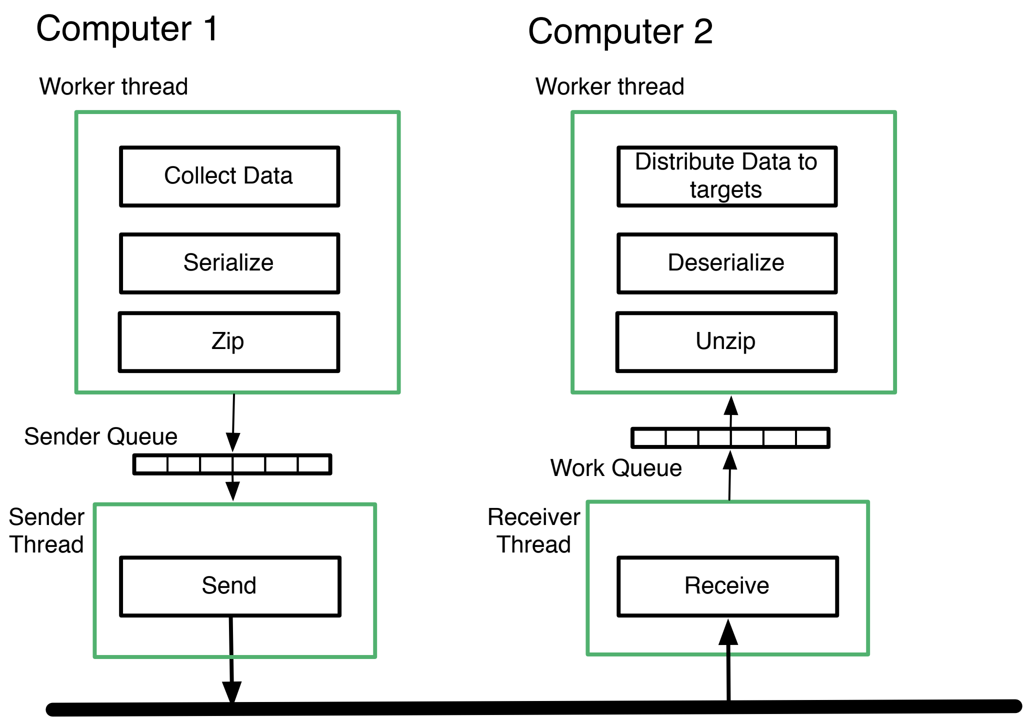


Figure 6.6. Inter computer communication: This figure shows the communication protocol between two computers. A worker thread collects the data to be sent, serializes it, compresses it and puts it in a queue to be sent. This queue is then accessed from a sender thread that sends the data in the queue. On the other end of the communication, there is another computer with a receiver thread. The receiver thread takes the message and puts it into the work queue. This work queue is processed by a worker thread which takes the received data, decompresses it, deserializes it and distributes the data to the target structures in Cx3Dp.

6.3.5 Margin Management

The simulation on a given computer needs to receive the result of computation of the previous step in the neighboring computers in order to advance the correct and seemless computation of the next step of the simulation. This means that each particle needs to know how its neighbors have changed since the last time-step. Therefore we need to know only about the state of those remote particles that interact with local particles. This requirement holds true for all particles.

Every computer handles a region of the model 3D space. These spaces are continous across machine boundaries. This means that a given computer needs to know which particles on neighboring machines (spaces) border its own space. The neighbors on the other hand need to know about the particles on the border of the space from this computer. This region where particles need to be sent to other computers is called the 'margin'. On the local computer particles in the margin region need to be collected and transmitted to the neighboring computers.

A special task object (such as the ones described in section 6.3.3) is provided with all the particles that need to be transferred for the margin. The task object uses a custom serialization procedure, only serializing the important data, such as physical manifestations (substances, mass, etc.), spatial properties (radius, position etc.) and biological properties (cell type, etc.). This data is then compressed and sent to the destination computer, where it is uncompressed and applied to that computer's marginal zone.

Using this tactic, each source computer only sends one margin package per simulation step to each of its two potential neighbors. Additionally, since this is a lot of data to be sent we select only the necessary particles and we transmit from each particle only the data that is needed by the local particles for their correct computation. We also compress this package to reduce the data to the minimum that we can send. It is a complicated task to do, but it is worth saving on the transfer cost since here the potential for losing time on communication is biggest. Figure 6.7 shows the margin relationships between computers and figure 6.8 shows a cartoon of the data distribution over time in a simulation.

6.3.6 Scheduler

The scheduler is the central organisation node in Cx3Dp. It leads the simulation through all the phases necessary to execute one simulation step. It dispatches work to the threading framework for execution. The scheduler is the one part of Cx3Dp that must be called at every time-step. There is one scheduler per machine. In order to execute the simulation, the process scheduler requires access to all of the structures representing the particles on the local system. This is done via a data structure which lists the particles on the local computer. This list is implemented as an array for fast access. Deleted entries are assigned nulls, while new entries are added only

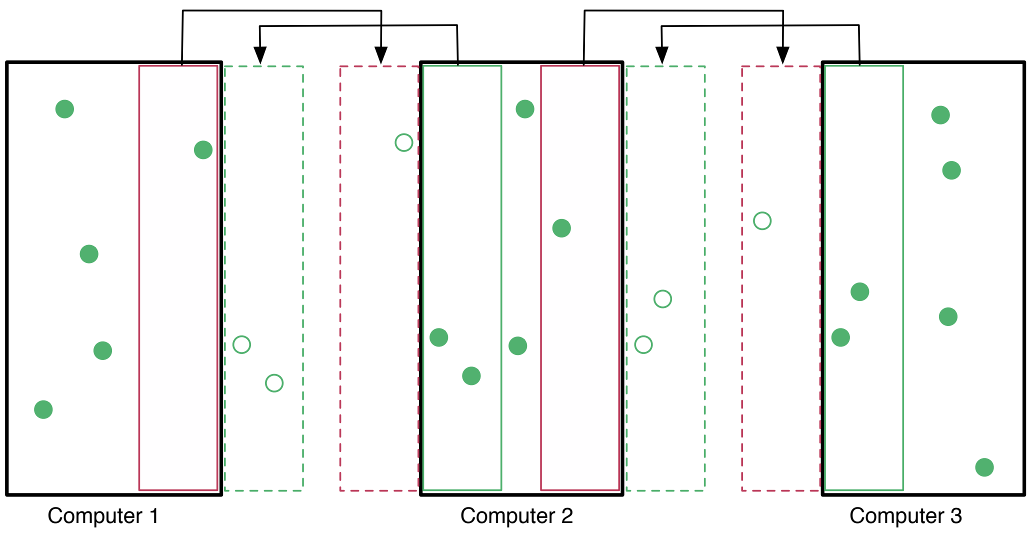


Figure 6.7. Space-margins for inter-computer communication: A computer system in which each computer sends parts of their local information as margins to their neighboring computers. The green filled dots represent particles, the green empty dots represent stub copies of the particles. The red/green boxes are the part of the computers space that has to be mirrored to the neighboring computer. The red/green-dotted boxes represents the mirrored marginal zones. The arrows indicate what is copied where. The green boxes are copied to the computers to the left (green dashed boxes) and are attached to their spatial models on the common boundary. The red boxes are copied to the right (red dashed boxes) and attached to the that computer's space model at the common boundary.

to the end of the list. The scheduler skips over the null entries and executes only the valid entries. Newly added particles are added at the end of the list and are only accessible in the next simulation step. This strategy obviates the costly rearrangements of the list. If the array fills up, it is doubled in size on each expansion. Such expansion occurs rather infrequently and does not much affect overall performance.

The schedulers generate tasks that are handled by the local workers, and they are responsible for the synchronization of the overall simulation's processing steps. Figure 6.9 shows the various operations a scheduler performs. First it generates the margins of the system for the neighboring computers. This step can be performed partly in parallel: the work manager waits for the local workers to complete their jobs by creating a barrier. Having gathered the necessary information to be sent to the neighboring computers, the information is then transmitted to them in one chunk. The local computer then waits for its neighbors to send their reciprocal information. When all the data is available on the local machine, the scheduler can procede to compute the next simulation step.

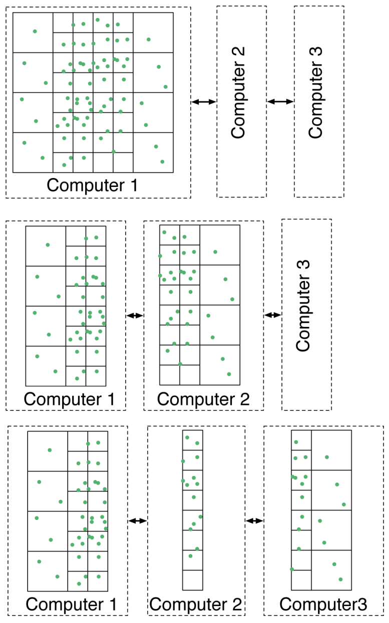


Figure 6.8. Distribution of space among computers: a typical simulation on three computers connected to each other. Top: The simulation is just about to get too big to be handled on Computer 1 and will 'diffuse' onto Computer 2. The simulation is split along planes perpendicular to the horizontal x-axis. The spatial boxes act as parts that can be shipped between computers. When this happens, the sending computer is relieved of the duty to compute this region of space. All the space-nodes (green dots) and diffusion boxes of the region are bound to the spatial box and are shipped off to the other computer along with it. Middle: The arrangement after the distribution step. Now Computers 1 and 2 need to exchange information at each simulation step in order to correctly compute neighboring relations between space-nodes and between diffusion boxes. Over time, the growth process will produce more and more space-nodes and diffusion boxes, increasing the computational requirement and changing the computation in terms of where in the space computational power is needed. Bottom: The system can then expand to more then two computers and slowly diffuse over the linear list of computers connected to the simulation. Communication (black arrows) is however only needed between neighboring computers in the list. Additional computers can simply be added to the simulation on the fly or at the start of the simulation, whenever the modeler sees fit to add more computational power to the system. Such computers will automatically be integrated into the simulation, and Cx3Dp will balance the computational tasks between the different computers such that all the computers in the system will be loaded approximately the same (in terms of execution time for a given time-step).

Each simulation step has three phases. (1) Gathering all the data needed for each particle (information about particle neighbors, parent and daughter nodes, diffusion partners, bonding partners, etc); (2) Calculating the next time step for each particle based on the information gathered in the previous phase; and (3) applying the calculated information locally to each particle. As shown in figure 6.9, each of these three phases is separated by a barrier. On the other hand, each 'Prefetch all neighbors', 'Calculate biology modules', 'Calculate physics / diffusion', and 'Apply local calculations' operation can be executed in parallel for each particle separately. Because there are usually far fewer cores available than there are particles, the particles must be grouped into chunks to make work packages (see figure 6.10). These chunks reduce the overhead of passing work to the workers.

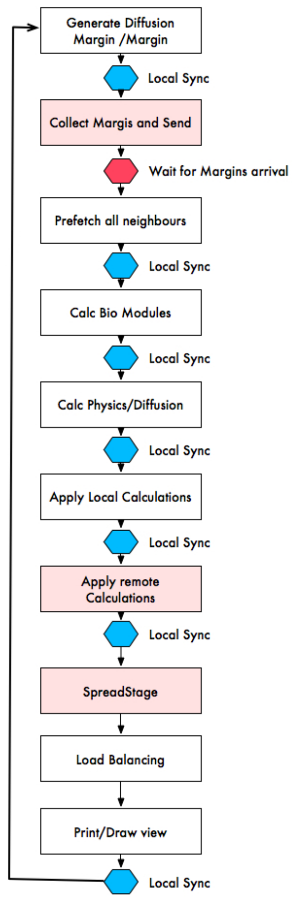


Figure 6.9. Scheduler of Cx3Dp: Each box indicates a separate operation to be performed. White boxes are tasks; red boxes indicate communication between neighboring computers. The blue hexagons are synchronization points where all worker threads are joined. The red hexagon is the synchronization point for this and the neighboring computers.

The scheduler separates in machine time all the 'Prefetch all neighbors', 'Calculate biology modules', 'Calculate physics / diffusion', and 'Apply local calculations' operations. This separation allows us to neglect read and write protection for the data, because reading and writing do not occur at the same time. Having executed the computational steps and applied all the

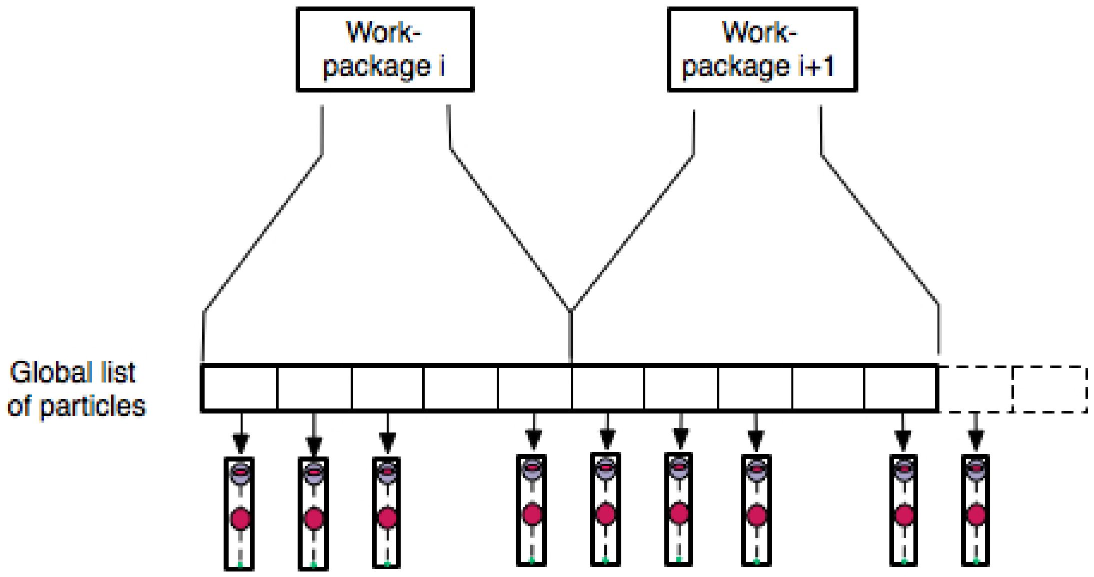


Figure 6.10. Work packages of particles: Work packages that are distributed to the Complex Workers are responsible for a sub-set of the list of particles. The particles (structures below the list in the diagram; with arrows) are stored in a computer wide accessible list; removing particles will cause null values (no arrow) that are ignored by the work packages. New particles are added to entries at the end of the list (dashed boxes).

results to the data, the simulation is back in a coherent state on this computer.

The scheduler then communicates which particles have changed spatial position such that they crossed computer boundaries, these particles are sent to the appropriate neighboring computer since they are no longer the responsibility of the local computer.

The scheduler now waits for all the neighboring computers to send it their particles that must now be handled on this computer. Having exchanged particles, the simulation is now coherent across all computers that are relevant to the calculations on this computer's scheduler. The scheduler then advances the simulation one time step and tells its neighbors in which time step it is. At this point load balancing can be executed if necessary. The topology of the system changes only slightly at each simulation step, and so the check for load balancing is performed only every n'th step. How many steps there are between balancing operations is defined by the user. In the simulations we present we used n values of either 49 or 101.

6.3.7 Visualization, Data-Logging and Persistence

Once the simulation is coherent and balanced, reporting such as view updates and exports can be performed on the local data. There is of course a need to report on and vizualize the results of huge simulations. The scheme for visualisation of the simulation has to be able to report the data from all threads on all computers in the simulation system, to give an accurate picture of the whole simulation.

We have implemented new reporting functionality that is able to export different aspects of the simulation. Each modeller can easily write their own set of exporters and/or vizualizations for the reporting functionality that they need. Each exporter or visualisation can be switched off completely if not needed to avoid overhead. This function has to be assessed carefully since reporting can be a bottleneck due to the fact that all the data has to be available to the reporting system.

The system can also be instructed to save its current state into files, so making a snapshot of the current simulation. This persistence permits the simulation to be stored, and possibly continued from this time step in future. Persistence also provides a fallback in case a machine fails during a simulation. The optional reporting and persistence completes one scheduler cycle, and one time step of the overall simulation.

Cx3Dp is intended to produce Big Data; we cannot expect to keep all data that is produced over time. This would exceed the capabilities of the computers doing the simulating. Bell reports on the fact [21] that big simulations can produce nearly unprocessable amounts of data. Therefore the reporting must be selective. This means that for each simulation, which aspects of the simulation need to be reported on in order to show the results of the simulation has to be carefully planned. The modeller must be aware that the right reporting is as important as the correctness if the simulated model in order to be able to present the desired results.

6.3.8 Load Balancing

Cx3Dp's ability to use a multi-computer system is achieved through the segmentation of space. The octree discretization is used along the x-axis of the simulation in order to distribute the space box-wise to all of the available computers. Using this discretization of space, each computer needs to communicate with only two neighboring computers.

Typically, the developmental simulation begins in just one computer. As development procedes, the model grows in its computational needs, and the simulation distributes itself onto an additional computer. If further additional computers are avaliable in the system then they can also accept work until all of the computational nodes are similarly loaded with work.

Thus the computational work begins on one computer and slowly 'diffuses' to the other computers that are available in the simulation system. This also means that computers can be assigned dynamically to the running simulation as the users of the system see fit. The number of computational nodes that can be deployed is limited only by practical considerations. And, because each computer communicates only with its neighbors in simulation space, there need be no centralized global control of the simulation. There is only local communication.

Conversely, simulation space cannot be pre-allocated to particular computers, because the computational needs of the simulation in a growth process change dynamically, are assigned dynamically and they cannot be easily predicted in advance.

Load balancing between the computers is also performed locally. Each computer compares its load to that of its neighbors in simulation space. If one of the computers is less busy then its neighbors it receives additional work from its busiest neighbor. A computer is considered less busy if it has to wait for the results of the neighboring computers longer than its neighbors. In fact, the computers are balancing which parts of the simulation space are computed by which computer. These parts are space boxes: each box contains space-nodes that are associated with particles and diffusion particles. On the busier machine, boxes on the border between two neighboring computers are selected for re-distribution. Their contents are removed from the local data structure and transferred to the destination computer. The boxes selected are transferred in one chunk, so minimizing the communication overhead.

We chose a strategy of load balancing which divides the simulation along one axis, which is analogous to simulating slices of dynamic thickness. We considered alternative strategies for minimizing communication between computers at each simulation step, i.e. which margins would have to be transferred in order to grant the neighboring computer the data that it requires to calculate the next step in the simulation. Our goal was to minimize the data sent between computers and to minimize the connection fan-out of each computer, so reducing the overhead in sending many packages of data and managing many sender and receiver threads. Simulations in Cx3Dp should therefore always be planned to be optimally distributable in the x-axis.

6.4 Results: Cx3Dp Applied

Cx3Dp was developed for exploring the development of neurons with their complex morphology, however it can also be used as a general purpose simulation environment for exploring growth processes in other tissues. Here we demonstrate the power of the parallel version of Cx3Dp using a number of characteristic cases, each chosen to exhibit some important aspect of Cx3Dp performance. For each simulation we use biological modules to control the behavior of the cell [213]. These biological modules can only act through their own cell, and each can only interact with its local environment. These restrictions mean that the models respect the locality property of cellular growth as observed in biology. Each simulation demonstrates a simple model that makes use of the Cx3Dp framework. The models can be downloaded from http://www.ini.uzh.ch/projects/cx3d/. All of our simulations have been configured to run with reporting turned off in order to minimize latencies in accessing mass storage.

6.4.1 Exponential Cell Growth

This growth process begins with one cell and then grows exponentially, according to just a few simple rules. This model is only for the purpose of illustrating the operation of Cx3Dp, and does not pretend to model any particular biological process. An initial cell is placed in Cx3Dp space. This cell is loaded with a biological module allowing it to divide and to grow. When the simulation is started, the biological module of the first cell begins to run and increases the size of the cell gradually. When a certain threshold of cell size is reached, the cell divides in a random plane, copying its own local biological module to its two daughter cells, and so these cells will repeat the growth and division behavior of their parent. This process leads to an exponentially growing mass of cells. When cells are formed through division, they must occupy space, and so they push their neighbors away. This means that there are local pressures on developing cells due to other cells in their neighborhood. Cell division is inhibited if the incident forces from the surrounding cells are too high. Cells color themselves yellow where the pressure threshold is exceeded. This exponential cell growth calls for exponential growth in computational resources. Here we simulate the growth until there are approximately one million cells spread across three computers (see figure 6.11).

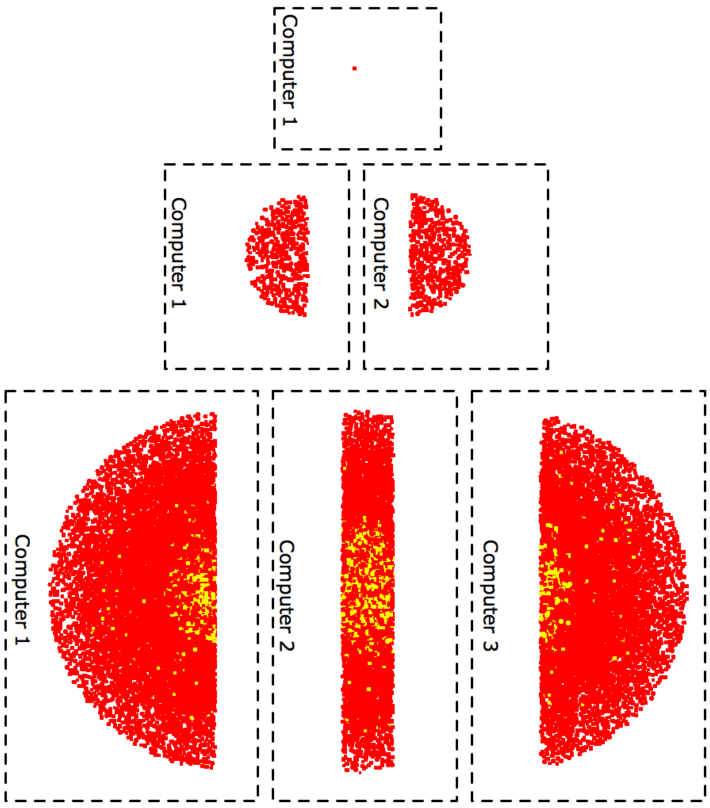


Figure 6.11. Growing tissue distributed over multiple computers: Three stages of the growth. Top: initial condition with a single cell, starting on one computer. Middle: A stage at which the tissue has expanded. The simulation has expanded to two computers. Bottom: Fully grown tissue consisting of one million simulated cells, spread over three computers. For better visibility only a slice through the simulation of a thickness of one cell is shown.

6.4.2 Comparison to Non-paralellized Cx3D Simulation

To confirm that the capabilities of the new Cx3Dp are the same as the old Cx3D, we demonstrate that a model that was constructed for the single threaded Cx3D can be used with little modification in the parallel version. We simulated the neuronal growth described in figure 9 of Zubler's paper [211]. The changes to the original code of the growth cone model from the Cx3D single threaded version were minimal. The overall growth model grows 11 excitatory neurons (gray) and one inhibitory (red) neuron in a dish like set up (pseudo-two dimensional). The outgrowth of the axonal structures is mediated by a random growth process. The neurons are placed at random x y coordinates in the dish and with a fixed z of 0. The thickness of the axon diminishes with length and branching points. It can be observed that the two dishes have a close resemblance to one another. Even though there are certain differences in how the forces are calculated and where they apply, we obtain qualitatively similar results in growth patterns in Cx3Dp, see figure 6.12.

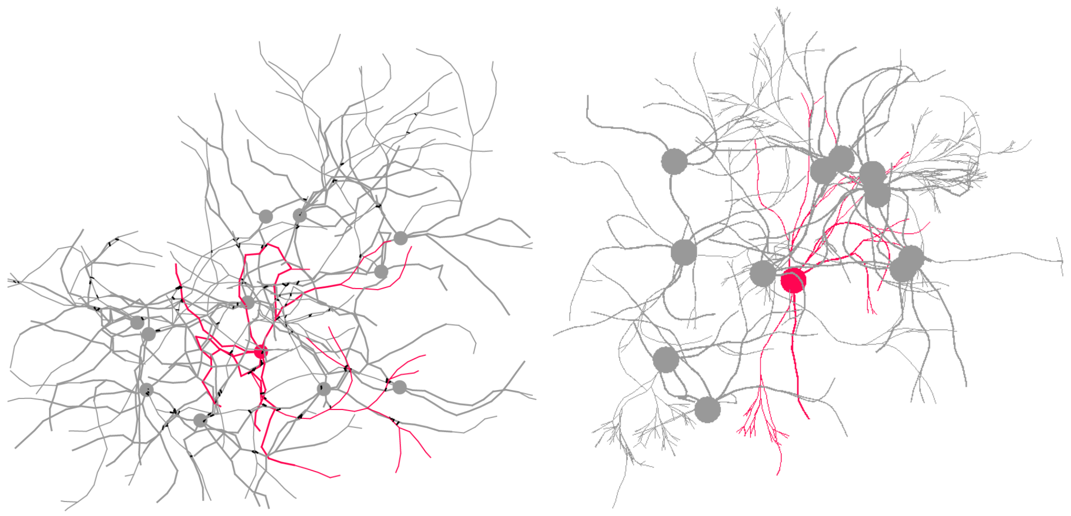


Figure 6.12. Two simulations to compare Cx3D and Cx3Dp. Left: Grown with the original version, Cx3D [211]. Right: Grown with the new parallel version, Cx3Dp.

6.4.3 Axons Crossing Computer Boundaries

To demonstrate the ability of Cx3Dp to grow cellular morphologies across machine boundaries we created a simple model in which growing axons cross computer boundaries. The model provides for two opposing planes of a thousand neurons each. Each of the neurons extends a neurite that grows towards the other plane following the initial outgrowth direction. Blue cells project a neurite in the direction of the green cells and vice versa. The axonal growth model is to elongate and bifurcate with a certain probability, and to stop the growth with a certain probability. After each bifurcation the probability of further bifurcation is increased. There are approximately 2000 cells in this system with a total of approximately 0.5 million cellular compartments. As can be seen in figure 6.13 the growth across computer boundaries works as expected.

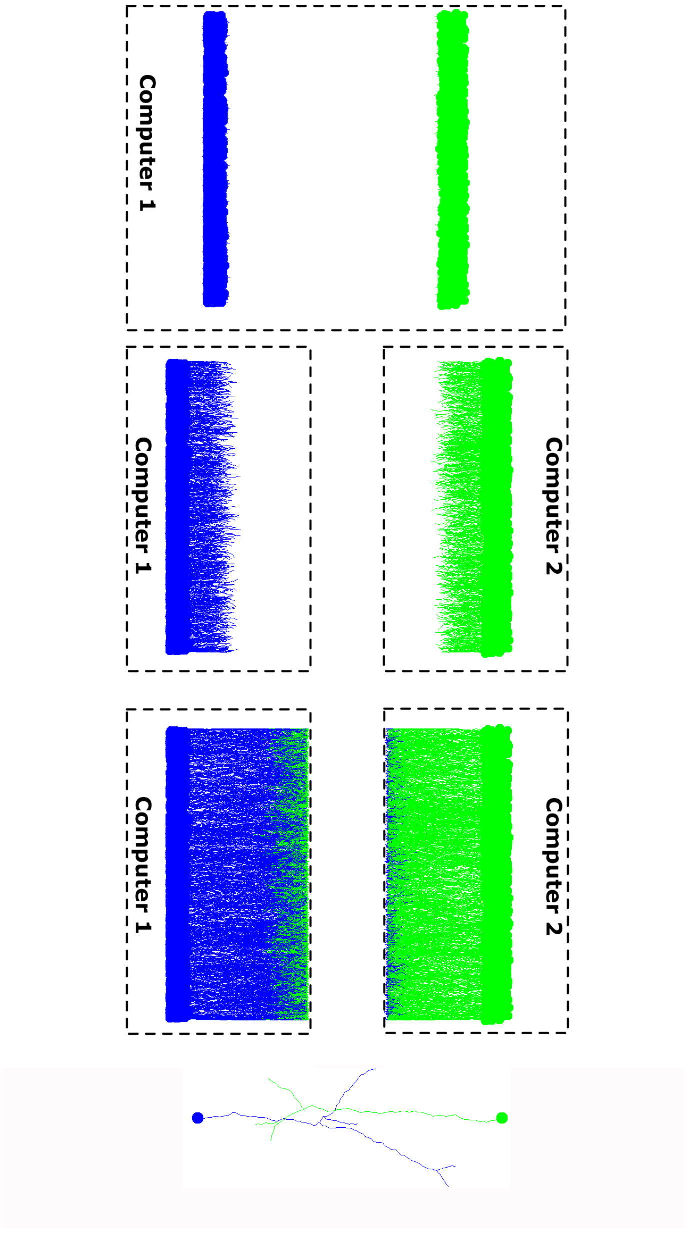


Figure 6.13. Inter-computer growth of neurities: Top box: The initial two planes of cells facing each other on one computer. Middle boxes: The two thousand cells growing their initial axons distributed over two computers. Bottom boxes: The neurites crossed the computer boundary and are meeting in the middle of the structure. At the very bottom, two of the opposing neurons are shown growing in each other's direction.

6.4.4 Diffusion of Extracellular Substances

Diffusion is a central aspect of Cx3Dp, a crucial feature of many growth models. We present here an example illustrating diffusion. The model specifies 671 randomly placed cells located in a $128 \times 128 \times 128$ micron space. The resolution of the diffusional octree was adjusted to provide a resolution of $1 \times 1 \times 1$ micron resulting in just over two million diffusional boxes. The diffusion constant was 320 m^2/s. The diffusible substance undergoes linear uptake (degradation) throughout the whole simulation volume with an uptake rate of 4 s^{-1}. The simulation step is $0.005s$. Steady state is reached after approximately ten modeled seconds (see figure 6.14). This simulation was executed on one computer having 24 cores.

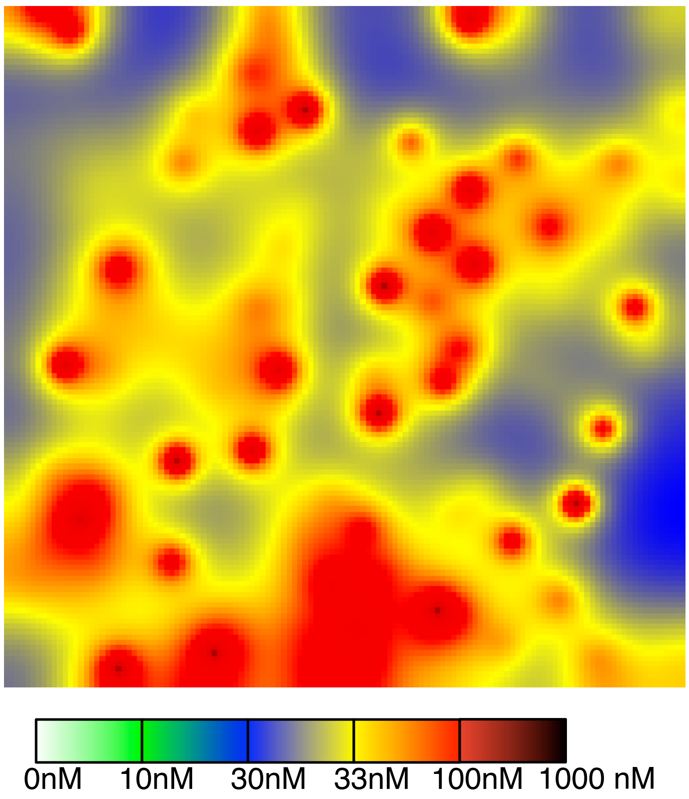


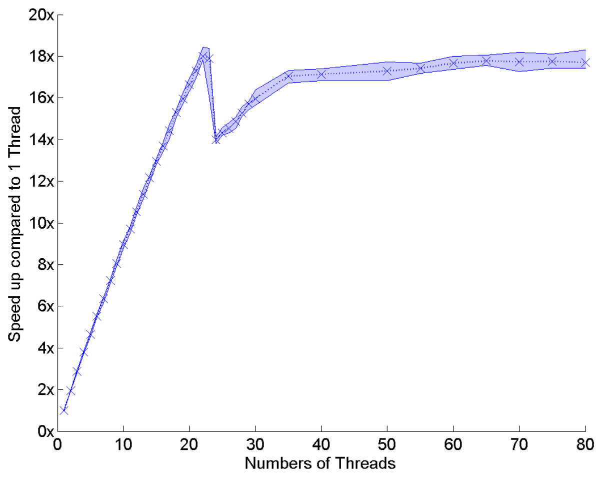
Figure 6.14. Diffusion simulation: Concentration levels of a chemical diffused through the Cx3Dp extracellular space. The colors indicate the concentration. We show a slice through the center of the space in the x-y plane. We can clearly distinguish radial diffusion patterns that overlap and build various shapes as would be expected with multiple emitting sources.

6.4.5 Scaling Behavior

We demonstrate the scalability of Cx3Dp using a simulation of a layer of cells, which regularly undergo division. The cells contain a biological module that cause them to migrate randomly in space, and so create a computational load. The exponentially increasing number of cells leads to exponentially increasing load.

At first the overhead for simulating the distributed system overwhelms the simplicity of the computations required for the cellular behavior. The question is, at what number of cells does one gain the advantages of having multiple threads and multiple computers? We use two configurations for the simulation. One compares the speed-up over the number of working threads, and the other compares the speed-up over the number computers. The advantages of numbers of threads cannot be directly related to the numbers of computers because the simulation configuration (in terms of communication, for example) must be modified slightly to operate with multiple computers. The number of cells in the multi-computer case is greater than in the single computer, varying number of threads case, so as to expose the advantages of splitting the simulation across multiple computers. The number of cells in the single computer, multi-threading case reaches 32,000, while in the multi-computer case it reaches 720,000.

Each simulation was repeated ten times to obtain a good average of the speed. Checks for load balancing were performed at the beginning of the simulations. Simulations were performed on four machines of the same type. Each has 64 GB of memory, 2 2GHz AMD Opteron 6168 series 12-core processors and is running Ubuntu version 10.04 'Lucid Lynx'. Hyper-threading was disabled. Figure 6.15 shows that speed-up scales very well with numbers of threads. The increase is linear until the processing speed saturates under the constraints of the single computer's architecture. The speed-up also scales linearly with the number of computers.



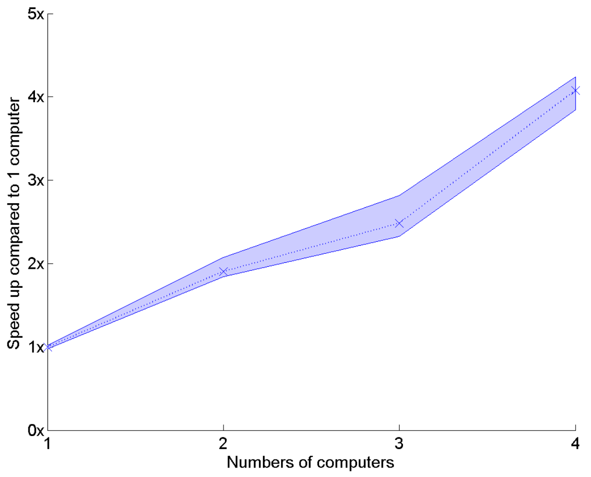


Figure 6.15. Scaling behaviour of Cx3Dp. Top: How Cx3Dp performance scales with the number of worker threads on a 24-core machine. We ran simulations using up to 120 worker threads. The blue area covers all the speeds measured, whereas the dotted blue line shows the interpolation of the average performance and the crosses show the actual measurements. For each simulation we measure the time of execution and compare it to the time used for one simulation. This yields a speed up factor. The performance drops at 23 threads because the scheduling overhead begins to slowdown the computation.
Bottom: The same type of measurement as above but with respect to numbers of computers used in the simulation. The speed-up measurement started as soon as the balancing strategy has successfully brought the workload to a stable state in order to avoid fluctuations in the measured data over the different trials. All of the computers in the simulation were working at the optimal point of 23 threads.

6.5 Discussion

Cx3Dp was created to enable modelers to perform large-scale simulations of neural tissue growth with millions of cells. There is no other software framework for simulation of neural development that provides for chemical and physical interactions with an environment as Cx3Dp does.

Cx3Dp provides a 3D space in which simulated cells can be placed, and the means to model local biological processes so that these cells can interact with each other. Cells can change their shape, grow axons and dendrites, divide, change their physical properties, and can secrete and detect chemicals intra- and extra-cellularly. All of these biological growth processes are based on a complex physical interaction system that is provided by Cx3Dp. Cells occupy space, exert forces on each other, and chemicals they produce diffuse intra- and extra-cellularly. Cx3Dp is well structured with layers of abstraction. The user of Cx3Dp writes models that interact with the interface of Cx3Dp that allows the control of biological processes such as cell division, migration and neurite outgrowth. This biological layer interacts automatically and hidden from the user with a physical layer that simulates all the physical processes such as intercellular forces, and intra- and extra-cellular diffusion. The physical layer uses the information provided by the spatial organisation layer to find out which cellular elements interact with each other. And finally, orthogonally to that, Cx3Dp implements a scheduler that executes the necessary biological, physical and spatial organisation processes in the right order.

The design and performance of Cx3Dp is a significant improvement over the original Cx3D and it also enables larger simulations. It solves rather elegantly the technical problems of parallelization of neuronal growth, and permits very large simulations to be run in multi-core, multi-computer environments. It is easy to add more computers to the system on the fly while the simulation is running. The computation spreads gradually over the computational landscape, depending on computational need. The performance of Cx3Dp scales well with the number of cores and networked computers, both in speed and in the size of simulation possible. It exploits efficiently the entire memory and processing power of the networked machines. Specific 'biological code' used to specify cell behavior in Cx3D runs in the new paralellized version with only minor modifications.

All parallelization is subject to Amdahl's law [3], which states that maximum speed-up scales hyperbolically with the number of cores available and the fraction of executing code which is parallelized. In the case of Cx3Dp we do not yet see the asymptote implied by Amdahl's law. Of course, there are a number of aspects of the execution of our simulation which we have not been able to completely characterize. We do not have a speed-up per core that is fully optimal. This may be due to spurious inter-locking in the code, the presence of barriers, and access to the memory that is shared between the computers. However, we can directly observe that the running Cx3Dp program usually spends well over 99.9% of the total execution time processing in parallel code, which promises good Amdahl behavior.

Our multi-computer speed-up scales well with the number of computers deployed, almost too well. This excellent performance can be explained by caching. If more of the data can be kept closer to the computation, the computational speed-up can appear to scale better than simply the increase in processing cycles offered by the number of computers in the system. We also observe large fluctuations in cycle time. This is to be expected because the computers must communicate intensively, and must sometimes wait for a neighbour's data before computing the next time step. This variability in the cycle time is probably due to variabilites in network speed and load balancing that varies between simulation trials. However, we must emphasize that the scale up in the memory domain provides the biggest advantage, because it permits ever larger simulations.

Nelson [136] and Modha [5] identified three challenges in multi-computer environments: memory, processing power, and communication between the computational nodes. Modha proposes an additional fourth challenge — the load balancing — which we see as given implicitly by the first three challenges. Additionally Bell indicates that the computation must happen close to the data in order for a system to scale [21].

The processing power challenge refers to the limited amount of computational cycles available to software on a system. Although the speedup of single core computers has saturated in recent years, the introduction of compact multiple core architectures has effectively sustained the Moore's law-like growth in processing power [132] [10]. Cx3Dp's threading framework is able to exploit these multi-core systems very well.

The memory challenge refers to the problem of optimizing the use of processing power by delivering data from memory to the CPU (and *vice versa*) as quickly as possible. Off-processor RAM speeds are typically an order of magnitude slower than processor speeds, and mass memory (such as disk) an order slower again than RAM. So the solution to the memory challenge involves minimising mass memory access, and maximising the amount of high speed CPU cache memory, as well as primary RAM. Fortunately the speed and the amount of memory available in commodity computers continues to rise every year, making ever larger simulations possible in reasonable time even on single machines. The ability to distribute the problem over a network of such fast RAM/processor units further improves the prospects for large scale simulation, as we have demonstrated using Cx3Dp's multi-computer framework (see figure 6.15). We respect Bell's proposition to keep the data close to the place where it is to be computed [21].

The communication challenge refers to the information that must be transmitted between the memory/processor units of the networked computers in order to maintain cohesion of the simulation across the entire system. Communication rates are typically very much slower than memory and CPU processing rates, and so the available communication bandwidth must be used as efficiently as possible. This means transmitting as little data, as sparsely, and as highly compressed (or encoded) as possible. In Cx3Dp we have devoted considerable effort to optimising inter-computer communication in these ways. In addition, we have chosen to distribute the simulation along only one spatial axis of the model, so reducing the number of neighbouring machines involved in communication. And, only the nodes in the margins of the spaces computed on each machine are communicated to the neighboring machines. These marginal nodes provide sufficient synchronisation of the distributed simulation. Other parellization schemes are no doubt possible, but we have demonstrated that our strategy for the Cx3D type of simulation scales excellently over multiple cores and machines.

Our aim is that Cx3Dp should be easily used on networks of relatively inexpensive standard computers, using only public domain software, and with a minimum of administrative overhead. For these reasons we chose to implement Cx3Dp in Java. Although Java is not the fastest language, it has the advantages of providing a relatively easy and secure environment for system developers as well as for the user modellers. Moreover, Java's cross platform compatiblity allows Cx3Dp to be run on almost any system, without the need for aditional specialized packages. Cx3Dp could probably be specialized for use on high performance supercomputers, but that has not been our aim.

There are other simulation tools for simulation of neural growth such as Netmorph [104], NeuGen [58] or L-Neuron [11] that focus specifically on the growth of axons and dendrites. There are simulation tools for single cells such as E-Cell [184] and for multi-cellular tissue growth CompuCell [93] [181] or Jeschke et al. [95]. However, none of these tools are designed for multiple cores and/or computers, and none address the physical interaction between neurons and their environment as Cx3Dp does. The need to simulate the electophysiology of large networks of neurons has led to a number of parallel implementations, for example Neuron [85], Genesis [17] and PCSIM [144]. However, these packages simulate only static topological structures and do not offer a simulation of the physical environment for growth and development of neurons. Our simulations are usually of growing structures that evolve over time. This means that one cannot predict what the overall best distribution of the computation is at the very beginning of the simulation. This has to be done during the course of the development of the simulated tissue. This requirement differs significantly from the needs of other simulation tools that exist. They often know how many computational particles they have and can distribute them across the computational landscape at the beginning of the simulation. This distribution of computation seems like a very complicated task, and hard to program, but the opposite is the case. If the programmer takes the local perspective of what a single cell has to do in the system and not how the global system has to behave, the programming of the cells is very straightforward;

no experience with concurrency is needed in order to implement models running in Cx3Dp. Taking the local perspective of a cell follows naturally from biology. In [213] we investigated what the minimal set of behaviours for cells is in order to program virtual cells such that they can behave as observed in biology. We call the resulting programming language G-Code. G-Code programs combined with Cx3Dp naturally scale to multi-threading and mutli-computer environments without the need for the modeller to explicitly write paralellizable code.

There exist state of the art frameworks that are optimized for interprocessor and interprocess communication, e.g. MPI and ProActive. MPI is defined as a standard [55]. Mostly it is implemented as a framework based on C/C++ and mostly used in cluster environments. Each MPI process is called a node where each node is usually run on one processor (lightweight threading in MPI is uncommon). Thus, although MPI relieves the developer of the responsibility of establishing connections between computers, it comes at the cost of monopolizing cores. We wanted to preserve local cores for productive simulation, and so running a separate process for each core (as MPI does) seemed like too much overhead. For the present we have chosen to avoid this overhead of MPI, and program our rather high-level communication scheme directly in Java. This decision brought the advantage that our simple communication framework neatly integrates with our scheme of tasks that distributes work among processors. However, it might be worthwhile casting our communication scheme onto the more standardized MPI framework in future.

ProActive is a framework that supports many communication schemes and many communication topologies are provided for [35]. At the center of ProActive are so-called ‘active objects’ that are completely thread safe (no other threads are allowed). In Cx3D we need to exchange a huge amount of data for all of the margins and parts of the simulation that move dynamically between the computers in the system. It was clear from the beginning of the Cx3Dp project that the communication load would be very high, and should be a focus of optimization. Therefore the profiling of communication is paramount. We started out using ProActive as a foundation for our parallelization, but soon discovered that it was difficult to detect and debug errors in the system, and moreover the overhead of ProActive slowed down the single threaded Cx3D by two orders of magnitude. We were therefore driven to our own less general, but highly optimized implementation. One should also bear in mind that using a standard framework that promises easy parallelization and deployment does not relieve the programmer of design decisions relating to the division of the simulation between computers, nor of the decisions of how to organize communication and what has to be communicated. In our case these aspects were the largest part of the work.

Tests of Cx3Dp on a four machine cluster have been successful, as we have shown here. This cluster is now being used for production runs that simulate the physical development of two areas of mouse cortex involving a quarter of a million neurons. It would be interesting in future to extend these simulations to clusters with many more computers, or perhaps to a supercomputer. We also plan to extend the functionality of Cx3Dp in various ways. One route will include electrophysiology, so that the developed neuronal circuits can express electrical processing. It is also widely believed that the electrical activity of maturing neurons influences their growth and connectivity. Therefore we are developing an additional module that can be added to the existing Cx3Dp framework that allows the simulation of simple compartmental electrophysiology. The major technical issue here is the combination of two very different timescales within a single simulation, because the electrophysiological behavior of neurons plays out on a much shorter time scale than growth processes. Another useful development path will be the improvement of reporting. Our current scheme could be greatly improved upon. For example, Java reflection and dynamic loading could be used to dynamically design, write, and load data exporters during the runtime of a simulation. Large simulations are inherently expensive in terms of computational and human resources. To be scientifically useful and economically justifiable, they should deliver high quality insights that compress the simulation data intelligently, and that go well beyond simple compression and pictorial representation of the large volumes of patterns that simulations generate. This will require a new generation of novel research tools for Cx3Dp that, for example, make use of simulation layer meta-data to detect and dynamically deploy intelligent reporters against improbable dynamics in the model.

Chapter 7

Discussion

In the discussion section of this thesis we will first assess whether we achieved the goals of the thesis and to what extent. Since we already discussed each chapter separately we will in the second part summarize the discussions and will provide an integrated discourse on the chapters.

7.1 Achievement of the Set Goals

In this section we show that the main three goals of the thesis have successfully been achieved. We will shortly highlight them one after the other:

1. *Simulation of Mouse Cortex Areas 3 and 6:* On the basis of biological findings on how the cortical development in mouse happens we created a model of a developing cortex of mouse area 3 and 6. Our model starts in a nearly unlabelled space, with only a few thousand progenitor cells of three types and constructs a cortical structure as observed in biology. The final structure contains of tens of different cell types and hundreds of thousands of cells, and two million cellular compartments. The model integrates cellular behaviour as described in the literature and all parts of the model are implemented in a biologically plausible way with the help of the G-Code programming language. Cx3Dp acts as the simulation tool to simulate this cortical structure. The whole cortical structure is guided only by the artificial genome inserted into the first few cells. Each cell is completely autonomous and is not guided by a global controller. The cells in the simulation autonomously take different roles and the cortex essentially self-constructs. This self-construction is guided by a gene regulatory network. A cortical developmental simulation of this complexity has not been achieved before and we have only been able to create it because we found the right abstraction level in G-Code.

2. *Extraction of Self-Construction Principles into an Engineering Technique:* Having achieved the simulation of corticogenesis from a few progenitor cells to a fully layered cortex as seen in biology, we wanted to abstract engineering principles that we could to program self-constructing systems. But to define a new engineering principle that is able to do self-construction we first needed a theoretical framework for how biology achieves self-construction in development. In chapter 4 we constructed a theory that goes from a single gene, via networks of genes, to behavior of cells, to cells interacting with each other and organizing themselves into structure. We were able to show that the work of Waddington [196] concerning the epigenetic landscape and the work of Zubler [213] on finding primitives to abstract cellular behavior fit perfectly into this theoretical framework. The important lesson learned from analyzing biological development that cells have a state and are able to replicate in symmetrical and asymmetrical ways is one of the key points for how biology controls development. In chapter 5 we showed that we have been able to derive an engineering technique called the Developmental Programming paradigm from this analysis. We explain in that chapter what a system has to fulfill in order to be programable such that it can exhibit self-constructing behavior. We identify categories of primitives that languages that follow the Developmental Programming paradigm must have. We explain the Developmental Programming paradigm on a general system, in relation to our simulation, the defined biological theoretical framework, and for the example of G-Code (a language following the paradigm). We show that languages that follow the Developmental Programming paradigm will be Turing complete and therefore universal in their computational power. We discuss the complexity of developing systems with relation to the complexity of the final object versus its genome. It becomes clear over the course of chapters 4 and 5 that such systems can organise themselves into a coherent global structure without a global controller only through local interactions between the containers (e.g. cells).

3. *Parallelization of Cx3D:* The goal for the Cx3D [211] simulation software was to make a parallel framework that takes advantage of multi-processor and multi-computer systems. Chapter 6 described how Cx3Dp was parallelized and the changes we applied to the system in order to make it efficiently use the available computational resources. Cx3Dp is designed to scale with the number of processors as well as the number of available computers. A clever balancing strategy ensures that all computers and processors in the system are used. It is possible to run Cx3Dp on any Java-capable operating system which includes Linux, Windows and Mac OSX, and all of them are potentially able to work together in a network of heterogeneous computers.

With this we have achieved the aims set out in section 1.2.

7.2 Conclusion to the Whole Thesis

In this thesis we have explored self-construction properties for the example of the mouse cortex areas 3 and 6. We have first given the background of the current knowledge about how the development of the cortex unfolds in time. Using G-Code and a new parallel version of Cx3D we created a model of the development that is in agreement with the literature, that recapitulates cortical development while respecting the biological properties and constraints. This developmental process by which the cortex essentially self-constructs has then been analysed and we show how biology is able to achieve self-construction from a theoretical point of view. Finally we generalise this biological self-construction to an engineering principle we call the Developmental Programming paradigm. The Developmental Programming paradigm defines what a system needs to have in order to be able to exhibit self-construction and it shows how such systems can be programmed. In the next few paragraphs we will discuss these results in more detail.

The mammalian brain is one of the most fascinating structures, especially the cortex that in human is thought to be the seat of intelligence. From very few precursor cells and cell types the cortical structure is generated with its billion of cells (in humans) and huge diversity of cell types. This process of construction is highly parallel, all the cells that are taking part in the development are autonomous and are not globally controlled. The only source of information the cells have at their disposal to assemble themselves into the cortex is the genome and their environment. The genome remains the same for all cells during the development. Only the activation of the different genetic expression patterns differs between the cells. With these limited sources of information, the cells are able to create a sophisticated structure such as the cortex. We created a model that is able to recapitulate this process following the same constraints: no global control, autonomous cells and one genome.

We model the cortical development in our simulation environment Cx3Dp and use our in-house developed programming language G-Code to compose the model. Our model starts at the beginning of cortical development of mouse at around E10 [171], at this stage of development certain environmental conditions are already established for the embryo. The model starts off with three types of cells, and two gradients all in the tangential plane. These three sources are the precursors to the cortical plate, precursors to the interneurons originating from the ganglionic eminence and neural cells from the thalamus. We do not model the development of the thalamus or the ganglionic eminence but only their role in the construction of the emerging cortical plate. Except for these starting conditions the space in our simulation is unlabeled.

When the simulation commences, the cortical precursors start to divide to create new cells that build the pre-plate, the pre-plate is then split into marginal zone and sub-plate by the emerging cortical plate. The precursors start to become radial glial cells, generate the ventricular zone cells, then subventricular zone cells and produce over this time the different layers of the cortical plate During this phase the ganglionic eminence produces inhibitory neurons that tangentially migrate and eventually turn up radially into the cortex in order to match up with the right layers of the cortical plate. Additionally the thalamic cells send out axonal fibers that connect to the appropriate layers of the cortex. (See video http://youtu.be/pBqZ8SouWdQ 'Development and 3D Rotation of Selected Neurons' for a visualization of this process.) In the model we integrated many detailed steps that are necessary for the cortical development to work, for example radial migration, radial glial fibers, inter-laminar axonal growth and apoptosis of misplaced cells. We found that even though the literature is very detailed on cortical development there are still a few mechanisms we have not found any information about but that were absolutely crucial for the success of the modeled development. For example the mechanism for distributing the interneurons in the tangential plane has not been described. In these cases we took the initiative and created a hypothesis for how these mechanisms could work. In the case of the tangential distribution we assume an addressing mechanism that maps the interneurons born in the ganglionic eminence to the cortical plate. This addressing mechanism depends on two gradients, one in the ganglionic eminence that is read at the birth of an interneuron just before its tangential journey begins and another gradient in the cortical plate that is read by the migrating cell and compared to the concentration at the birthplace of the cell. Through a mechanism such as this we could create a tangential distribution of the intereurons. We do not claim that our suggestions for this tangential addressing or other mechanisms are correct, but mechanisms similar to the proposed ones must exist in order for the cortical construction to work. We are however sure that the implemented mechanisms are at least biologically plausible. Our addressing mechanism for example can be seen as a hypothesis that can be tested experimentally in a real mouse model with cellular labeling.

The basis of the model is the Gene Regulatory Network. The gene regulatory network generates all the different types of cells in the right order at the right time. It is able to generate this huge amount and variety of differently behaving cells with two important mechanisms: symmetrical and asymmetrical division. The gene regulatory network is activated in all the precursor cells at the beginning of the simulation. In each cell at its birth the gene regulatory network activates behavioral G-machines corresponding to the cell's type. These behavioral G-Machines are simple sequential programs executed in each cell separately. Each G-Machine on its own does not elicit interesting cellular behavior. Only in association with other cells and their G-Machines does the development of cortex emerge.

Our model is restricted by the simulated physical properties that exist in Cx3Dp and the biological restrictions that the G-Code language imposes. Additionally our model is restricted by the findings in the literature and therefore the knowledge about the cellular behavior during cortical development. G-Code was powerful enough to model all the documented behaviors without exception.

The model presented here is not the first model of development. There have been many models of development ranging from neural tube formation [38] and regionalization of the neural tube [101] [117], to cortical growth [211] and many other models that model aspects of cortical development. (A more detailed discussion of the literature can be found in the conclusion to chapter 3.) The model presented here is however the first to recapitulate cortical construction to that level of detail. It starts in a nearly unlabeled space and all the cells are completely autonomous and not globally controlled. The autonomy of the cells allowed us to separate the model into sub-compartments and simulate the development in a highly parallel fashion in our newly developed simulation framework Cx3Dp, therefore we were able to scale the simulation to hundreds of thousands of cells and millions of cellular compartments.

Our model is far from complete, but it is a good starting point for further cortical development simulations. First, electrophysiology is missing, after having grown all of these complicated neuronal connections one could use these to create functional networks out of it. Another direction to push the model would be to simulate the developmental process from even earlier stages, possibly from one initial cell. Or to move towards a developmental model of higher mammalian cortices such as monkey cortex. Additionally the model could serve as a starting point for showing how biology implements on a molecular level the behavior we simulate more abstractly with G-machines. Models and simulations such as these can help biologists to find theories about development. These theories could then be tested first on the *in silico* model and then confirmed by doing actual biological experiments. This could help speed up the process of finding new interesting mechanisms.

The power of the development is given by the autonomy of the cells. Through their membranes, cells are only able to react to specific simple signals in the environment that can then start a cascade of more complex processes internally. The membrane encapsulates the cell's internal environment and its function. This allows for a very localized control of development where each cell controls very specifically a small part of the overall system. In engineering it is a sign of a good product if the functionality is well encapsulated and the communication between the different parts of the system is simple and well defined. In development we find these principles as well, simple extracellular signals can encourage complex intracellular processes that lead to cell behavior. We see in our model and the corresponding biology that the communication between the different cells is very sparse, this fact decouples the cells from one another and allows for the high degree of parallelization we observe.

Another interesting observation is that the earlier born cells provide the scaffold for their successors. They label the space with molecular signals and mechanical boundaries, so enabling their successors to orient themselves. Like the preplate does for the emerging cortical plate. This is also a technique known from engineering using scaffolds for the construction of ever larger, more complex products.

That our model shows emergent complex behavior arising out of very simple and small behavioral programs is due to the level of abstraction we chose. We program the G-machines that define the behaviors of the simulated cells in G-Code. G-Code is constructed such that all the primitives offered to the programmer are biologically plausible, G-Code itself is therefore constructed from the bottom up not with the behavior but with the actual biological implementation in mind. G-Code itself can then be taken by a programmer and used as a tool to construct a model top-down, using the primitives to construct an observed behavior without having to look at the detailed biology. However G-Machines (behaviors) written in G-code remain biologically plausible because of how G-Code was designed. This abstraction gives the modeler a powerful tool that allows for a separation of the overall problem into simpler sub-problems. In our case it allowed us to take apart the complex self-construction process of the cortex into sub-tasks for each type of cell, sub-tasks such as radial migration or lateral inhibition. The autonomous cells have different G-machines activated at their birth. Through the communication protocols between the G-machines running in these cells, complex emergent behavior can be controllably programmed into the self-constructing system without the need for a global controller.

Once we had implemented an example of a self-constructing structure we were in the position to give a theoretical framework that abstracts biology in such a way that the framework allows us to explain the developmental process with basic rules (see chapter 4). The framework starts by recapitulating the transcription of single genes into proteins and that these genes can be arranged in a sequence as the genome and that they influence each other in their expression. The interaction of genes builds a Gene Regulatory Network (GRN) that has been intensely analyzed by Pfister in her thesis [146]. The GRNs implement a function in a large dimensional space called the GRN landscape. The GRN landscape is fixed for each genome. Only if the genome changes do the dimensions change. We explained how cells operate in this GRN landscape and that each point in that space corresponds to a single protein expression pattern that a cell can have and is therefore a state of the cell. Regions in this state space can be understood as the classical cell types. And cells operating in that space can transition between different states by following the gradient of the Gene Regulatory Network function. In this way cells can transition between different cell types. The GRN landscape has a close relation to the epigenetic landscape of Waddington but is not exactly the same. Whereas Waddington's epigenetic landscape [196] really is two dimensional, we only projected our landscape onto two dimensions for visualization purposes, but in truth it is extremely high-dimensional, for each gene there is one dimension. Even though Waddington describes the high dimensionality he does not phrase his visualizations that way. Furthermore, Waddington does not exactly explain how multiple cells can take different paths in this landscape but only says that this choosing of different paths happens somehow. There Waddington stops with his explanation of the epigenetic landscape. We on the other hand elaborated on the behavior of cells based on the GRN landscape. The importance of replication has been shown. Dividing cells have to distribute their contents to their offspring. Through division the daughter cells perform a jump in the GRN landscape since their gene expression pattern alters abruptly. Symmetrical and asymmetrical division allows cells to take different paths in the GRN landscape. With that the cells will end up in different local minima of the gene regulatory network function and eventually in different fully differentiated states.

The behavior of single cells is shown to be a direct effect of the gene regulatory network that might in a certain space activate proteins that code for behavioral function. We show that these proteins can be combined into ensembles of proteins coding for a particular function and that certain proteins can be reused for other functions. We name these behavioral functions machines. This reusability leads to a component composite principle of recombination of machines, where for example replication is one of the behaviors that can be exhibited. It has been described by Zubler [213] how one can think of these assemblies of proteins in terms of primitives. He describes how these primitives are biologically plausible and implementable by proteins. He showed that it is possible to abstract more complex behavior as a recombination of these primitives. We show that it is possible for cells to reach states in the GRN landscape where genes get actively transcribed that contribute to building these primitives and therefore also to the building of the behavioral machines. Since each of the cells in the developing organism can potentially reach different states through asymmetrical division, an organism with billions of cells can end up with thousands of cell types Through replication this process of proliferation can potentially follow exponential growth. In a further step we discussed how cells can communicate and organize themselves in a decentralized fashion and build sophisticated structures. The communication between cells is a crucial point for their ability to organize themselves.

This process of development looks very complex since many of the cells seem to magically know where to go in the organism and all of this happens in parallel. We touched on this complexity by explaining the flow of information in the system. The end state of the developed organism can be seen as an unfolding of the code encrypted in the DNA. The DNA and the environment with its rules and preconditions for development is enough to describe reproducibly the development of the organism. Therefore the DNA and the environment present at the beginning of the developmental process can be seen as the program and the final structure as the computed output. The organism is therefore fully compressible to its DNA and the environmental rules, implying that the final organism is of the same complexity as the code it is described by, according to Kolmogorov complexity [105]. With this we have shown a framework for how to understand the developmental process from a single gene to a fully assembled organism. Of course this framework is a first attempt and might not hold true down to every detail of molecular biology but it is a good first attempt at a framework for how to think about development.

Having explained the biological framework, this left us at a point where we understand development as a process. This allowed us to make one step further to abstract away the principles of construction from the biology into an engineering principle (see chapter 5). We describe the new Developmental Programming paradigm that defines how a system has to be designed in order to be programmed to exhibit developmental-like construction, also known as self-construction. We explain the Developmental Programming paradigm with the example of G-Code, the language that was developed in order to simulate the cellular behavior in Cx3Dp [213]. Each such system needs an environment, a container (cell like structure) and machines that govern the behavior of the containers, where a container is a localized structure in the environment and the machines are the agents controlling the container and interacting with the environment through the container. This design does not allow for a global controller but only for many local ones nested in the containers. The machines are based on primitives that can be assigned to one of eight different categories that range from inter-container communication (read/write/filter) to behavioral instantiation (activate/terminate) to proliferation of containers (replicate/eliminate) and for displacement in the environment. One of the most important categories is the replication category. Replication must copy the container of the machine that is executing a primitive of this category. But replicate must define the new state of the emerging two containers that do not necessarily need to be the same following the natural process of replication that can also put cells after a division in completely different states. This process is very important since it allows for very controlled symmetry breaking in the developing or self-constructing system. With a system of this type, it is ensured that a programmer of the system is able to make use of the advantages of self-construction. Programs written in a language following the Developmental Programming paradigm are automatically fully parallel executable, locally controlled rather then globally, allow for self-repair, have no single point of failure, self-construct without needing interaction with a human, and exhibit potentially exponential growth in computing power. Especially the last point could be of great interest. If a highly parallel computational substrate were available, an implementation of the Developmental Programming paradigm would allow for fast spreading of the computation. Or assuming a system with self-replicating containers exists and implements the Developmental Programming paradigm, the system could create its own computing substrate and thereby allow for exponentially growing computational power. By defining a new programming paradigm we followed the suggestions of Roth [159] who asks for a higher level language for doing self-construction. With this paradigm we provided a description on how such languages have to be implemented. In addition our work is comparable to the work of von Neumann in which he gave the first description of a self-replicating universal machine [194]. He defines all the steps necessary to achieve self-replication but then does not explain how to organize the self-replicated entities such that they work towards a common global goal.

Languages following the Developmental Programming paradigm are Turing complete, like languages of the functional and the object-oriented paradigms. The differences of these two paradigms to the Developmental Programming paradigm have been pointed out. Whereas in object-orientation the object is the central point of abstraction, in Developmental Programming it is the machine or thee behavior of the container. All the objects in the Developmental Programming paradigm are already given, they are the containers. Only the emerging structure or sub-structure can be seen as another object and this is completely built out of containers. As Roth already mentioned, a need for patterns arises for the Developmental Programming paradigm, much like the patterns for object-oriented programming [68]. These will have to be defined in the future and are not part of this thesis, but a few examples were given of what kinds of patterns could be worked out. With the current developments in synthetic biology where we are on the verge of being able to program cells, the need for programming in a developmental oriented way will arise sooner or later [71]. Additionally current chip designs go in the direction of having more and more cores that work in parallel and become more like decentralized networks, as is the case in the SpiNNaker project currently [102]. These decentralized networks of cores will have to be instructed in a way that is not globally controlled, which creates an opportunity for using the Developmental Programming paradigm because it achieves an organized structure in a decentralized way. Of course there are also the obvious candidates such as nano-robots, large sensor networks and smart dust that need to be instructed such that the individual elements work together as a whole. Achieving global behavior through local interaction is preferable rather than having a global controller that controls each entity separately. Furthermore, as soon as it is possible to have any kind of physically self-replicating programmable entities, the developmental programming scheme is the obvious choice for creating a construction process. Developmental programing can be of use even today: large clusters, clouds and other large computer networks could be integrated into a virtual environment that is built such that it supports the Developmental Programming paradigm. Developmental programing would allow for the integration of these networks into a big artificial ‘organism’ with local communication only, no global control, and completely distributed, but the system would still exhibit a coherent global behavior.

Lastly we have created a parallel implementation of Cx3D [211] that is called Cx3Dp (see chapter 6). We have shown the simulation of a self-constructing cortex with more than two million compartments (see chapter 3). To simulate such large systems would not have been possible without the extension of the Cx3D software package to a fully parallel system. Cx3D was a single-threaded simulator of tissue growth that was capable of growing tissue on the size of thousands of neurons. In order to be able to simulate large scale tissue growth such as our model of cortical growth that we describe in this thesis we had to extend Cx3D to run in parallel to handle such big systems in reasonable time. To achieve this goal the basic structure of Cx3D was reimplemented such that all the computation can be executed on the cellular compartments separately. For that, the underlying spatial structure that gives the neighbor relationships of the cell compartments in space that was implemented using a Delaunay triangulation had to be reimplemented as an octree. The octree scales much better for multi-computer systems. We explain what we see to be the main challenges in parallel programming, namely the challenge of processing power, the challenge of memory access and the challenge of communication. The challenge of processing power is to use as many CPU cycles as possible for the actual simulation while wasting as few cycles as possible on waiting. In an ideal case, all the CPUs would be maximally busy with processing the simulation. We showed that using a concurrent system, Cx3Dp can exploit the multi-core architectures that are expected to increase further the numbers of cores per processor. The memory challenge is about having the data as close as possible to its processing core, with current trends RAM will grow even further in the future and we would expect to have ever larger amounts of memory available on local machines. If the data is on another machine or on a slow medium like the hard drive, CPU cycles will be lost by waiting for the data to be transferred to the processing core. The last challenge is communication between multi-computer nodes. Communication takes time, ideally as little as possible should be transferred between the different computer nodes. If the communication of shared data between computer nodes is too slow, CPU cycles will be lost on waiting for the data to arrive at the processing cores.

Cx3Dp was compared to state of the art parallel systems such as Neuron [85], Gadget-2 [175] and NAMD [136] that are not related to tissue growth but face the same challenges of how to parallelize simulations. For nearly all parallel systems the MPI framework [55] is used, or a similar framework such as ProActive [35]. We deliberately used none of these systems because they posed serious limitations for Cx3Dp. For example with MPI we would need to create a separate process for each core and that would have meant a lot of overhead (JVM, shipping memory between the processes etc.). We prefer the more light-weight threads that all have access to the same memory and can all be run in the same process. In general we wanted to have the maximal control over the system to define optimal connection schemes, for debugging and to be more independent of third-party software. Therefore we did not rely on any other framework than Java itself for Cx3Dp. It was explained that Cx3Dp uses a scheduling scheme that works in three phases and overcomes with this the need for interlocking between the threads to access data. First a data gathering phase is performed for each thread (read), then the future values of the variables are computed (compute) and lastly these values are applied to the data in the system (write). With this three phase protocol read and write do not overlap and that saves a lot of thread interlocking problems. Finally we explain that we can with this scheme reach a parallelizable degree of execution in the 99.9% range allowing therefore nearly all the computationally heavy code to run in parallel. In Cx3Dp the computational tasks are spread to all of the computers dynamically. This distribution of computational tasks is done via a balancing scheme that checks how busy a computer is compared to its neighbor. The balancing will be done locally between the computers for the complete system. It can be viewed as if the computational tasks diffuse one dimensionally between the computers. The balancing strategy directly addresses the three challenges of parallel computing. Following this strategy lets the computers keep an equilibrium of speed so that all of them finish one computation step of the system in nearly the same time, minimizing therefore the waiting time between computers. The balancing automatically keeps the data that has to be computed close to the processors that compute it. The balancing addresses the processing power and the memory challenges. Further communication is only done with neighboring computers (maximally two) and we only communicate the necessary data to the neighboring computers. This strategy addresses therefore the communication challenge. In the future it would be interesting to see how Cx3Dp runs on a cluster machine, another possibility still remaining is to include electrophysiology and to further optimize the data structures of Cx3Dp in order to achieve faster simulations. In terms of reporting, we imagine more dynamical ways of loading reporting structures into the running systems that can be loaded on demand. If all aspects of the simulation were to be reported on at once, the reporting would take even longer than the simulation step itself.

With this we conclude this thesis. We have shown a growth model for mouse cortex areas 3 and 6 that follows the known literature and the biophysical constraints of development, and that can construct a cortex in a self-constructing way (see chatper 3). We deduced from this modelling experience a theory for how biology can achieve self-construction (see chapter 4). From this we deduced a theory of self-construction as an engineering technique that we call the Developmental Programming paradigm (see chapter 5). Lastly we gave an overview of the tool Cx3Dp that enabled us to simulate the cortical model on the required scale (see chapter 6).

Acknowledgements

I would like to thank all the people that supported me professionally during my PhD. For the intense and sometimes passionate discussions to make my ideas more crystallised. And for reading and correcting parts of this thesis. Especially my supervisor Rodney Douglas but also Kevan Martin, Matthew Cook, Michael Pfeiffer, Fred Zubler, Roman Bauer, Sabina Pfister, Gabriela Michel, Nuno Costa, Andreas Steimer, Isabelle Spühler, Dennis Göhlsdorf and Adrian Whatley. I would like to thank Henry Kennedy, Collette Dehay, Marion Bétizeau and Elodie Gautier from Inserm Lyon for discussing my biological ideas with me.

I would also like to thank my examiners, Christoph von der Malsburg and David Willshaw, who have gone to the trouble of reading and evaluating this thesis.

Finally I would like to thank my family and friends who supported me, fed me and distracted me: Marion Bétizeau, Brigitta, Pascal, Cornelia, Marta Hauri, Hedi Baumann, Hugo Barreiro, Jasmin Gisisger, André 'Ron' Schmidt, Andreas Lienhard, Sandra Meier, Raffael Koller, Barbara Furrer, Natalja Früh, Stephanie Hess, Mirjam De Ventura, Nina Baldinger and Michaela Baldinger.

This work was supported by the EU grant 216593 'SECO'.

Appendix A

The New G-Code

In chapter 2 we described the original G-Code. The actual implementation was done in Java. In the original implementation of G-Code, the primitives are each implemented as an objects. Each primitive has input and output ports that read in values from other primitives and write out values to other primitives. These primitives (objects) can be combined with links that are unidirectional, these are basically single-valued containers that forward a value output by one primitive to the input of another. Each primitive could have certain properties which could be set, for example in the primitive replicate, one could define which new machines are to be started in the new cell after replication. The machine was also a class that could contain multiple of these primitives and links, both had to be registered with the machine. This is depicted in figure A.1. Machines could be reused as primitives in other machines. The G-Code language was written for Cx3D.

A.1 Issues with the Old G-Code Implementation

There where a few issues with the old implementation. The coding of machines was done either directly in Java or in an XML form. Both ways of coding were very inconvenient and led to enormous files. There was no checking of data types, no autocompletion for fast writing and when giving wrong arguments to a primitive in the code it was not always corrected. There was no support from an integrated development environment like Eclipse that is really essential for writing code fast. Using objects for the primitives also had another disadvantage on debugging as it was hard to find the source of an error since type checking was not done. So errors that could have been seen at the compilation stage only showed up at runtime, which was a very unpleasant experience for the G-Code programmer. The objects had in addition the runtime problem of being extremely slow because of the immense call depth of functions to communicate

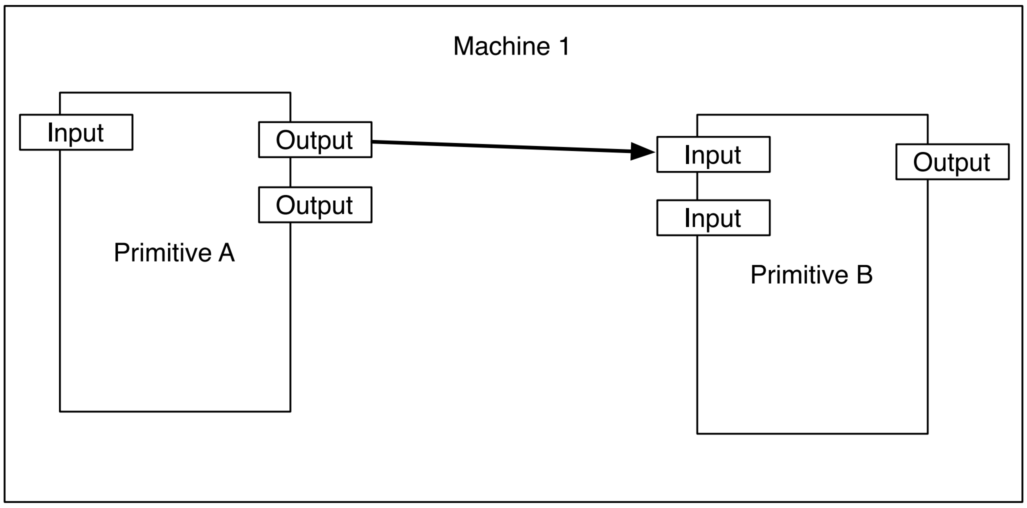


Figure A.1. A typical machine that has two primitives with input and output ports. Primitive A is connected to primitive B and feeds information to primitive B. This is how the old G-Code was directly implemented. Each of the primitives, input/output ports and links between the ports were Java objects.

between objects. Since machines could run in the soma and in the growth cones there were two different implementations intermixed in one primitive. For example move would have to execute completely different code for growth cones or somas. Some of the primitives had a multitude of functionality, for example morph was used to change volume, change mass and any other physical property, all at once. This made some of the primitives overly powerful. Due to all these problems, many programmers chose to circumvent G-Code by building additional high level machines directly in Java and one could therefore no longer be sure whether they complied with G-Code's principle of locality and what cells in Cx3D should be able to do. A new implementation had to be found that solves these problems.

A.2 The New Implementation

The new implementation of G-Code uses an abstract Java class Machine as its base machine. For a derived class the method to be implemented is the run method. This run method describes the machine and is its actual implementation. The abstract Machine class provides all the primitives as functions, overloading is used to handle primitives with multiple input and varying parameters. All the parameters that had been properties of the primitive before have been merged into the call of the primitive. Primitives that served multiple purposes such as morph have been taken apart into different functions. Functions either return a value when they are supposed to deliver information into the machine (reading primitives) or do not return values (writing primitives) but change the state of the cell or the environment. In order to distinguish between somas and growth cones, the Machine class has been derived further to produce an abstract class SomaMachine and an abstract class NeuriteMachine. SomaMachine implements all the primitives from a soma point of view whereas the NeuriteMachine implements the primitives for the growth cone. The modified primitives are is shown in table 5.1 where they are also sorted into the categories of the Developmental Programming paradigm.

Since G-Code is very Java like it can be edited in an integrated development environment like Eclipse, but this comes with the problem that all Java code would be valid in the run method of machines. To prevent this problem as much as possible before compilation, the primitives only accept data types that are specified separately as classes. G-Code specific data types allow for control over how data can be manipulated. Furthermore, to prevent any control of the machine from the outside, all the functions of primitives are private and therefore are only accessible from the run method. Since it would still be possible to write Java code that does not interact directly with the machines, a code checker was written that makes sure that only G-Code compliant code is written in the machines.

One of the problems with this way of coding the machines is that we lose the ability to directly reuse a Machine in another machine. Reuse is now only possible trough subroutines and inheritance, which means that certain functionality that is used by the run method is extracted and moved into a subroutine. This subroutine can be used from another class that extends the original one by inheritance. This is not ideal but bearable regarding all the advantages of writing G-Code in this way. The advantages include that it is faster to write, it has integrated development environment support, data type checking on compilation, very short and readable code, and debugging directly in the code. The artificial DNA is in theory now the collected code of all the machines that derive from the original SomaMachine and NeuriteMachine, but it is usually not as nicely collected as in the XML version.

Appendix B

Download and Execution Instructions

All the code that has been produced for this thesis can be downloaded from http://www.ini.uzh.ch/~haurian/workspace.zip. In order to run the code, workspace.zip has to be unpacked. Then eclipse can be used to open the unpacked directory as a workspace (eclipse can be downloaded from: http://www.eclipse.org/downloads/.)

The main simulation, the one that is described in chapter 3 and appendix C, can be executed by running the main program 'Launch' (package: ini.Cx3D.simulations.fred.ctx).

The programs in chapter 6 can be executed by running the main programs in the package Cx3DpPaperSim.

Appendix C

Machine Description

In this appendix we describe the G-Machines that are necessary in order to achieve an inter-laminar connection between the layers of the modelled cortex. This appendix is an extension of chapter 3.

Layer 5 Pyramidal Cells

The differentiation machine of layer 5 will start the basal dendrite outgrowth G-Machine(BasalDendrite), start an apical dendrite outgrowth G-Machine(P5ApicalDendriteMain) and an axonal outgrowth G-Machine(PyramidalAxonMainL5). Along with the PyramidalAxonMainL5 an additional G-Machine(P5SideOutgrowth) is launched in the same growth cone that controls the outgrowth of side branches in the appropriate regions. Figure C.1 shows the growth of layer 5 cells as a cartoon and figure C.2 in simulation. The development of a layer 5 pyramidal cell can be observed in the following video: http://youtu.be/gTSQbjLjwow 'Development of a layer 5 pyramidal neuron'.

P5ApicalDendriteMain:

- Grow the current dendrite in the direction of the marginal zone.
- Check if it has reached the marginal zone.
    - Yes: wait for a while. (Layers 2/3 and 4 might not yet be established.)
    - Check whether the marginal zone has in the meantime been pushed upward
        - Yes: Grow further up.
- Check if in layer 2/3 and check probabilistically whether to bifurcate

  - Yes: bifurcate and create two new growth cones containing instances of G-Machine GM(P5ApicalDendriteMain)

- Reduce the diameter
- Check if the diameter is too small
  - Yes: stop growing.

PyramidalAxonMainL5:

- Grow in the direction of the subplate
- Check if the growth cone is in reach of a thalamic fiber.
  - Yes: fasiculate to it and change the growth mode to fasciculation growth.
  - Grow along the thalamic fiber
  - Check if the growth has reached the thalamus.
    - Yes: stop growing.

P5SideOutgrowth:

- Check if in layer 4 and check probabilistically whether to sidebranch
  - Yes: Grow out two to three side branches with G-Machines(P5SideLong)

P5SideLong:

- Grow in the direction of the marginal zone.
- Check if there are no layer 5, 4 or 2/3 cells present
  - Yes: wait. (Layers 4 and 2/3 might not yet be established.)
- Check if in layer 4 and check probabilistically whether to bifurcate
  - Yes: bifurcate and create two new growth cones containing instances of G-Machine GM(P5SideLong)
- Reduce diameter
- If the diameter is too small
  - Yes: stop growing.

This PyramidalAxonMainL5 deviates from the biology in that not only thalamus is targeted by layer 5 pyramidal cells but also the brain stem. These machines produce patches in layer 2/3 and a growth through layer 4 of many sidebranches where the sidebranches do not bifurcate in layer 4 but only in layer 2/3.

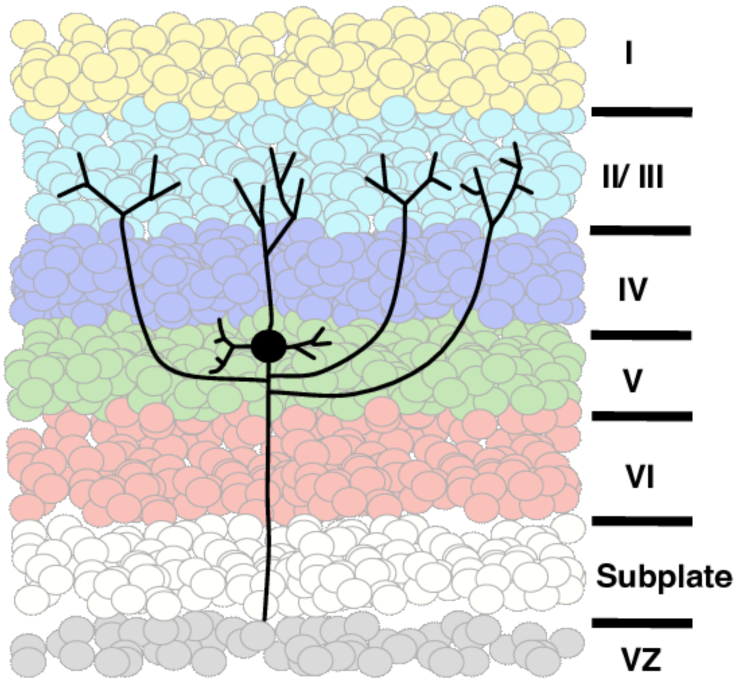


Figure C.1. Cartoon of a mature layer 5 pyramidal neuron with local basal dendrites, an apical dendrite in layer 2/3, and an axon that grows down to the subplate and projects side branches to layer 2/3.

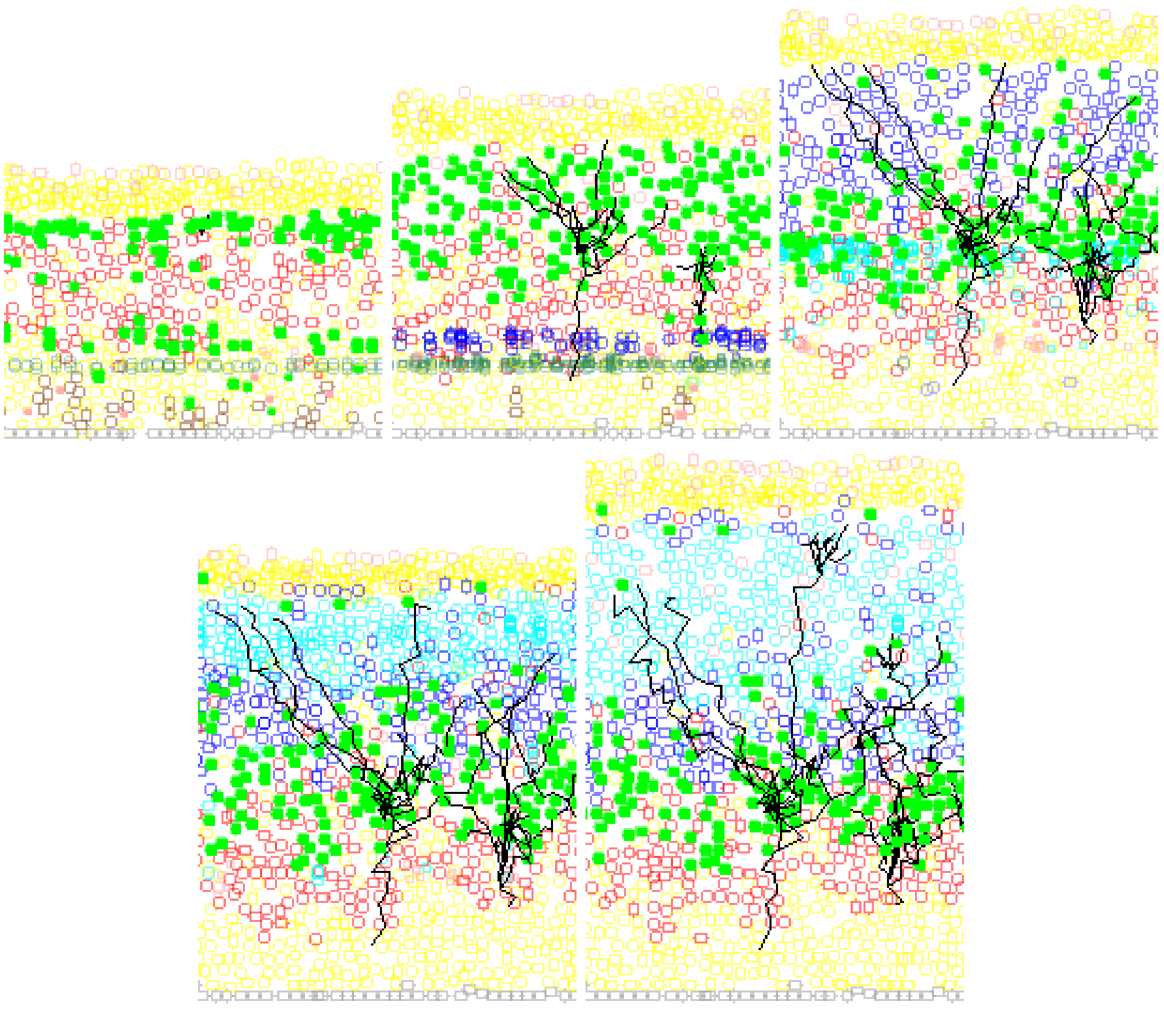


Figure C.2. From top left to bottom right: The development of a layer 5 pyramidal cell, growing out an initial axon in the direction of the subplate, sending out side branches in the direction of layer 2/3. The axons bound for layer 2/3 wait at the marginal zone border for the future layers to appear. The basal dendrites are sent out radially and the apical dendrite grows in the direction of layer 2/3.

Layer 4 Pyramidal Cells

The differentiation machine of the layer 4 pyramidal cells starts the basal dendrite outgrowth G-Machine(BasalDendrite) and an axon generating G-Machine(PyramidalAxonMain). Along with the machine PyramidalAxonMain a second G-Machine(P4Side1Outgrowth) is placed into the growth cone that controls the outgrowth of side branches.

PyramidalAxonMain:

- Grow in the direction of the subplate
- Check if the growth has reached the subplate.
  - Yes: stop growing.

P4Side1Outgrowth:

- Check if in layer 4 and check probabilistically whether to sidebranch
  - Yes: Grow out two to three side branches with G-Machine(P4Side1) and G-Machine (P4Side2Outgrowth)

P4Side2Outgrowth:

- Check probabilistically whether to sidebranch
  - Yes: Grow out two to three side branches with G-Machines(P4Side2)

P4Side1:

- Grow in the direction of the marginal zone.
- Reduce the diameter
  - Yes: stop growing.

P4Side2:

- Grow in the direction of the marginal zone.
- Check if the marginal zone is reached
  - Yes: wait. (Layer 2/3 might not yet be established and the marginal zone is not growth permissive for this axon.)

- Reduce the diameter
- Check if the diameter is too small
  - Yes: stop growing.
- Check if layer 2/3 cells are detected and check probabilistically whether to bifurcate
  - Yes: bifurcate and create two new growth cones containing instances of G-Machine(P4Side2)

In figure C.3 we show a cartoon of a layer 4 cell and figure C.4 shows a simulation of a layer 4 pyramidal cell that is growing. The development of a layer 4 pyramidal cell can be observed in the following video: http://youtu.be/qIh1jlqzZgM 'Development of a Layer 4 Pyramidal Neuron'.

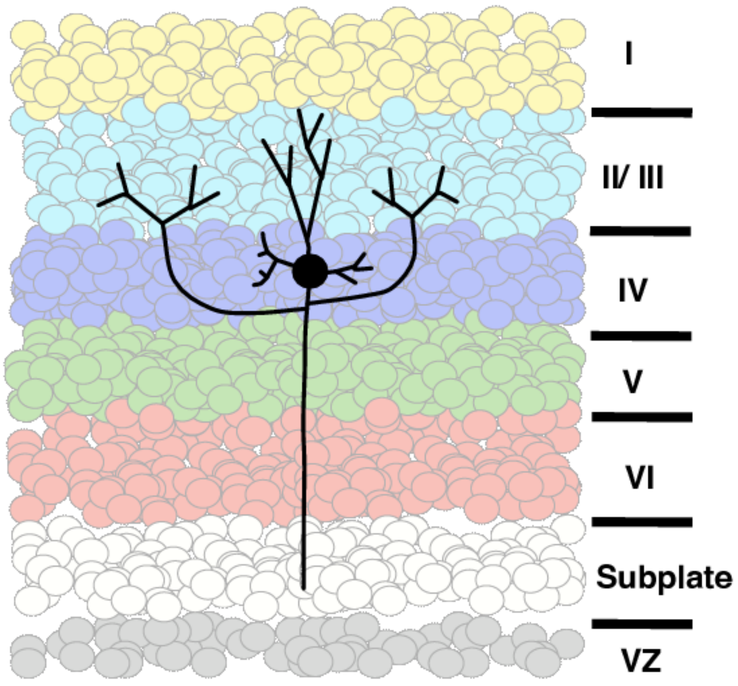


Figure C.3. Cartoon showing a mature layer 4 pyramidal cell projecting to layer 2/3 with axonal side branches, and with the main axon growing down to the subplate. The apical dendrite projects to layer 2/3 and a basal dendrite arborizes locally.

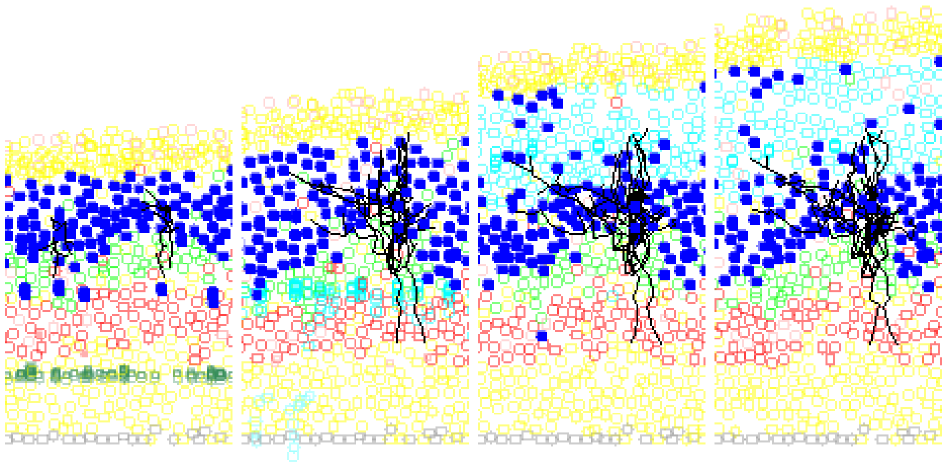


Figure C.4. From left to right: The outgrowth of a layer 4 pyramidal cell with local basal dendrites growing out. The cell is sending out an axon in the direction of the subplate that branches in layer 4 and projects to layer 2/3.

Layer 2/3 pyramidal cells

The layer 2/3 pyramidal neuron outgrowth pattern is started by the differentiation G-Machine of this type. This G-Machine(BasalDendrite) grows out the basal dendrite as described previously, an apical dendrite G-Machine(P23ApicalDendrite) is created and an axon is grown in the direction of the white matter by the G-Machine(PyramidalAxonMain).

P23ApicalDendrite:

- Grow the current dendrite in the direction of the marginal zone.
- Reduce the diameter
- Check if the diameter is too small
  - Yes: stop growing.
- Check probabilistically whether to bifurcate
  - Yes: bifurcate and create two new growth cones containing instances of G-Machine GM(P23Side23)
- Check if marginal zone cells or Layer 2/3 cells are present
  - Yes: stop growing.

The down growing axon is driven by the PyramidalAxonMain G-Machine (the same as in the Layer 4 Pyramidal cells) using additionally the P23Side23Outgrowth and P23Side5Outgrowth machines in the same growth cone to produce side branches in layers 2/3 and 5.

P23Side23Outgrowth:

- Check if in layer 2/3 and check probabilistically whether to grow out
  - Yes: grow out two to three side branches containing G-Machine(P23Side23) in the growth cone

P23Side5Outgrowth:

- Check if in layer 5 and check probabilistically whether to grow out
  - Yes: grow out two to three side branches containing G-Machine(P23Side5) in the growth cone

P23Side23:

- Grow in the outgrowth direction.
- Check if layer 2/3 has been left
  - Yes: stop growing.
- Reduce the diameter.
- Check if the diameter is small enough and check probabilistically whether to bifurcate
  - Yes: bifurcate and create two new growth cones containing instances of G-Machine GM(P23Side23)
- Check if the diameter is too small.
  - Yes: stop growing.

P23Side5:

- Grow in the outgrowth direction.
- Check if layer 5 has been left
  - Yes: stop growing.
- Reduce the diameter.
- Check if the diameter is small enough and check probabilistically whether to bifurcate
  - Yes: bifurcate and create two new growth cones containing instances of G-Machine GM(P23Side5)
- Check if the diameter is too small.
  - Yes: stop growing.

The recursive calls to P23Side23 and P23Side5 produce patches at a certain distance from the main trunk due to the waiting time until the diameter of the axon is small enough. P23Side5 works like P23Side23 except that it reduces the diameter faster and that it needs layer 5 cells around to grow. This results in layer 5 innervation that has shorter side branches than layer 2/3 and therefore less innervation. The development of a layer 2/3 pyramidal cell can be observed in the following video: http://youtu.be/unD0BqpcDq0 'Development of a Layer 2/3 Pyramidal Neuron'

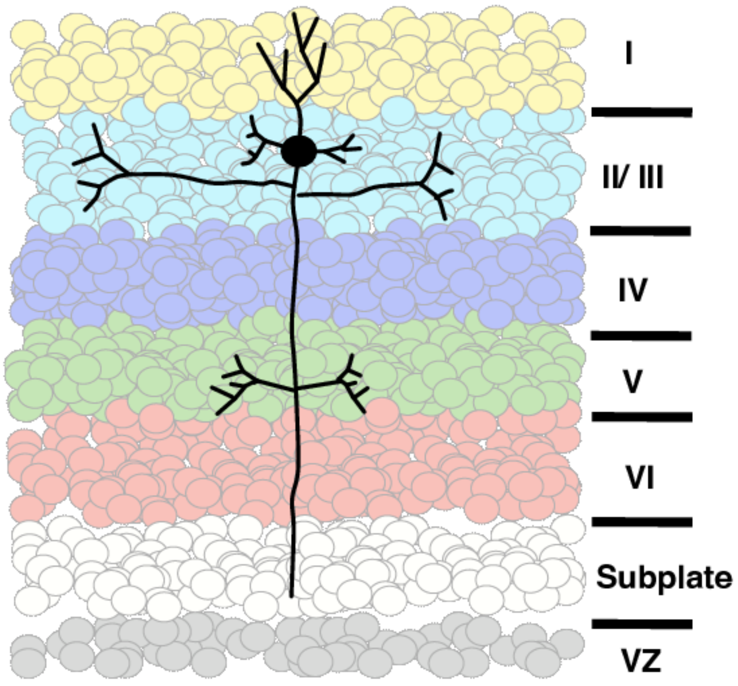


Figure C.5. A cartoon showing a fully grown layer 2/3 pyramidal cell that has projected an apical dendrite to layer 1. Its axon has grown down to the subplate and it has projected to layer 2/3 and layer 5 with each projection having a patch at the end.

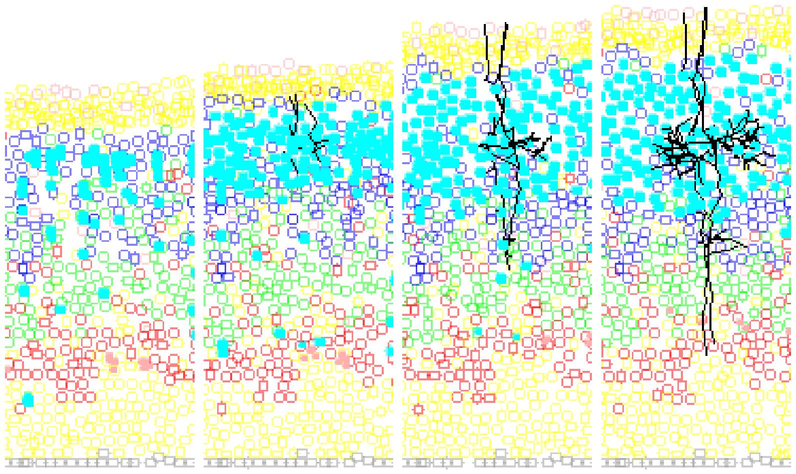


Figure C.6. From left to right: The outgrowth of a layer 2/3 pyramidal cell, its axon growing down to the subplate, projecting side branches in layer 2/3 and in layer 5 creating patches away from the main trunk. It sends out an apical dendrite that projects to the marginal zone or future layer 1.

Layer 2/3 Basketcell (Interneuron)

The layer 2/3 interneurons will settle into the cortex after their migration tangentially into the cortex. This tangential migration G-Machine starts the differentiation G-Machine of the interneurons in layer 2/3 after nesting. The differentiation G-Machine will start the basal dendrite outgrowth G-Machine (BasalDendrite) and an axonal growth G-Machine (Bas23StemDown). The same growth cone also contains a G-Machine(Bas23HorizontalOutgrowth) that will create side branches.

Bas23StemDown:

- Grow out in the direction of the marginal zone.
- Grow in the direction of the subplate
- Check if the growth has reached the subplate cells
  - Yes: stop growing.
- Check if layer 6 cells are detected
  - Yes: stop the G-Machine in the growth cone and replace it with the G-Machine GM(Bas23BasketPatch6).

Bas23HorizontalOutgrowth:

- Grow out four to five side branches where the growth cones contain the G-Machine GM(Bas23BasketPatch23).
- Grow out three side branches where the growth cones contain the G-Machine(Bas23Horizontal).

Bas23BasketPatch23:

- Grow in the outgrowth direction.
- Check if layer 2/3 has been left
  - Yes: stop growing
- Check probabilistically whether to stop growing:
  - Yes: stop growing
- Check probabilistically whether to bifurcate

  - Yes: bifurcate and start two new growth cones with G-Machine(Bas23BasketPatch23) containing an increased probability to stop growing.

Bas23Horizontal:

- Grow in the outgrowth direction.
- Check if layer 2/3 has been left
  - Yes: stop growing
- reduce the diameter.
- Check if the diameter is too small.
  - Yes: stop the G-Machine and replace it with G-Machine(Bas23BasketPatch23)

Bas23BasketPatch6:

- Grow in the outgrowth direction.
- Check if layer 6 has been left
  - Yes: stop growing
- Check probabilistically whether to stop growing
  - Yes: stop growing
- Check probabilistically whether to bifurcate
  - Yes: bifurcate and start two new growth cones with G-Machine(Bas23BasketPatch6) containing an increased probability to stop growing.

The discrepancy between outgrowth direction and growth direction allows the cells to create a looping structure at the beginning of the axon. The two axonal patch generating G-Machines Bas23BasketPatch6 and Bas23BasketPatch23 are essentially the same, just intended for other layers to create a patchy structure through recursion.

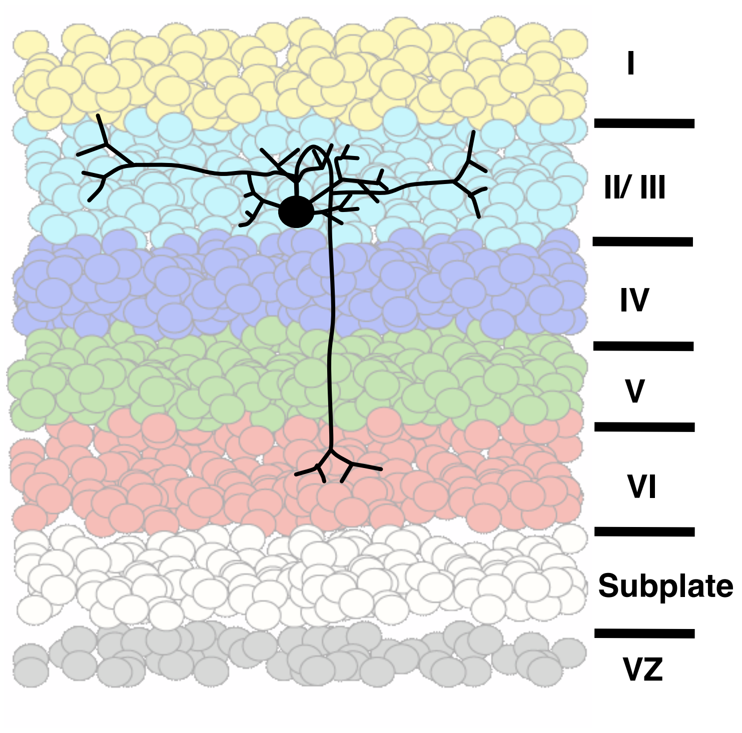


Figure C.7. An interneuron of layer 2/3 with a local basal dendrite and an outgrowing axon that projects to layer 4.

Bibliography

1. J. Aćimović, T. Mäki-Marttunen, R. Havela, H. Teppola, and M. L. Linne. Modeling of neuronal growth in vitro: Comparison of simulation tools NETMORPH and CX3D. *EURASIP Journal on Bioinformatics and Systems Biology*, 2011, 2011.

2. K. L. Allendoerfer and C. J. Shatz. The subplate, a transient neocortical structure: its role in the development of connections between thalamus and cortex. *Annual review of neuroscience*, 17(1):185–218, 1994.

3. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

4. Pierre America, Jaco de Bakker, Joost N. Kok, and Jan J. M. M. Rutten. Operational semantics of a parallel object-oriented language. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '86, pages 194–208, New York, NY, USA, 1986. ACM.

5. Rajagopal Ananthanarayanan, Steven K. Esser, Horst D. Simon, and Dharmendra S. Modha. The cat is out of the bag: cortical simulations with 109 neurons, 1013 synapses. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 63:1–63:12, Portland, Oregon, 2009. ACM. ACM ID: 1654124.

6. Mohammad H. Ansari and Lee Smolin. Self-organized criticality in quantum gravity. *arXiv:hep-th/0412307*, December 2004. Class.Quant.Grav.25:095016,2008.

7. Todd E. Anthony, Corinna Klein, Gord Fishell, and Nathaniel Heintz. Radial glia serve as neuronal progenitors in all regions of the central nervous system. *Neuron*, 41(6):881–90, March 2004.

8. D. Arbuckle and A. Requicha. Self-assembly and self-repair of arbitrary shapes by a swarm of reactive robots: algorithms and simulations. *Autonomous Robots*, 28(2):197–211, 2010.

9. Spyros Artavanis-Tsakonas, Matthew D. Rand, and Robert J. Lake. Notch signaling: Cell fate control and signal integration in development. *Science*, 284(5415):770–776, April 1999.

10. Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

11. G. A Ascoli and J. L Krichmar. L-neuron: a modeling tool for the efficient generation and parsimonious description of dendritic morphology. *Neurocomputing*, 32(1):1003–1012, 2000.

12. Hilary L. Ashe and James Briscoe. The interpretation of morphogen gradients. *Development*, 133(3):385–394, January 2006.

13. C. Auladell, P. Prez-Sust, H. Supr, and E. Soriano. The early development of thalamocortical and corticothalamic projections in the mouse. *Brain Structure and Function*, 201(3):169–179, 2000.

14. J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, et al. Report on the algorithmic language ALGOL 60. *Numerische Mathematik*, 2(1):106–136, 1960.

15. John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

16. E. H. Baehrecke et al. How death shapes life during development. *Nature Reviews Molecular Cell Biology*, 3(10):779–787, 2002.

17. M. Baldi. Simulations of structure formation in interacting dark energy cosmologies. *Nuclear Physics B - Proceedings Supplements*, 194:178–184, October 2009.

18. Renata Batista-Brito and Gord Fishell. Chapter 3: The Developmental Integration of Cortical Interneurons into a Functional Network. In *Current Topics in Developmental Biology*, volume 87, pages 81–118. Elsevier, 2009.

19. Roman Bauer. A model for the self-organization of winner-take-all structure and function. 2013. (To be published).

20. Roman Bauer, Frederic Zubler, Andreas Hauri, Dylan R. Muir, and Rodney J. Douglas. Developmental origin of patchy axonal connectivity in the neocortex: A computational model. *Cerebral Cortex*, November 2012.

21. G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *Computer*, 39(1):110–112, January 2006.

22. G. Beni and J. Wang. Swarm intelligence in cellular robotic systems. *Robots and Biological Systems: Towards a New Bionics?*, pages 703–712, 1993.

23. Tom Binzegger, Rodney J. Douglas, and Kevan A. C. Martin. A quantitative map of the circuit of cat primary visual cortex. *The Journal of Neuroscience*, 24(39):8441–8453, September 2004.

24. Tom Binzegger, Rodney J. Douglas, and Kevan A. C. Martin. Axons in cat visual cortex are topologically self-similar. *Cerebral Cortex*, 15(2):152–165, January 2005.

25. Bletchley's code-cracking Colossus. http://news.bbc.co.uk/2/hi/technology/8492762.stm, February 2010.

26. C. Boehm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

27. N. Bohr. On the constitution of atoms and molecules. *Philosophical Magazine*, 26(151):1–25, 1913.

28. James M. Bower and David Beeman. *The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System.* TELOS, 1995.

29. Maureen P. Boyle, Amy Bernard, Carol L. Thompson, Lydia Ng, Andrew Boe, Marty Mortrud, Michael J. Hawrylycz, Allan R. Jones, Robert F. Hevner, and Ed S. Lein. Cell-type-specific consequences of reelin deficiency in the mouse neocortex, hippocampus, and amygdala. *The Journal of Comparative Neurology*, 519(11):2061–2089, August 2011.

30. Olga Britanova, Amanda F. P. Cheung, Camino de Juan Romero, Kenneth Y. Kwan, Manuela Schwark, Andrea Gyorgy, Tanja Vogel, Sergey Akopov, Miso Mitkovski, Denes Agoston, Nenad Sestan, Zoltán Molnár, and Victor Tarabykin. Satb2 is a postmitotic determinant for upper-layer neuron specification in the neocortex. *Neuron*, 57(3):378–92, 2008.

31. Arthur Walter Burks. *Essays on cellular automata.* University of Illinois Press, 1970.

32. Anna Q. Cai, Kerry A. Landman, and Barry D. Hughes. Modelling directional guidance and motility regulation in cell migration. *Bulletin of Mathematical Biology*, 68(1):25–52, January 2006.

33. Iván J. Cajigas, Georgi Tushev, Tristan J. Will, Susanne tom Dieck, Nicole Fuerst, and Erin M. Schuman. The local transcriptome in the synaptic neuropil revealed by deep sequencing and high-resolution imaging. *Neuron*, 74(3):453–466, May 2012.

34. Scott Camazine. *Self-Organization in Biological Systems.* Princeton University Press, 2003.

35. Denis Caromel. HAL :: [hal-00125034, version 1] ProActive: an integrated platform for programming and running applications on grids and P2P systems. http://hal.archives-ouvertes.fr/hal-00125034/, 2006.

36. A. Cavagna, S. M. Duarte Queirós, I. Giardina, F. Stefanini, and M. Viale. Diffusion of individual birds in starling flocks. *Proceedings. Biological sciences / The Royal Society*, 280(1756):20122484, April 2013. PMID: 23407827.

37. Bin Chen, Laura R. Schaevitz, and Susan K. McConnell. Fezl regulates the differentiation and axon targeting of layer 5 subcortical projection neurons in cerebral cortex. *Proceedings of the National Academy of Sciences of the United States of America*, 102(47):17184–9, November 2005.

38. Xiaoguang Chen and G. Wayne Brodland. Multi-scale finite element modeling allows the mechanics of amphibian neurulation to be elucidated. *Physical Biology*, 5(1):015003, March 2008.

39. Joshua L. Cherry and Frederick R. Adler. How to make a biological switch. *Journal of Theoretical Biology*, 203(2):117–133, March 2000.

40. Scott Christley, Yiming Lu, Chen Li, and Xiaohui Xie. Human genomes as email attachments. *Bioinformatics*, 25(2):274–275, January 2009.

41. A. Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 33(2):346–366, 1932.

42. F. Clasca, A. Angelucci, and M. Sur. Layer-specific programs of development in neocortical projection neurons. *Proceedings of the National Academy of Sciences*, 92(24):11145, 1995.

43. J. R. Collier, N. A. M. Monk, P. K. Maini, and J. H. Lewis. Pattern formation by lateral inhibition with feedback: a mathematical model of delta-notch intercellular signalling. http://eprints.maths.ox.ac.uk/450/, 1996.

44. International Human Genome Sequencing Consortium. Finishing the euchromatic sequence of the human genome. *Nature*, 431(7011):931–945, October 2004.

45. Marcos R. Costa, Oliver Bucholz, Timm T. Schroeder, and Magdalena Götz. Late origin of glia-restricted progenitors in the developing mouse cerebral cortex. *Cerebral cortex (New York, N.Y. : 1991)*, 19 Suppl 1(July):i135–43, July 2009.

46. Peter B. Crino and James Eberwine. Molecular characterization of the dendritic growth cone: Regulated mRNA transport and local protein synthesis. *Neuron*, 17(6):1173–1187, December 1996.

47. P Cuatrecasas. Membrane receptors. *Annual Review of Biochemistry*, 43(1):169–214, 1974.

48. Beatriz Cubelos, Alvaro Sebastián-Serrano, Seonhee Kim, Juan Miguel Redondo, C. Moreno-Ortiz, Christopher A. Walsh, and M. Nieto. Cux-2 controls the proliferation of neuronal intermediate precursors of the cortical subventricular zone. *Cerebral Cortex*, 18(8):1758, 2008.

49. Hermann Cuntz, Alexander Borst, and Idan Segev. Optimization principles of dendritic structure. *Theoretical Biology & Medical Modelling*, 4:21, 2007. PMID: 17559645.

50. L. Davis, P. Dou, M. DeWit, and S. B. Kater. Protein synthesis within neuronal growth cones. *The Journal of Neuroscience*, 12(12):4867–4877, January 1992.

51. Colette Dehay and Henry Kennedy. Cell-cycle control and cortical development. *Nature Reviews Neuroscience*, 8(6):438–450, June 2007.

52. Colette Dehay, F. Polleux, and Henry Kennedy. The timetable of laminar neurogenesis contributes to the specification of cortical areas in mouse isocortex. *The Journal of Comparative Neurology*, 385(1):95–116, 1997.

53. Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

54. Christoffel Dinant, Martijn S. Luijsterburg, Thomas Hfer, Gesa Von Bornstaedt, Wim Vermeulen, Adriaan B. Houtsmuller, and Roel Van Driel. Assembly of multiprotein

complexes that control genome function. *The Journal of Cell Biology*, 185(1):21–26, June 2009.

55. Jack Dongarra, Rolf Hempel, Anthony J. G. Hey, and David W. Walker. A proposal for a user-level, message-passing interface in a distributed memory environment, 1993.

56. Duncan E. Donohue and Giorgio A. Ascoli. A comparative computer simulation of dendritic morphology. *PLoS Comput Biol*, 4(6):e1000089, June 2008.

57. K. E. Drexler and M. Minsky. *Engines of creation.* Anchor Press/Doubleday, 1990.

58. J. P. Eberhard, A. Wanner, and G. Wittum. NeuGen: a tool for the generation of realistic morphology of cortical neurons and neural networks in 3D. *Neurocomputing*, 70(1-3):327–342, December 2006.

59. Chris Englund, A. Fink, C. Lau, D. Pham, Ray. A. M. Daza, Alessandro Bulfone, Tom Kowalczyk, and Robert F. Hevner. Pax6, Tbr2, and Tbr1 are expressed sequentially by radial glia, intermediate progenitor cells, and postmitotic neurons in developing neocortex. *The Journal of Neuroscience*, 25(1):247, 2005.

60. Helen Everett and Grant McFadden. Apoptosis: an innate immune response to virus infection. *Trends in Microbiology*, 7(4):160–165, April 1999.

61. Manuel Faehndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 302–312, New York, NY, USA, 2003. ACM.

62. R. P. Feynman. There's plenty of room at the bottom. *Engineering and Science*, 23(5):22–36, 1960.

63. D. B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, January 1994.

64. J. Fox and R. Andersen. Using the R statistical computing environment to teach social statistics courses. *Department of Sociology, McMaster University*, 2005.

65. K. Fox. A critical period for experience-dependent synaptic plasticity in rat barrel cortex. *The Journal of Neuroscience*, 12(5):1826–1838, January 1992.

66. G. D. Frantz, J. M. Weimann, M. E. Levin, and S. K. McConnell. Otx1 and Otx2 define layers and regions in developing cerebral cortex and cerebellum. *The Journal*

of neuroscience : the official journal of the Society for Neuroscience, 14(10):5725–40, October 1994.

67. D. Galorath. Software reuse and commercial off-the-shelf software, 2007.

68. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *ECOOP 93 Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer Berlin / Heidelberg, 1993.

69. Sonia Garel and John L. R. Rubenstein. Intermediate targets in formation of topographic projections: inputs from the thalamocortical system. *Trends in Neurosciences*, 27(9):533–539, September 2004.

70. Clare E. Giacomantonio and Geoffrey J. Goodhill. A boolean model of the gene regulatory network underlying mammalian cortical area development. *PLoS Comput Biol*, 6(9):e1000936, September 2010.

71. Daniel G. Gibson, John I. Glass, Carole Lartigue, Vladimir N. Noskov, Ray-Yuan Chuang, Mikkel A. Algire, Gwynedd A. Benders, Michael G. Montague, Li Ma, Monzia M. Moodie, Chuck Merryman, Sanjay Vashee, Radha Krishnakumar, Nacyra Assad-Garcia, Cynthia Andrews-Pfannkoch, Evgeniya A. Denisova, Lei Young, Zhi-Qing Qi, Thomas H. Segall-Shapiro, Christopher H. Calvey, Prashanth P. Parmar, Clyde A. Hutchison, Hamilton O. Smith, and J. Craig Venter. Creation of a bacterial cell controlled by a chemically synthesized genome. *Science*, 329(5987):52–56, February 2010.

72. K. Gilpin, A. Knaian, and D. Rus. Robot pebbles: One centimeter modules for programmable matter through self-disassembly. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2485–2492, May 2010.

73. A. Goldberg, A. Kay, and Xerox Corporation. *Smalltalk-72: Instruction Manual.* Xerox Corporation, 1976.

74. P. R. Gordon-Weeks. Microtubules and growth cone function. *Journal of neurobiology*, 58(1):70–83, 2004.

75. A. S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *Computer*, 26(5):39–51, May 1993.

76. Barry M. Gumbiner. Cell adhesion: The molecular basis of tissue architecture and morphogenesis. *Cell*, 84(3):345–357, February 1996.

77. Eva Hartfuss, R Galli, Nico Heins, and Magdalena Götz. Characterization of CNS precursor subtypes and radial glia. *Developmental biology*, 229(1):15–30, January 2001.

78. Andreas Hauri and Rodney J. Douglas. Self-construction in the context of cortical growth: From one cell to a cortex. (To be published).

79. T. F. Haydar, C. Y. Kuan, R. A. Flavell, and P. Rakic. The role of cell death in regulating the size and shape of the mammalian forebrain. *Cerebral Cortex*, 9(6):621, 1999.

80. Jia-Qiang He, Yue Ma, Youngsook Lee, James A. Thomson, and Timothy J. Kamp. Human embryonic stem cells develop into multiple types of cardiac myocytes: Action potential characterization. *Circulation Research*, 93(1):32–39, November 2003.

81. Nico Heins, Paolo Malatesta, Francesco Cecconi, Masato Nakafuku, Kerry Lee Tucker, Michael a Hack, Prisca Chapouton, Yves-Alain Barde, and Magdalena Götz. Glial cells generate neurons: the role of the transcription factor Pax6. *Nature neuroscience*, 5(4):308–15, April 2002.

82. Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 365–377, New York, NY, USA, 1998. ACM.

83. H. G. E Hentschel and A. Van Ooyen. Models of axon guidance and bundling during development. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 266(1434):2231–2238, July 1999.

84. H. H. Hesselink, H. Kuiper, and J. M. van den Akker. *Application of genetic algorithms in the aerospace domain.* Citeseer, 1997.

85. M. L. Hines and N. T. Carnevale. The NEURON simulation environment. *Neural Computation*, 9(6):1179–1209, 1997.

86. Yusuke Hirabayashi and Yukiko Gotoh. Epigenetic control of neural precursor cell fate during development. *Nature Reviews Neuroscience*, 11(6):377–388, June 2010.

87. Sui Huang, G. Eichler, Y. Bar-Yam, and D. E. Ingber. Cell fates as high-dimensional attractor states of a complex gene regulatory network. *Physical review letters*, 94(12):128701, 2005.

88. Sui Huang, Y. P. Guo, Tariq Enver, and G. May. Bifurcation dynamics in lineage-commitment in bipotent progenitor cells. *Developmental Biology*, 305(2):695–713, 2007.

89. P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

90. Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1 – 12–55, New York, NY, USA, 2007. ACM.

91. IBM archives: 1928. http://www-03.ibm.com/ibm/history/history/year_1928.html.

92. T. Ishida, Y. Sasaki, and Y. Fukuhara. Use of procedural programming languages for controlling production systems. In *Artificial Intelligence Applications, 1991. Proceedings., Seventh IEEE Conference on*, volume i, pages 71–75, February 1991.

93. J. A. Izaguirre, R. Chaturvedi, C. Huang, T. Cickovski, J. Coffland, G. Thomas, G. Forgacs, M. Alber, G. Hentschel, S. A. Newman, and J. A. Glazier. CompuCell, a multi-model framework for simulation of morphogenesis. *Bioinformatics*, 20(7):1129–1137, January 2004.

94. Peter Jagers. Branching processes. In *Encyclopedia of Biostatistics*. John Wiley & Sons, Ltd, 2005.

95. Matthias Jeschke, Roland Ewald, Alfred Park, Richard Fujimoto, and Adelinde M. Uhrmacher. A parallel and distributed discrete event approach for spatial cell-biological simulations. *SIGMETRICS Perform. Eval. Rev.*, 35(4):22–31, March 2008.

96. Patrick O. Kanold and Heiko J. Luhmann. The subplate and early cortical circuits. *Annual Review of Neuroscience*, 33(1):23–48, June 2010.

97. Stuart A. Kauffman. The origins of order: self-organization and selection in evolution, 1993.

98. H. Kennedy and C. Dehay. The importance of developmental timing in cortical specification. *Perspectives on Developmental Neurobiology*, 1(2):93–99, 1993. PMID: 8087537.

99. Henry Kennedy and Colette Dehay. Cortical specification of mice and men. *Cerebral Cortex*, 3(3):171 –186, May 1993.

100. Mary B. Kennedy, Holly C. Beale, Holly J. Carlisle, and Lorraine R. Washburn. Integration of biochemical signalling in spines. *Nature Reviews Neuroscience*, 6(6):423–434, June 2005.

101. Michel Kerszberg and Lewis Wolpert. Mechanisms for positional signalling by morphogen transport: a theoretical study. *Journal of Theoretical Biology*, 191(1):103–114, March 1998.

102. M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber. SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 2849–2856, June 2008.

103. H.P. Killackey, R.W. Rhoades, and C.A. Bennett-Clarke. The formation of a cortical somatotopic map. *Trends in neurosciences*, 18(9):402–407, 1995.

104. Randal A. Koene, Betty Tijms, Peter Hees, Frank Postma, Alexander Ridder, Ger J. A. Ramakers, Jaap Pelt, and Arjen Ooyen. NETMORPH: a framework for the stochastic generation of large scale neuronal networks with realistic neuron morphologies. *Neuroinformatics*, 7(3):195–210, August 2009.

105. A. N. Kolmogorov. Three approaches to the quantitative definition of information. *International Journal of Computer Mathematics*, 2(1-4):157–168, 1968.

106. Arnold Kriegstein and A. Alvarez-Buylla. The glial nature of embryonic and adult neural stem cells. *Annual review of neuroscience*, 32:149, 2009.

107. Eva Kutejova, James Briscoe, and Anna Kicheva. Temporal dynamics of patterning by morphogen gradients. *Current Opinion in Genetics & Development*, 19(4):315–322, August 2009.

108. B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. *SIGCHI Bull.*, 20(SI):69–73, March 1989.

109. R. Greg Lavender and Douglas C. Schmidt. Active object — an object behavioral pattern for concurrent programming, 1995.

110. Michael Levine and Eric H. Davidson. Gene regulatory networks for development. *Proceedings of the National Academy of Sciences of the United States of America*, 102(14):4936–42, April 2005.

111. Aristid Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, March 1968.

112. G. López-Bendito and Z. Molnár. Thalamocortical development: how are we going to get there? *Nature Reviews Neuroscience*, 4:276–289, 2003.

113. Agnès Lukaszewicz, Véronique Cortay, P. Savatier, Pascale Giroud, Michel Berland, C. Huissoud, Henry Kennedy, and Colette Dehay. G1 phase regulation, area-specific cell cycle control, and cytoarchitectonics in the primate cortex. *Neuron*, 47(3):353–364, 2005.

114. Paolo Malatesta, Michael A. Hack, Eva Hartfuss, Helmut Kettenmann, Wolfgang Klinkert, Frank Kirchhoff, and Magdalena Götz. Neuronal or glial progeny: regional differences in radial glia fate. *Neuron*, 37(5):751–64, March 2003.

115. Benoit B. Mandelbrot. Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

116. Luciano Marcon, Carlos G. Arqus, Miguel S. Torres, and James Sharpe. A computational clonal analysis of the developing mouse limb bud. *PLoS Comput Biol*, 7(2):e1001071, February 2011.

117. V. Marigo and C. J. Tabin. Regulation of *Patched* by Sonic hedgehog in the developing neural tube. *Proceedings of the National Academy of Sciences*, 93(18):9346–9351, March 1996.

118. Oscar Marin and John L. R. Rubenstein. Cell migration in the forebrain. *Annual Review of Neuroscience*, 26(1):441–483, March 2003.

119. K. C. Martin, M. Barad, and E. R. Kandel. Local protein synthesis and its role in synapse-specific plasticity. *Current opinion in neurobiology*, 10(5):587–592, 2000.

120. Athanasius F. M. Mare and Paulien Hogeweg. How amoeboids self-organize into a fruiting body: Multicellular coordination in *Dictyostelium discoideum*. *Proceedings of the National Academy of Sciences*, 98(7):3879–3883, March 2001.

121. Pierre Mattar, Lisa Marie Langevin, Kathryn Markham, Natalia Klenin, Salma Shivji, Dawn Zinyk, and Carol Schuurmans. Basic helix-loop-helix transcription factors cooperate to specify a cortical projection neuron identity. *Molecular and cellular biology*, 28(5):1456–69, March 2008.

122. D. J. R. Meagher. Octree encoding: a new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. Technical Report IPL-TR-80-111, Rensselaer Polytechnic Institute Image Processing Laboratory, October 1980.

123. Roeland M. H. Merks, Erica D. Perryn, Abbas Shirinifard, and James A. Glazier. Contact-inhibited chemotaxis in de novo and sprouting blood-vessel growth. *PLoS Comput Biol*, 4(9):e1000163, September 2008.

124. G. Meyer, J. P. Schaaps, L. Moreau, and A. M. Goffinet. Embryonic and early fetal development of the human neocortex. *The Journal of Neuroscience*, 20(5):1858–1868, 2000.

125. G. Michal, editor. *Biochemical Pathways: An Atlas of Biochemistry and Molecular Biology*. J. Wiley; Heidelberg: Spektrum, 1999.

126. M. Migliore, C. Cannia, W. W. Lytton, Henry Markram, and M. L. Hines. Parallel network simulations with NEURON. *Journal of Computational Neuroscience*, 21(2):119–129, May 2006.

127. B. Miller, L. Chou, and B. L. Finlay. The early development of thalamocortical and corticothalamic projections. *The Journal of comparative neurology*, 335(1):16–41, 1993.

128. Takaki Miyata, Ayano Kawaguchi, H Okano, and M. Ogawa. Asymmetric inheritance of radial glial fibers by cortical neurons. *Neuron*, 31(5):727–41, September 2001.

129. Goichi Miyoshi and Gord Fishell. GABAergic interneuron lineages selectively sort into specific cortical layers during early postnatal development. *Cerebral Cortex*, 21(4):845–852, January 2011.

130. Z. Molnar and C. Blakemore. How do thalamic axons find their way to the cortex? *Trends in neurosciences*, 18(9):389–397, 1995.

131. B. J. Molyneaux, P. Arlotta, J. R. L. Menezes, and J. D. Macklis. Neuronal subtype specification in the cerebral cortex. *Nature Reviews Neuroscience*, 8(6):427–437, 2007.

132. G. E. Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

133. Bernhard K. Mueller. Growth cone guidance: First steps towards a deeper understanding. *Annual Review of Neuroscience*, 22(1):351–388, 1999.

134. B. Nadarajah, J. G. Parnavelas, et al. Modes of neuronal migration in the developing cerebral cortex. *Nature Reviews Neuroscience*, 3(6):423–432, 2002.

135. Radhika Nagpal. Programmable self-assembly using biologically-inspired multiagent control. In *Proceedings of the first international joint conference on Autonomous agents and*

multiagent systems: part 1, AAMAS '02, pages 418–425, New York, NY, USA, 2002. ACM.

136. Mark T. Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant V. Kal, Robert D. Skeel, and Klaus Schulten. NAMD: a parallel, object-oriented molecular dynamics program. *International Journal of High Performance Computing Applications*, 10(4):251–268, December 1996.

137. I. Newton. *Philosophiæ Naturalis Principia Mathematica (mathematical principles of natural philosophy).* Royal Society, London, 1687.

138. M. Nieto, E. S. Monuki, H. Tang, Nicole Haubst, J. Imitola, S. J. Khoury, Magdalena Götz, J. Cunningham, and Christopher A. Walsh. Expression of Cux-1 and Cux-2 in the subventricular zone and upper layers II–IV of the cerebral cortex. *The Journal of comparative neurology*, 479(2):168–180, 2004.

139. Hitoshi Niwa, Yayoi Toyooka, Daisuke Shimosato, Dan Strumpf, Kadue Takahashi, Rika Yagi, and Janet Rossant. Interaction between Oct3/4 and Cdx2 determines trophectoderm differentiation. *Cell*, 123(5):917–29, December 2005.

140. Stephen C. Noctor, Alexander C. Flint, Tamily A. Weissman, R. S. Dammerman, and Arnold R. Kriegstein. Neurons derived from radial glial cells establish radial units in neocortex. *Nature*, 409(6821):714–720, March 2001.

141. Stephen C. Noctor, Alexander C. Flint, Tamily A. Weissman, Winston S. Wong, Brian K. Clinton, and Arnold R. Kriegstein. Dividing precursor cells of the embryonic cortical ventricular zone have morphological and molecular characteristics of radial glia. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 22(8):3161–73, April 2002.

142. Stephen C. Noctor, Verónica Martnez-Cerdeño, Lidija Ivic, and Arnold R. Kriegstein. Cortical neurons arise in symmetric and asymmetric division zones and migrate through specific phases. *Nature Neuroscience*, 7(2):136–144, January 2004.

143. Jamie K. Paik and Robert J. Wood. A bidirectional shape memory alloy folding actuator. *Smart Materials and Structures*, 21(6):065013, June 2012.

144. Dejan Pecevski, Thomas Natschläger, and Klaus Schuch. PCSIM: a parallel simulation environment for neural circuits fully integrated with Python. *Frontiers in Neuroinformatics*, 3:11, 2009.

145. S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, January 1989.

146. Sabina Pfister. Inference of developmental motifs in the developing mouse cerebral cortex. (To be published).

147. F. Polleux, C. Dehay, H. Kennedy, et al. The timetable of laminar neurogenesis contributes to the specification of cortical areas in mouse isocortex. *The Journal of comparative neurology*, 385(1):95–116, 1997.

148. F. Polleux and W. Snider. Initiating and growing an axon. *Cold Spring Harbor Perspectives in Biology*, 2(4):a001925–a001925, April 2010.

149. Adria Pontious, Tom Kowalczyk, Chris Englund, and Robert F. Hevner. Role of intermediate progenitor cells in cerebral cortex development. *Developmental neuroscience*, 30(1-3):24–32, January 2008.

150. D. J. Price, H. Kennedy, C. Dehay, L. Zhou, M. Mercier, Y. Jossin, A. M. Goffinet, F. Tissir, D. Blakey, and Z. Molnár. The development of cortical connections. *European Journal of Neuroscience*, 23(4):910–920, 2006.

151. P. Rakic. Radial versus tangential migration of neuronal clones in the developing cerebral cortex. *Proceedings of the National Academy of Sciences of the United States of America*, 92(25):11323–11327, December 1995. PMID: 8524778 PMCID: PMC40392.

152. P. Rakic. The radial edifice of cortical architecture: from neuronal silhouettes to genetic engineering. *Brain research reviews*, 55(2):204–219, 2007.

153. Pasko Rakic. Evolution of the neocortex: a perspective from developmental biology. *Nature Reviews Neuroscience*, 10(10):724–735, October 2009.

154. Brian G. Rash and Elizabeth A. Grove. Area and layer patterning in the developing cerebral cortex. *Current opinion in neurobiology*, 16(1):25–34, March 2006.

155. A. T. Rasmussen. The percentage of the different types of cells in the male adult human hypophysis. *The American Journal of Pathology*, 5(3):263–274, May 1929. PMID: 19969849 PMCID: PMC2007246.

156. Anne J. Ridley, Martin A. Schwartz, Keith Burridge, Richard A. Firtel, Mark H. Ginsberg, Gary Borisy, J. Thomas Parsons, and Alan Rick Horwitz. Cell migration: Integrating signals from front to back. *Science*, 302(5651):1704–1709, May 2003.

157. F. Roegiers and Y. N. Jan. Asymmetric cell division. *Current opinion in cell biology*, 16(2):195–205, 2004.

158. William J. Rosoff, Jeffrey S. Urbach, Mark A. Esrick, Ryan G. McAllister, Linda J. Richards, and Geoffrey J. Goodhill. A new chemotaxis assay shows the extreme sensitivity of axons to molecular gradients. *Nature Neuroscience*, 7(6):678–682, May 2004.

159. Fabian Roth. *Explicit design, and adaptation in self-construction.* PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, 2007.

160. Fabian Roth, Hava Siegelmann, and Rodney J. Douglas. The self-construction and -repair of a foraging organism by explicitly specified development from a single cell. *Artificial Life*, 13(4):347–368, October 2007. ACM ID: 1288841.

161. Kristine Roy, Kathleen Kuznicki, Qiang Wu, Zhuoxin Sun, Dagmar Bock, Gunther Schutz, Nancy Vranich, and a Paula Monaghan. The Tlx gene regulates the timing of neurogenesis in the cortex. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 24(38):8333–45, September 2004.

162. Elizabeth F. Ryder, Lindsey Bullard, Joel Hone, Jonas Olmstead, and Matthew O. Ward. Graphical simulation of early development of the cerebral cortex. *Computer Methods and Programs in Biomedicine*, 59(2):107–114, May 1999.

163. Yuichi Sakumura, Yuki Tsukada, Nobuhiko Yamamoto, and Shin Ishii. A molecular model for axon guidance based on cross talk between Rho GTPases. *Biophysical Journal*, 89(2):812–822, August 2005.

164. David C. Samuels, H. G. E. Hentschel, and Alan Fine. The origin of neuronal polarization: A model of axon formation. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 351(1344):1147–1156, September 1996.

165. Jerome N. Sanes and John P. Donoghue. Plasticity and primary motor cortex. *Annual Review of Neuroscience*, 23(1):393–415, 2000.

166. D. Schubert, R. Kotter, Heiko J. Luhmann, K. Zilles, and J. F. Staiger. Cell type-specific circuits of cortical layer IV spiny neurons. *Journal of Neuroscience*, 23(7):2961–2970, 2003.

167. Stephen M. Schwartz. The definition of cell type. *Circulation Research*, 84(10):1234–1235, May 1999.

168. A. Sessa, C.-A. Mao, G. Colasante, A. Nini, W. H. Klein, and V. Broccoli. Tbr2-positive intermediate (basal) neuronal progenitors safeguard cerebral cortex expansion by controlling amplification of pallial glutamatergic neurons and attraction of subpallial GABAergic interneurons. *Genes & Development*, 24(16):1816–1826, August 2010.

169. Yaki Setty, Irun R. Cohen, Yuval Dor, and David Harel. Four-dimensional realistic modeling of pancreatic organogenesis. *Proceedings of the National Academy of Sciences*, 105(51):20374–20379, December 2008.

170. David J. Sharp, Gregory C. Rogers, and Jonathan M. Scholey. Microtubule motors in mitosis. *Nature*, 407(6800):41–47, September 2000.

171. Qin Shen, Yue Wang, John T. Dimos, Christopher A. Fasano, Timothy N. Phoenix, Ihor R. Lemischka, Natalia B. Ivanova, Stefano Stifani, Edward E. Morrisey, and Sally Temple. The timing of cortical neurogenesis is encoded within lineages of individual progenitor cells. *Nature Neuroscience*, 9(6):743–751, May 2006.

172. Troy Shinbrot. Simulated morphogenesis of developmental folds due to proliferative pressure. *Journal of Theoretical Biology*, 242(3):764–773, October 2006.

173. Julie A. Siegenthaler, Amir M. Ashique, Konstantinos Zarbalis, Katelin P. Patterson, Jonathan H. Hecht, Maureen A. Kane, Alexandra E. Folias, Youngshik Choe, Scott R. May, Tsutomu Kume, Joseph L. Napoli, Andrew S. Peterson, and Samuel J. Pleasure. Retinoic acid from the meninges regulates cortical neuron generation. *Cell*, 139(3):597–609, October 2009.

174. Lloyd M. Smith. Nanostructures: The manifold faces of DNA. *Nature*, 440(7082):283–284, March 2006.

175. V. Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

176. ISO-International Organization for Standardization. ISO/IEC 2382-15. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=7257, 1999.

177. Kaoru Sugimura, Kohei Shimono, Tadashi Uemura, and Atsushi Mochizuki. Self-organizing mechanism for development of space-filling neuronal dendrites. *PLoS Comput Biol*, 3(11):e212, November 2007.

178. Mriganka Sur and John L. R. Rubenstein. Patterning and plasticity of the cerebral cortex. *Science (New York, N.Y.)*, 310(5749):805–10, November 2005.

179. Michael A. Sutton and Erin M. Schuman. Dendritic protein synthesis, synaptic plasticity, and memory. *Cell*, 127(1):49–58, October 2006.

180. Michael A. Sutton, Nicholas R. Wall, Girish N. Aakalu, and Erin M. Schuman. Regulation of dendritic protein synthesis by miniature synaptic events. *Science*, 304(5679):1979–1983, June 2004.

181. Maciej H. Swat, Susan D. Hester, Ariel I. Balter, Randy W. Heiland, Benjamin L. Zaitlen, and James A. Glazier. Multicell simulations of development and disease using the CompuCell3D simulation environment. In Ivan V. Maly, editor, *Systems Biology*, volume 500 of *Methods in Molecular Biology*, pages 361–428. Humana Press, 2009.

182. D. H. Tanaka, S. Mikami, T. Nagasawa, J. Miyazaki, K. Nakajima, and F. Murakami. CXCR4 is required for proper regional and laminar distribution of cortical somatostatin-, calretinin-, and neuropeptide y-expressing GABAergic interneurons. *Cerebral Cortex*, 20(12):2810, 2010.

183. Victor Tarabykin, A. Stoykova, N Usman, and P. Gruss. Cortical upper layer neurons derive from the subventricular zone as indicated by Svet1 gene expression. *Development (Cambridge, England)*, 128(11):1983–93, June 2001.

184. M. Tomita, K. Hashimoto, K. Takahashi, T. S. Shimizu, Y. Matsuzaki, F. Miyoshi, K. Saito, S. Tanida, K. Yugi, J. C. Venter, and C. A. Hutchison. E-CELL: software environment for whole-cell simulation. *Bioinformatics*, 15(1):72–84, January 1999.

185. Michinori Toriyama, Yuichi Sakumura, Tadayuki Shimada, Shin Ishii, and Naoyuki Inagaki. A diffusion-based neurite length-sensing mechanism involved in neuronal symmetry breaking. *Molecular Systems Biology*, 6, July 2010.

186. Krasimira Tsaneva-Atanasova, Andrea Burgo, Thierry Galli, and David Holcman. Quantifying neurite growth mediated by interactions among secretory vesicles, microtubules, and actin networks. *Biophysical Journal*, 96(3):840–857, February 2009.

187. A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

188. A. M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, 237(641):37–72, August 1952.

189. F. Valverde and M. V. Facal-Valverde. Postnatal development of interstitial (subplate) cells in the white matter of the temporal cortex of kittens: A correlated golgi and electron microscopic study. *The Journal of Comparative Neurology*, 269(2):168–192, March 1988.

190. Arjen van Ooyen. Using theoretical models to analyse neural development. *Nat Rev Neurosci*, 12(6):311–326, June 2011.

191. Arjen van Ooyen and David J. Willshaw. Competition for neurotrophic factor in the development of nerve connections. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 266(1422):883–892, July 1999.

192. Jaap van Pelt and Andreas Schierwagen. Morphological analysis and modeling of neuronal dendrites. *Mathematical Biosciences*, 188(1-2):147–155, 2004.

193. Jaap van Pelt and Harry B. M. Uylings. Growth functions in dendritic outgrowth. *Brain and Mind*, 4(1):51–65, 2003.

194. J. von Neumann. Theory of self-replicating automata. *Urbana, Illinois: University of Illinois Press*, 1966.

195. J. von Neumann. First draft of a report on the EDVAC. *Annals of the History of Computing, IEEE*, 15(4):27–75, 1993.

196. Conrad Hal Waddington. *The strategy of the genes: a discussion of some aspects of theoretical biology.* Allen & Unwin, 1957.

197. Tong Wang, Ruojie Sha, Rmi Dreyfus, Mirjam E. Leunissen, Corinna Maass, David J. Pine, Paul M. Chaikin, and Nadrian C. Seeman. Self-replication of information-bearing nanoscale patterns. *Nature*, 478(7368):225–228, October 2011.

198. Scott Wasson. AMD's dual-core opteron processors - the tech report. http://techreport.com/articles.x/8236/1, April 2005.

199. George M. Whitesides and Mila Boncheva. Beyond molecules: Self-assembly of mesoscopic and macroscopic components. *Proceedings of the National Academy of Sciences of the United States of America*, 99(8):4769–4774, April 2002. PMID: 11959929 PMCID: PMC122665.

200. Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, 1994.

201. H. Wichterle, M. Alvarez-Dolado, L. Erskine, and A. Alvarez-Buylla. Permissive corridor and diffusible gradients direct medial ganglionic eminence cell migration to the neocortex. *Proceedings of the National Academy of Sciences*, 100(2):727–732, 2003.

202. N. Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971.

203. N. Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, 1976.

204. Sheng-Xi Wu, Sandra Goebbels, Kouichi Nakamura, Kazuhiro Nakamura, Kouhei Kometani, Nagahiro Minato, Takeshi Kaneko, Klaus-Armin Nave, and Nobuaki Tamamaki. Pyramidal neurons of upper cortical layers generated by NEX-positive progenitor cells in the subventricular zone. *Proceedings of the National Academy of Sciences of the United States of America*, 102(47):17172–7, November 2005.

205. Jun Xu, William J. Rosoff, Jeffrey S. Urbach, and Geoffrey J. Goodhill. Adaptation is not required to explain the long-term response of axons to molecular gradients. *Development*, 132(20):4545–4552, October 2005.

206. K. W. Yau. Phototransduction mechanism in retinal rods and cones: The Friedenwald lecture. *Investigative Ophthalmology & Visual Science*, 35(1):9–32, January 1994.

207. Céline Zimmer, Marie-Catherine Tiveron, Rolf Bodmer, and Harold Cremer. Dynamics of Cux2 expression suggests that an early pool of SVZ precursors is fated to become upper cortical layer neurons. *Cerebral cortex (New York, N.Y. : 1991)*, 14(12):1408–20, December 2004.

208. G. A. Zimmerman, D. E. Lorant, T. M. McIntyre, and S. M. Prescott. Juxtacrine intercellular signaling: another way to do it. *American journal of respiratory cell and molecular biology*, 9(6):573–577, December 1993. PMID: 7504925.

209. F. Zubler and R. Douglas. Simulation of corticogenesis as a self-organizing system. *Frontiers in computational neuroscience*, 3, 2009.

210. F. Zubler and R. Douglas. An instruction language for the explicit programming of axonal growth patterns. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–4, July 2010.

211. Frederic Zubler and Rodney Douglas. A framework for modeling the growth and development of neurons and networks. *Frontiers in Computational Neuroscience*, 3:25, 2009.

212. Frederic Zubler, Andreas Hauri, Roman Bauer, Sabina Pfister, John C. Anderson, Adrian M. Whatley, and Rodney J. Douglas. Simulating cortical development as a self constructing process: a novel multi-scale approach combining molecular and physical aspects. (Submitted for publication). 2013.

213. Frederic Zubler, Andreas Hauri, Adrian M. Whatley, and Rodney Douglas. An instruction language for self-construction in the context of neural networks. *Frontiers in Computational Neuroscience*, 5:57, 2011.