Monod

A biologically-inspired computational model

Mathieu Gagné

Monod

Table of Contents

1 Introduction 2
- 1.1 Orientation 2
- 1.2 Goals 3
  - 1.2.1 To Study the Future of Computation 4
  - 1.2.2 To Study Biology 5
- 1.3 Overview of Monod 6
  - 1.3.1 The Design Stack 8
  - 1.3.2 Overview of the Results 10
- 1.4 Other Biologically-Inspired Approaches 10
  - 1.4.1 Genetic Programming 11
  - 1.4.2 Neural Networks 12
  - 1.4.3 Molecular Computing 12
  - 1.4.4 Artificial Immune Systems 13
- 1.5 So what is Monod 13
  - 1.5.1 Disclaimers 13

2 Three Design Patterns 15
- 2.1 The Hive Design Pattern 15
- 2.2 The Swarm Design Pattern 17
- 2.3 The Incubator Design Pattern 20
  - 2.3.1 The Incubator Calculator Example 21
  - 2.3.2 Another example 24
  - 2.3.3 Some Definitions 24
  - 2.3.4 Turing Soup 26

3 The Cytoplasm and the Monod Cell 34
- 3.1 Biological Inspiration 34
- 3.2 Proteins 39
  - 3.2.1 Domains 43
    - 3.2.1.1 Simple Ligand Binding Domain 47
    - 3.2.1.2 Ligand Binding Domain with Remapping 48
    - 3.2.1.3 Logical Integration Domain 50
    - 3.2.1.4 Boolean Multiplexor Domain 51
  - 3.2.2 Protein Construction and Properties 51
    - 3.2.2.1 Behavior Functions 52
    - 3.2.2.2 Realization Functions 53
    - 3.2.2.3 Proteins, finally 56
- 3.3 The Cytoplasm 56
  - 3.3.1 Compartments 56
  - 3.3.2 Release 56
  - 3.3.3 Energy 57
- 3.4 The Monod Cell 57

4 Monod Cultures 59
4.1 Evolution Driver 59
4.2 Cheating 59

5 Results and Future Projects 60

6 Compilation and Usage 61
6.1 Compilation and Installation 61
6.1.1 Prerequisites 61
6.1.1.1 Hardware 61
6.1.1.2 make 62
6.1.1.3 OCaml 62
6.1.1.4 Texinfo 62
6.1.1.5 Perl 62
6.1.2 Compilation 63
6.2 Documentation 63
6.2.1 Manual 63
6.2.2 Source Code Documentation 64
6.3 Usage 64
6.3.1 Command-line Arguments 64
6.3.2 Test Suite 65
6.3.3 The singlecelltest Executable 67
6.3.4 Provided Examples 67
6.3.4.1 incubcalc 67

7 Implementation Details 68
7.1 Contributing to Monod 68
7.2 Development Timeline 69
7.2.1 History 69
7.2.2 Future 69
7.3 Source Code Structure 70
7.3.1 SourceForge Administration 73
7.4 Architecture 73
7.5 Logging 73
7.6 Simulation-Specific Algorithms 75
7.6.1 The Swarm Design Pattern Implementation 75
7.6.1.1 Serialized Swarm 75
7.6.1.2 True Threaded Swarm 76
7.6.1.3 Distributed Swarm 76
7.6.2 Cytoplasm Topology Implementation 76

Appendix A Combinations 77
A.1 Two Ligand Binding Sites 77
A.1.1 Without Remapping 77
A.1.1.1 Truth-table Driven Logical Functions 77
A.1.1.2 Other Logical Functions 87
A.1.2 With Remapping 87
A.1.2.1 Truth-table Driven Logical Functions 87

Appendix B References 88

Concept Index 90

1 Introduction

Monod is an abstract computational model inspired by cellular microbiology. In Monod, a program is not a linear sequence of instructions, but a set of simple programlets which operate on each other and on data according to well-defined rules and stochastic forces, in analogy with proteins and nucleotide sequences in a cell. Monod is also a software implementation of this computational model on standard computer hardware and so provides an accessible software laboratory with which one can run experiments.

Monod should naturally accomodate parallel processing, and fits very nicely in the context of evolutionary algorithms alongside genetic programming, where it offers homologous crossover, among other aspects. The basic principle upon which Monod is premised is that biological cells perform computations. The underlying computational model seems to possess many desirable qualities, like high parallelism, adaptability and tolerance of complexity. These qualities are thoroughly lacking in traditional computational paradigms. Monod offers an opportunity to understand the origin of these qualities, their relationships and perhaps to deduce useful lessons.

The name **"Monod"** should be pronounced the same as the word "mono", since the last 'd' is silent. It refers to Jacques Monod, the celebrated, Nobel-prize-winning microbiologist who participated in the discovery of basic cell regulation mechanisms, allostery and messenger RNA, to name a few contributions. With his frequent colleague Fran\cois Jacob, he made many predictions, some dating from before the discovery of the structure of DNA, concerning the operational control of gene expression. Most predictions have been validated over time. He is also the author of a wondrous book about the philosophy of biology, *Le Hasard et la Necessité: Essai sur la philosophie naturelle de la biologie moderne* (*Chance and Necessity: Essay on the natural philosophy of modern biology*), published in 1970 [*Monod 1970*].

In the rest of this chapter, we present an overall introduction to the project, its goals and an overview of the large-scale design. Then we contrast Monod with other existing biologically-inspired computational approaches. We present a quick overview of the results, a history and future prospects. Finally, we answer the question: "What *is* Monod?"

1.1 Orientation

This documentation accompanies the Monod program, and describes the purpose, principles, usage, design and implementation, results and future prospects of this program. Monod is an open-source project, released under the GNU Public License (GPL), and can be downloaded in its entirety from SourceForge at `http://monod.sourceforge.net/`.

The present documentation collates all information (short of the source code) concerning Monod, from purpose to theory to usage to implementation. It serves as a *brain dump* for the maintainer(s) and developer(s) to make sure good ideas don't get forgotten (hence the perpetual disarray) before someone has time to implement them, and as a repository of failed attempts. It also serves as a reference manual for those aspects of Monod that would otherwise be forgotten and unmaintainable. Finally, and most importantly, the documentation can be used to quickly ramp up with the current state of the project and get in gear to provide much needed collaboration!

We now present an overview of the documentation. The current chapter provides an introduction to the Monod project, including the initial motivation for the project; the large-scale architecture of Monod; comparisons with other biologically-inspired computational approaches; and a history and description of the major future milestones of the project.

The second chapter, Chapter 2 [Three Design Patterns], page 15, describes the first three layers at the bottom of the hierarchical design stack introduced in the previous chapter.

The next chapter, Chapter 3 [The Cytoplasm and the Monod Cell], page 34, gives details of the full abstract computational model. We present those biological processes which served as inspiration for Monod and the model proper, contained in the cytoplasm, which contains formalized computational analogues of these biological processes. The Section 3.2 [Proteins], page 39 section is one of the most central sections of the entire documentation. Then we wrap the cytoplasm with a set of tools to make it into a self-contained system, the Monod Cell.

The fourth chapter, Chapter 4 [Monod Cultures], page 59, is the last chapter which includes non-implementation aspects of the model, and is the culmination of the project. It shows how evolutionary techniques may be applied to the Monod model in order to create more complex and robust "programs" that are intrinsically parallel.

The next chapter, Chapter 5 [Results and Future Projects], page 60, discusses in some detail many of the experiments that have been attempted with Monod, and others that are planned, expanding on the previous overview in the first chapter. The examples run from the simple to the complex. Those examples that are complete are included in the standard Monod distribution, as described in the next chapter.

The sixth chapter, Chapter 6 [Compilation and Usage], page 61, describes the pre-requisites needed to compile and/or run the various parts of Monod. The various '`Makefile`' targets are described, along with their products and how they can be run.

The final chapter, Chapter 7 [Implementation Details], page 68, describes the code used to implement Monod. The first section (Section 7.1 [Contributing to Monod], page 68) is useful if you're planning to contribute to the project, which is highly encouraged! The other sections describe various aspects of the implementation, from the language and data structures to overall architecture and testing.

1.2 Goals

Monod was started by the maintainer as *something to do* to assuage his curiosity and nagging preoccupations with

- **The future of computation**. Current programming approaches are apparently unable to deal with such issues as complex problems, parallel programming and bug prevalence, leading to the suspicion that programming just shouldn't be done the way it's done today.
- **Biology**. Certain categories of biological phenomena, namely, cellular biochemistry and evolutionary mechanisms, that seem to possess strong analogies with *computation*.

A hallucinatory flash originally hinted that the one may be somehow related to the other (yes, in a new way, as described below), and Monod is the attempt to *do something about it*, to flesh out the flash. The rest of this section describes further the two starting points above. The rest of the book describes the solution.

1.2.1 To Study the Future of Computation

The traditional computational model — let's call it the Turing / von Neumann model — is precise, fast, practical from an industrial point of view and cognitively approachable. However, it also suffers from defects. It does not lend itself naturally to parallelism, to the creation of complex yet robust programs or to the creation of adaptive programs.

We believe that computation is in its prehistory. To paraphrase Arthur C. Clarke, computation as it will be done in the future would appear to us today as magic. The magic would cover both the creation of computational tools or programs and our interaction with these tools.

The latter concern, how humans interact with programs, is clearly a major focus of contemporary software engineering. Graphical user interfaces, input/output devices, privacy and data safety concerns are all under tight scrutiny. Who knows what the future holds? Natural-language driven interfaces, 3D visualization glasses or cyborg implants: the stuff of science fiction yesterday. The rewards for new and improved human interfaces can be great, as software is now a part of so many people's lives.

In contrast, the first concern mentioned above, the task of helping the process of software creation, has received comparatively scant attention. Certainly the dismal state of commercial software and, by deduction, of its development, is a common lament among programmers and others. There are also a number of efforts to assist the task of the programmer, covering methodologies (extreme programming, design patterns), modeling languages (UML), powerful development frameworks that go beyond IDEs (intentional programming) and formal design verification programs (Alloy). However, the lot of software engineers is not a mainstream topic of conversation or even a preoccupation of science fiction writers!

All of the efforts cited in the previous paragraph are important and some of them will be fruitful. But none seem to address fundamental questions: How do we unleash creative power in the act of programming? How do we make simple things simple to do and difficult things less difficult? How do we create interesting and even *surprising* programs? (While humans crave surprise, it is a very bad thing indeed in today's software engineering world.) How do we avoid errors — or better yet, avoid *caring* about errors? And just *how* do we even begin to create a program which can pass the Turing test?

Whatever the precise answers to these questions, we believe that the following will be salient features:

- There will be **tight coupling between the developers and the computers** during the engineering phase(s). The computer will participate actively in the creation of software, from requirements to debugging. The focus of the human will not be on "code" anymore. The interactions will be done on human basis. (Hence there is a feedback loop between the capability to interact with programs and the capability to create them.)
- Programs will be **naturally parallelizable** in that their operational characteristics such as performance will benefit from increased processing power, without requiring human intervention. This is certainly not the situation today, when creating or adapting programs to run on mutiple processors is a tremendous pain.
- Programs will be able to **adapt** in different ways. Whether it be adapting to the quirks of different users, tolerating errors, self-healing and building immunity, or learning entire domains of competence from a clean but appropriate slate, there's no doubt that

programs need to feature the ability to change. This ability is close to nil today for most commercial products, other than multiple choice configurations.

Central to adaptability are *weak algorithms*, also called *algorithmic templates* or *context-free algorithms*. Instead of hard-coding a knowledge domain in a program, setting up a *meta-program* which can acquire this domain through the exercise of a weak algorithm is often preferable. Examples of weak algorithms abound, such as "make a first guess, and then refine your guess" or "divide the problem into subproblems". Applications include a natural language translator which *learns* how to translate from language `A` to language `B` simply through a generic statistical analysis.

- Programs will **tolerate complexity**. Complexity is present in the number of features of a program, in the intrinsic complexity of its domain matter and in its operational characteristics such as clustering and fault-tolerance, for instance. Already today, the complexity of a program is a significant limiting factor. The limit exists in great part because humans shoulder the entire burden of the complexity and this is not something they do very well. The resulting products manifest large numbers of bugs, expensive and distended release cycles and short lifetimes.

Of course, these features are related. It is difficult to imagine a close interaction between a human and a program without adaptability and the tolerance of complexity, parallelizability and adaptability probably introduce complexity on their own, while it may be difficult to achieve adaptability and the tolerance of complexity without parallelizability, and so forth.

In the above we have used the word "program" in a very general sense. We certainly do not mean "source code" the way it is understood today, or the resulting binary executables. Rather, we refer to a tool created to accomplish a particular computational purpose. Unfortunately, there is no extant adequate terminology. This lack is unfortunate, because the word "program" implies *programmability*. We definitely think that the deterministic programmability of computers will gradually disappear. Nevertheless, we will often stick to the word "program" for lack of a better one.

How does all of this relate to Monod? A goal of Monod is to try to understand the last three characteristics of the future of computation mentioned above, namely: natural parallelizability, the tolerance of complexity and adaptability. Monod provides a computational model which, at first glance, accomodates these characteristics. Note that Monod is categorically not a production-grade tool. Rather, it is an attempt to learn and understand.

1.2.2 To Study Biology

There's a lot going on in biology at present, in many different areas. A lot of activity has to do with various aspects of computation. Certain algorithms or algorithmic templates have been identified which are reused across many vastly different domains. For instance, evolution, as a general principle, takes place among genes, among immune system molecules and, more controversially, among spatio-temporal neural firing patterns within the brain ([*Calvin 1996*], [*Edelman 1988*]). As another example, consider the Baldwin effect, whereby phenotypic plasticity smooths out the fitness landscape and may accelerate genotypic evolution, operates at vastly different scales, from bacteria to mammals [*Dawkins 1982*, chapter 9]. As a final example, note how *modularization* appears in so many different contexts in biology: the modularization introduced by cellular organelles, by the organs of an animal, by the areas of the cortex. Also of note is that modularization in biology is almost always *leaky*.

To an interested outside observer, reading or even browsing the standard cellular biology textbook [*Alberts et al. 2002*] brings a surprise at every page, to the effect that "Wow, I had no idea we knew how this works!". If in addition the reader is interested in the nature and principles of computation, then the surprises are compounded by the feeling that much of what happens in a cell has to do with... computation.

The particular aspects of cellular biology which have been singled out as "computationally relevant" are given later, in Section 3.1 [Biological Inspiration], page 34. Most of them are easily accessible in the first quarter of the textbook cited above, which describes them from a biology perspective with the occasional computer analogy:

> "The Regulation of Cdk and Src Protein Kinases Shows How a Protein Can Function as a Microchip" [*Alberts et al. 2002*, title page 179].

How does all this relate to Monod? A goal of Monod is to provide a practical test bed for computational analogues of some of the mechanisms identified as computationally relevant. Monod allows the user to vary operational parameters or to entirely turn off certain features in order to ascertain the impact on the computational ability of the model. For example, what is the impact of various degrees of modularization on the model? What is the significance of leakiness?

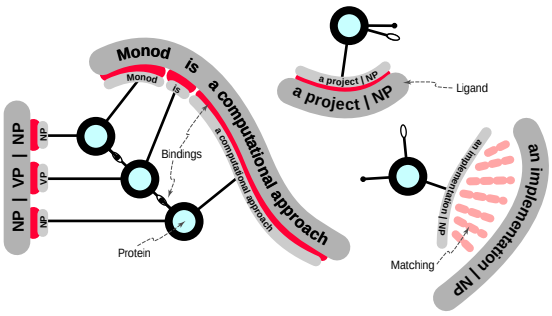
1.3 Overview of Monod

The present documentation describes in detail the computational approach of Monod. We present here an overview. This overview takes the form of a mental image.

The traditional Turing / von Neumann model posits a single *program* acting on *data* in a sequential, well-defined fashion. The action consists of different types of operations, such as arithmetic, logical operations and branch operations.

In contrast, the Monod model, called the Monod Cell, consists of a multitude of small *programlets*, which we call *proteins* by analogy with biology, all acting on the same data at the same time and interacting with one another. The data consist of strings, imagined floating along with the proteins, and are called *ligands*. The basis of interactions between and among proteins and ligands is *matching*, whereby active sites on the various entities recognize each other. Matching leads to *binding* and triggers actions. The actions of the proteins on ligands include the modification, breakage and linkage of segments of ligands. The actions of proteins on each other include modification of internal state. For instance, proteins may act as regulators of other proteins through expression of certain characteristics or repression of others. (As we mentioned earlier, Jacques Monod was instrumental in the

discovery of regulation mechanisms between proteins. This aspect of our program is why it is named after him.) The following figure displays the elements discussed above.



The protein actions, including recognition, all take place in parallel. There is a restriction that each binding site on string or on a protein may participate in at most a single binding. While each protein's behavior is completely deterministic, the bindings are not, so that a program consisting in a set of proteins may act non-deterministically. This non-determinism may be either a feature or a bug, depending on the intention.

The proteins in the Monod model are assembled out of a finite number of building blocks according to precise instructions. The buildings blocks, called domains, can be combined in myriad ways to create proteins which perform different actions. The set of available domains is rich enough that the computing model is Turing complete. In fact, the model looks like a soup of Turing machines working on the same set of tapes. At the same time, the domains are individually powerful enough that one speaks of *combinations* of domains, in a way that one does not speak of the source code of a program as a combination of letters. This makes it possible to practically consider the domains as evolutionarily *terminal* and apply genetic algorithms to Monod programs.

The Monod Cell also includes other computational mechanisms inspired by biology. The intake and processing of data strings is the analogue of the cell feeding and transforming its intake into useful products or rejecting them. These processing steps affect an overall energy level, which is closely linked with the fitness of the cell. The cell can be subdivided into compartments, and elements (data or proteins) shuttled from one compartment to another, or from one cell to another, to increase computational performance. And so on.

FIXME: Insert graphic

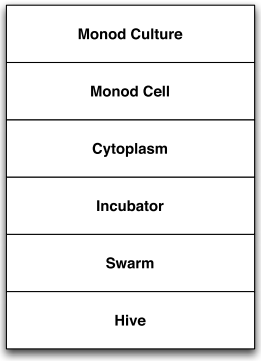
Finally, Monod lends itself naturally to evolutionary algorithms. The task performed by a Monod Cell is specified by that cell's *genome*. The genome is then the analogue of a computer program in the Turing / von Neumann world. Perhaps because of the fact that Monod genomes are strongly inspired by the world of biology, they fit very well in the contect of evolutionary algorithms. Evolving Monod genomes is analoguous to evolving programs using, say, genetic programming. However, Monod genomes possess many desirable properties which make them possibly more suitable to these techniques.

In the rest of this section, we introduce the a hierarchical structure which helps explain the Monod approach and implementation. We also present an overview of the results obtained using Monod.

1.3.1 The Design Stack

The Monod computational approach is somewhat more complex than, say, a Turing machine. However, it is fairly easy to describe as the top of a hierarchy of structures. We introduce in the next section this hierarchy. Happily, the implementation of the Monod Cell model has been made to closely mirror this abstract *design stack*. Hence the uses of the hierarchy are twofold: to reduce the explanatory load on the reader and to provide a good framework for the implementation.

The stack can be seen in the following image, where each layer uses and is more complex than the one below it.



The bottom three layers are useful and fairly generic *design patterns*. Above them, the Cytoplasm model contains the complete computational model, properly speaking. The Monod Cell adds some practical gewgaws. Finally, the Monod Culture is a framework for applying evolutionary algorithms to Monod Cells. We present a quick overview of the various layers. More details for each layer may be found in dedicated sections in further chapters.

The Hive

The Hive is a design pattern that we use to abstract away the computational difficulty of maintaining a multitude of independent execution threads. A Hive contains *residents*, which are simply programs wrapped in an object. The Hive can be seen as a fairly simple scheduler for the residents, which can be assimilated to threads. Many considerations have led to the adoption of the Hive design pattern rather than using the underlying threading model of the Operating System (OS). The Hive is described in Section 2.1 [The Hive Design Pattern], page 15.

The Swarm

The Swarm is another design pattern built on top of the Hive and introduces formalized interactions between the residents. Residents of a swarm *expose* certain *projections*, and these projections are tagged with *markers*. Markers have variable affinities for one another, and may trigger *bindings* between the projections that are propagated to the control flow of resident and may affect its behavior. Bound projections may also be released, and markers altered.

Binding is the most fundamental principle driving Monod. It is introduced very early, and further refined in the higher layers of the design stack.

Details are given in Section 2.2 [The Swarm Design Pattern], page 17.

The Incubator

An Incubator is a Swarm where the residents are of two kinds: *ligands* and *processing units*. Ligands are akin to simple passive text strings. Processing units can be arbitrarily complex, but their projections operate in a fashion similar to text strings and regular expressions, binding with each other and with ligands. Furthermore, processing units (*procunits*, for short) may alter the ligands by changing the string, cutting them up, binding them, etc.

An analogy can be made by imagining a *soup* of Turing machines, with the machines binding, releasing and reassembling the tapes floating in the soup according to their individual programs and to their binding markers. Another analogy is with biological cells, identifying processing units with proteins and ligands with DNA and/or RNA strings.

The Incubator is described in more details in Section 2.3 [The Incubator Design Pattern], page 20, along with this analogy.

The Cytoplasm

The Cytoplasm model builds on the Incubator design pattern by introducing several concepts inspired by cell microbiology. It is not called a design pattern anymore because of its lack of generality. The most significant difference introduced in the cytoplasm is that processing units — now called *proteins* — are made out of a finite set of *domains*, each of which performs a specific action, such as calculating a logical function or altering a ligand based on a logical input. Domains must be assembled, lego-like, to create valid, fully-functional proteins, and different proteins may be creating by different combinations of the same domains.

Other innovations also appear in the Cytoplasm. For instance:

- some protein operations consume *energy*;
- certain ligand patterns are identified as *poisonous* and others as *nutrients*;
- all residents (proteins and ligands) have *densities* that vary along a *geometry* of the cell;
- bindings between and across proteins and ligands are subject to a stochastic *release mechanism*.

The Cytoplasm incorporates all the details of the computational model of Monod. It is described in detail in Section 3.3 [The Cytoplasm], page 56.

The Monod Cell

The Monod Cell complements the Cytoplasm with a *nucleus* to coordinate protein synthesis and introduce a *compilation* step from a *genetic representation* of the proteins to the proteins themselves (akin to the transcription/translation process that takes place in the eukaryotic cell); and a *membrane* to coordinate ligand transport to and from outside the cell. Many hooks are also added to be able to control and monitor processing within the cell. The Monod Cell

is essentially a Cytoplasm equipped with a practical "interface" that can be used by an appropriately design *harness* to run experiments. It is described in Section 3.4 [The Monod Cell], page 57.

The Monod Culture

The Monod Culture is the culmination of the project. One of the original inspirations for Monod is to provide a convenient breeding ground for *programs*. A Monod Culture is a glorified harness that employs evolutionary algorithms to alter the genomes of Monod Cells according to various programmable policies and, hopefully, find something, er, better. The Monod Culture framework is described in Chapter 4 [Monod Cultures], page 59.

Much of the interesting work envisioned consists in running experiments with different Monod Cultures. Some of these results are described in Chapter 5 [Results and Future Projects], page 60, and alluded to in Section 1.3.2 [Overview of the Results], page 10 below.

1.3.2 Overview of the Results

The previous section presented the rudiments of the Monod model as well as its implementation. Monod also consists of the use of this implementation to run experiments.

FIXME: Clearly, there's nothing here yet...

What is an appropriate program in the Monod world? Not a shuttle guidance system; not a financial analysis package; not a database; etc. Rather, programs that have some of the same requirements as those for which neural networks are appropriate, which have a special emphasis on string manipulation. Refer to AIS, page 218.

What would be a happy outcome of the Monod project? An "interesting" compiler "evolved" in Monod from basic building blocks (with a little help).

FIXME: The Baldwin effect! Previous simulations: Hinton and Nowlan. FIXME: The Baldwin effect as very central. In fact, there was a long hesitation between the names Baldwin and Monod. A primary goal is to see a Monod culture develop enough plasticity to exhibit the Baldwin effect. FIXME: fundamental driving problem: the Baldwin effect parable to lead to the "invention" of "concepts" or "variables" in the genotype

1.4 Other Biologically-Inspired Approaches

There are many other biologically-inspired computational approaches besides Monod. Not surprisingly, our understanding of biology has increased in parallel with our ability to create more powerful computational machines. There have been many interactions in both directions between these two fields.

> "As a result of these bilateral interactions between computing and biology, it is possible to identify three different approaches, namely *biologically motivated computing*, *computationally motivated biology* and *computing with biological mechanisms*. [...] [In the first approach] Biology provides sources of models and inspiration for the development of computational systems (e.g., ANN [Artificial Neural Networks] and EC [Evolutionary Computing]). In the second approach, computing provides models and inspiration for biology (e.g., ALife and CA [Cellular Automata]). The last approach involves the use of information

> processing capabilities of biological systems to replace, or at least supplement, the current silicon-based computers (e.g., Quantum and DNA computing)." [*de Castro and Timmis 2002*, p. 3].

Monod falls in the first approach named above.

In this section, we quickly contrast Monod with three other biologically-motivated computational approaches (genetic programming, artificial neural networks and artificial immune systems) and one approach which involves computing with biological mechanisms (molecular computing). Beyond being biologically-motivated in one sense or another, all of the approaches cited here also share with Monod a basic concern with the the triad of attributes mentioned in Section 1.2 [Goals], page 3: natural parallelizability, tolerance of complexity and adaptability. Some of these approaches have met with great success and are already an important part of the most advanced computational techniques available (artificial neural networks and evolutionary algorithms).

1.4.1 Genetic Programming

Genetic programming is one of many different computational approaches which attempt to harness the power of mathematical algorithm of evolution to solve problems. Many of these approaches have met with great success. The *creative power* of evolution has been amply demonstrated. Most evolutionary techniques require a very precise fit between between the problem space and the solution search space. This is no different for the particular subfield of genetic programming. However, there solutions are represented by actual executable computer programs, which are executed to calculate their fitness. The program may take many different forms, such as trees or linear sequences of instructions.

The programs are evolved by repeatedly applying to them a number of *evolutionary operators* according to a particular algorithm (there are many such possible algorithms). These operators include: a *mutation* operator, which changes the solution more of less randomly; a *crossover* operator, which combines many different solutions to create new ones; an *evaluation* operator, which calculates the fitness of the many different solutions — that is, how well they solve the problem at hand; a *selection* operator, which chooses a number of solutions from the present generation to participate in the next one. These operators are combined and sequenced according to many different, personal recipes. An exposition is presented in [*Banzhaf et al. 1998*].

Genetic programming has seen many successes. It has been used successfully in image and pattern recognition, robot control and data mining, for example.

Nevertheless, many aspects of genetic programming have been disappointing to even its proponents. Perhaps the most important ones have to do with the amount of tailoring that needs to be done to the evolutionary operators listed above (more specifically, mutation and crossover) to make them apply to computer programs. In particular, the crossover operator is the subject of intense scrutiny. A chapter in [*Banzhaf et al. 1998*] is titled "Chapter 6: Crossover — The Center of the Storm". From this chapter:

> In nature, most crossover events are successful — that is, they result in viable offspring. This is a shart contrast to GP crossover, where over 75% of the crossover events are what would be termed in biology "lethal" [*Banzhaf et al. 1998*, p. 157].

Among the differences noted between biological crossover and GP crossover are the following three (also from the same source):

- Biological crossover takes place only between members of the *same* species.
- Biological crossover occurs with remarkable attention to preservation of "semantics".
- Biological crossover is *homologous*.

The last property is often appealed to as a source of the creative power of crossover. While GP crossover lacks these three properties, in the Monod Cell the operator can easily accomodate them. This is because a Monod Cell program is defined by a genome consisting of a set of separate *Monod genes*, each defining an individual programlet or *Monod protein*. By adopting a computational approach explicitly inspired by the biology of the cell, we obtain, for free, powerful extra properties.

Another essential distinguishing factor between GP and Monod is that, by its definition, Monod creates parallelizable programs.

FIXME: More here.

1.4.2 Neural Networks

FIXME: Lack of "regulation" leads to prohibitive computational cost

FIXME: Add Elman as reference

1.4.3 Molecular Computing

Molecular computing is an endeavor to exploit the computational abilities of biological systems using non-classical substrates, mainly biological molecules themselves. Like in Monod, *binding* — also known as *molecular recognition* — is a fundamental principle of molecular computing:

> "Ignoring for the present the question as to whether proteins are the ultimate optimal mechanism or whether nature (and evolution) simply used what was available, it should be pointed out that a very important aspect is the dependence of biological systems for their "information processing" capabilities on what is known as *molecular recognition*. Molecules bind weakly with other molecules... This recognition is, at base, a quantum effect and is one of the mechanisms by which parallelism is introduced into the system." [*Sienko et al. 2003*, p. xv]

The mention of quantum physics is meaningful. There is much discussion and controversy as to whether the physical substrate of computation is relevant. Without a doubt, quantum effects play a significant role in the microbiology of the cell, as in molecular recognition above. Whether these effects or other heretofore unknown effects are *essential* is what's at issue. There are positions in molecular computing on either side of the fence. Some simply advocate that the "independence of a specific representation makes it possible to use new concepts for the computational process, based on real elements such as chemical reactions and quantum mechanical devices, or on virtual elements such as cellular automata and populations of artificial genes" [*Gramss et al. 1998*, p. 1], while others claim that "it is the physical characteristics of material systems — whether they be relatively simple chemical systems or material in biological cells — that allow highly complex information processing to occur" [*Sienko et al. 2003*, p. xii]. But the issue is irrelevant for our purposes

At the very least, it is recognized that the substrate plays a significant role in the *performance* of the computations:

> "Biological systems in nature are clearly highly evolvable. In principle, it should be possible to use a structurally programmable machine to simulate the structure-function plasticity that allows for this evolvability. ... But this comes at a computational cost; the computational work required to simulate plastic structure-function relations puts a severe practical limit on the degree of evolvability that can be retained." [*Sienko et al. 2003*, p. 5]

And also:

> "Enzymes, as catalysts, are thermodynamically reversible; their pattern-recognition work is free, driven only by the heat bath." [*Sienko et al. 2003*, p. 11]

As we have already mentioned, Monod is not meant as a production-grade system to do computations. Hopefully the shortcomings — including the performance ones — will leave some wiggle room for exploration.

Monod does not advocate substrate independence even while the project consists in a simulation on traditional computing hardware. The main goal is the isolation and exploration of certain computational principles that may yet play a role in traditional Turing / von Neumann machines, as explained in Section 1.2 [Goals], page 3 earlier. Hence, much of the biological inspiration that applies to molecular computation can be used for the Monod project as well.

1.4.4 Artificial Immune Systems

FIXME: Content

1.5 So what is Monod

"Monod" refers to different things:

- **Monod is a computational approach**. This approach is strongly inspired by biology and its goals have been described earlier in this chapter. In this sense, Monod is entirely described in this manual.
- **Monod is an implementation** of this computational approach. This implementation runs on traditional computer hardware, and can be used to run experiments. These experiments and their results presumably reflect properties of the computational approach.

  The Monod implementation is open source and contributions are most welcome.
- **Monod is a project** where experiments concerning the Monod computational approach are performed using the Monod implementation.

The word "Monod" will be used to refer to these different aspects indiscriminately.

1.5.1 Disclaimers

To the descriptions of Monod above, one needs to add that **Monod is in progress**. Both code and documentation are in an early planning and prototyping phase. The main consequence is that the project is recruiting contributors. If you are interested in contributing in any

form, please write to the maintainer quickly! See the section Section 7.1 [Contributing to Monod], page 68 below for more details.

Throughout the code and documentation, the '`FIXME:`' tag indicates material that is specifically tagged for future revision — and the language accompanying it is probably going to be rather cryptic. Furthermore, to paraphrase the Securities and Exchange Commission, this documentation contains *forward looking material* — it describes anticipated results and not-yet-implemented aspects of the code base.

Monod is principally a source of *fun* and *surprise*, done during the maintainer(s)'s free time, and should be viewed as such, with whatever grain of salt this entails in the reader's mind. It is not subsidized by either a government grant, academia or by a private company, and no peer-reviewed articles have been published about it to date.

Have a little fun.

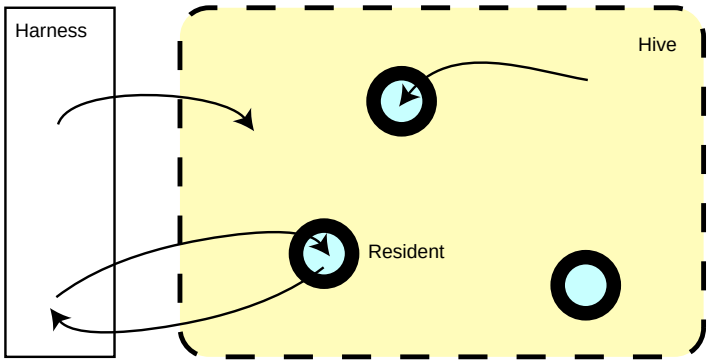
2 Three Design Patterns

A Monod Cell — or more specifically, the Cytoplasm within it — must keep track of a multitude of semi-independent, interacting objects, the ligands and the proteins. The design stack is a gradual build-up to the ultimate functionality needed. The bottom three layers are fairly generic *design patterns* in the context of concurrent programming. While they have a distinctive "biological" flavor (even forgetting the names), each one can actually can be used quite independently from the rest of the Monod implementation (see the examples in the '`testing`' directory).

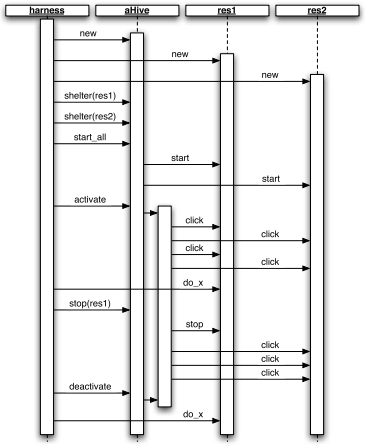
The three patterns described here are described abstractly, without reference to an implementation (or by referring to many possible implementations). The implementations used in the Monod code base are described in Chapter 7 [Implementation Details], page 68. While *special-purpose design patterns* sounds like an oxymoron, and new design patterns should in general be avoided, these serve the purpose not only of reuse or validation, but also of explanation - they help explain an abstract layers of the Cytoplasm, and the serialization strategy that was employed for the implementation.

2.1 The Hive Design Pattern

The Hive design pattern simply encapsulates running multiple threads of execution — it's as simple as that. A Hive object manages the execution of *resident objects*. A resident object should have a `click` method. Residents can be *sheltered* and *removed* from the Hive, and they can be *started* and *stopped*. When in the started state, the `click` method of a resident is called repeatedly by the Hive. The `click` method indicates through its boolean return value whether the resident should be called again in the next round. If not, the resident can be *woken* by external means later to put it back in the pool of running residents. A *harness* is responsible for creating the Hive and the various residents, sheltering them and managing the overall flow. The following figure is a representation of a Hive, harness and three residents.



The sequence diagram below shows the basic calling sequence required to operate a Hive.



Most operating systems do threading very well and the hive relies on it. An extra abstraction was deemed necessary for two reasons: first, to guarantee of *time consistency* across the multiple residents; second, so that the model is abstract enough for the implementation to be changed easily in order to incorporate different threading models easily without affecting the higher layers of the design stack.

Time consistency is provided by guaranteeing that the `click` method is called fairly uniformly across all residents. This ensures that the different time lines of the residents are more or less in sync.

The order in which the `click` method is called on different residents by the Hive is *random*, or, at least, unpredictable from the point of view of the residents. This is very similar to how OSes schedule threads today, of course. However, in the context of Monod, where individual residents play a role more akin to subroutines than to threads, this randomness is significant. We will revisit it in due time.

The Hive, as a design pattern, needs much support in order to be actually useful. Specifically, it needs residents and a harness. We've already discussed the residents. The harness is also essential. It performs different roles:

- It is responsible for starting and stopping the Hive;
- It can add and remove residents;
- It can provide a well-known, common meeting point for the residents.

The harness can be as simple as a wrapper to allow for direct interaction from a human person, or it can be a very complex management program. The most important thing that it can *not* control is the scheduling of the resident execution, which is strictly the responsibility of the Hive.

The Hive abstraction is flexible enough to allow for many different implementations. For instance, the residents can run in a single OS thread, with the Hive acting as scheduler; or the residents can be spread across multiple threads (possibly one per resident); or they can be spread across multiple machines. Indeed, the current implementation of the Hive is the first example just cited, is an extremely lightweight implementation which does not

consume much OS resources. In Monod, all the residents are very simple and require very little processing power, but fairly constant attention. However, if all goes well and Monod is indeed interesting, the plan is to expand it to a multi-platform implementation with remote method invocations. Much of the work would be concentrated in the Hive, and not in the rest of the design stack.

The residents individually do not need to be thread-safe — the `click` method will always return before being called again on the same resident. However, a resident should not assume anything about other residents since, as described above, there could be many threads running.

Hence, the Hive captures the *essentially parallel* aspect of the Cytoplasm. "Parallel" because many operations are conceivably happening at the same time. "Essentially" because the parallelism needs to be *flattened* or *serialized* to run on a standard computer.

2.2 The Swarm Design Pattern

The Swarm design pattern extends the functionality of the Hive by adding formalized interactions between the residents. Each resident may have zero, one or more *projections*, which are fixed for the lifetime of the resident and are individually identifiable by the resident code. Each projection is *tagged* with a *marker* structure, which is initialized and changeable at the discretion of the resident. Each projection can be either *active* or *inactive*.

Given two active markers, the Swarm can calculate a list of *matchings* between these two markers. There can be no matching (in which case we say the projections don't match), a single matching (an unambiguous match), or many matchings (ambiguous matchings — for instance, if a regular expression matches many substrings of a string). Each matching is further associated with an *affinity*, which is an integer representing the strength of the matching. This integer is non-zero if and only there is a match between the two markers.

A particular Swarm collects all the markers from all the active projections of all the residents, and calculates and attempts to execute the matchings. Executing a matching means turning it into a *binding* by calling an appropriate method on each of the two residents involved. If both method calls are successful, the binding is made and both projections are removed from the list of active projections, so that they will not participate in further matchings. If either method call is unsuccessful, the binding is not made. Bindings can be released later on at the request of either resident, or of the harness. Any projection can be made active or inactive at the whim of the resident code. Making a bound projection inactive triggers an automatic release.

If there is more than one matching involving the same projection with the same affinity, *a single one is picked at random*. The randomness introduced here injects a probabilistic flavor to this pattern, which suffuses the rest of Monod in an essential way, adding to the random `click` scheduling introduced in the Hive. We will return to this aspect many times throughout the documentation.

FIXME: Insert graphic.

The probabilistic nature of matching operates with the following two constraints. First, if there is any matching possible, a binding operation will be triggered within a bounded time (FIXME: can we specify "one click"?). Second, the probability of a matching is proportional to the affinity between the two markers. As the affinity can not be zero if there is a matching, we are assured that all matchings have some chance to become bindings.

Collectively, the types of the projections, markers, matchings and bindings are called the *exposure* of the Swarm. They can be defined independently from the residents.

Like the Hive, the Swarm requires a *harness* to control its execution. In addition to being responsible for the activation state of the Swarm and for adding and removing residents, the Swarm harness has an additional ability: it can order the release of bound residents. This behavior can be as simple as not doing anything, since residents are perfectly capable of releasing their bindings on their own. Or it can be significantly more complex, as we will see in the Cytoplasm layer (see Section 3.3 [The Cytoplasm], page 56).

In real biological systems, proteins interact with each other first by recognizing each other through a matching process. This *molecular recognition* process is an extremely complex physical phenomenon, relying on quantum mechanical and thermodynamic effects, and has sometimes been called *Brownian search* to emphasize its computational role. An analogy which has been made often (since the late 19th century) is with the fit between *a lock and a key*. However, the Swarm presents a much more symmetrical view of the matchers than what lock and key might lead one to imagine. Projection matching in the Swarm is but a very simplified analogue of molecular recognition, and we can only hope that we've extracted some of the essence of the process rather than missed it altogether. Certainly, Monod does not benefit from the performance afforded by a real physical system due to the highly parallel nature of the microbiology — at least as long as it is implemented on traditional computing hardware. However, maybe we can still achieve some of the goals stated in the introduction.

The Swarm design pattern should be used when a large number of programs should be thought of as executing concurrently, and interacting with each other. A Swarm can be the *foundation* for many different developments: a host of interacting and self-regulating agents, with one or more distinguished residents serving as scratch pads; Selfridge's Pandemonium; a Hive, if no resident has any projection; an artificial life competition where small programs vie with each other for dominance; or finally, for the Incubator model, discussed next.

We now give more precise details of the abstract Swarm model. The interested reader should refer to the signature source file '`swarm.mli`', which contains the abstract definitions of a Swarm, with no reference to the implementation. Like other parts of the Monod project, the Swarm is a *simulation-dependent algorithm*. Details of the implementation can be found in Section 7.6.1 [The Swarm Design Pattern Implementation], page 75.

FIXME: Precise definitions and details.

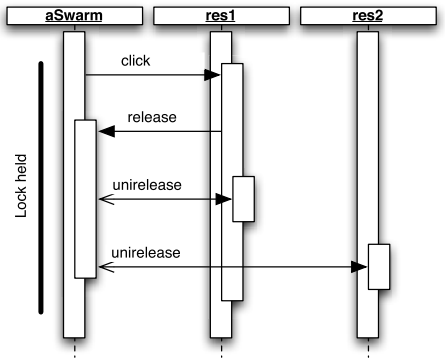
The binding operation between two residents can be decomposed into two distinct steps: *docking* and *propagation*. Both steps are independently visible to the residents. Docking is the initial step of binding. It is a sort of handshake between both residents. The result of a successful docking is that the residents have indeed agreed to bind, and a *binding object* has been computed. The binding object contains enough information to undo the binding. Docking may fail for any reason, at the discretion of either resident. In case of failure, the state of the system is restored to what it was before the binding initiation.

When docking is complete, propagation is initiated on each of the two residents. Propagation is guaranteed to be called atomically with docking. That is, no other operation (clicking or other binding) can have been launched involving either of the residents. During the propagation phase, the residents are free to use the information encoded by the binding. For instance, they can trigger its release, which they could not do during the docking phase

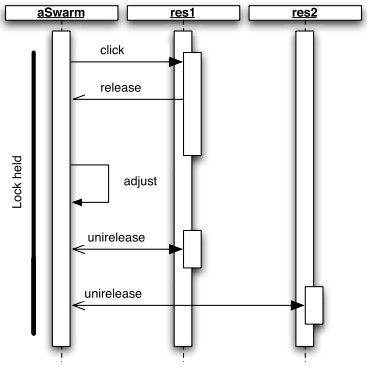
since the binding was not yet consumated. However, in contrast to docking, propagation is expected to never fail.

FIXME: Insert sequence diagram

The most subtle aspect of the Swarm model concerns the release of bindings. A binding may be released in two different ways: the release can be ordered by the harness or can be ordered by one of the two residents involved in it. In the latter case, the resident may be in one of many different hooks, namely, its `click` method, `unimate` or even in `unirelease`, so that we have a recursive cascade of releases. The subtlety comes from the fact that when a resident orders a release, the resident is called back to change its state in order. If the resident is called while it is in the middle of its `click` method, for instance, then we get a complicated calling structure. The following sequence diagram is a representation of this structure.



This calling sequence structure violates our principle that resident code be logically single-threaded. The state of the resident during execution becomes difficult to understand. Hence, the Swarm model calls for a different release calling sequence, which does not thread the resident code. When a resident demands a release, from its `click` method or any other part of the resident code, the demand is simply queued by the Swarm and executed *after* the caller has returned. This new calling structure is shown in the following sequence diagram.



The "adjust" call represents the Swarm structure changing its internal state to reflect the release. Hence, the Swarm is updated *before* the resident `unirelease` methods are called. This calling sequence should be kept in mind when creating Swarm residents.

2.3 The Incubator Design Pattern

The Incubator design pattern does not quite *extend* the Swarm, but rather restricts the exposure used and the residents that are allowed. Firstly, Incubators use the same kind of exposure, which is based on ASCII strings. And secondly, only two kinds of residents are allowed: on the one hand, passive *ligands*, which are simple strings; and on the other hand, *processing units*, which are general programs that can perform certain operations on ligands. We discuss these two topics in parallel.

A ligand is an ASCII string. As a resident of the Swarm underlying an Incubator, it exposes a single projection, whose marker is a simple ASCII string. Processing units can recognize fragments of this string (as described later) and bind to it. When a processing unit is bound to a fragment of a ligand, that fragment can not participate in further matching operations until the binding is released. However, many processing units may be bound to the same ligand, as long as the fragments they're binding to do not overlap.

Processing units are general residents of the underlying Swarm. They can expose multiple projections. However, the projections fall into one of two kinds: *ligand binding projections*, which bind to ligands as discussed above, and *structural binding projections*, which bind to other processing units.

The marker type of ligand binding projections on a processing unit is the *matcher*, which is just a regular expression able to bind to fragments of ligands.

The marker type of structural binding projections on a processing unit is the *snippet*. A snippet is a pair consisting of a string and a matcher. Two snippets `s1 = (str1, mat1)` and `s2 = (str2, mat2)` match if and only if `str1` matches `mat2` and `str2` matches `mat1` — that is, the strings and regular expressions must match crosswise. Snippet matching captures an essential biochemical feature: when proteins interact, they both need to recognize a region of their companion, while at the same time expressing a complementary region.

FIXME: Insert graphic.

FIXME: Woops — explain how the *affinity* in the Incubator is computed, depending on the lengths of the strings which are being matched.

Processing units are otherwise general-purpose residents. They are free to react to binding and release actions, and can emit orders to the Incubator as well to release ligands, change markers, change ligands, etc. More precisely, in a processing unit to ligand interaction, the processing unit may:

1. Release the ligand.
2. Change the string underlying the ligand.
3. FIXME: Split the ligand and/or join ligand fragments.
4. FIXME: "Slide" along the ligand.

In a processing unit to processing unit interaction, each processing unit may:

1. Release the other processing unit.
2. FIXME: Create a communication "channel" with the other side for arbitrary communication?

For more information, see the signature source file '`incubator.mli`', which contains an implementation-free description of the capabilities of the Incubator layer.

In addition to the residents (ligands and processing units), the Incubator needs a harness in order to be fully operational. The harness has the same abilities as in the Swarm. We enumerate them again here:

- It can start and stop the Incubator;
- It can add and remove residents — both ligands and processing unit;
- It can serve as a "base camp" for all the processing units, so they may communicate with a common object;
- It can order the release of any particular binding.

We proceed to give a simple application example of the Incubator, and a quick overview of a possible way to interpret the Incubator, the "Turing Soup".

2.3.1 The Incubator Calculator Example

We give details of an application implemented using an Incubator. It consists of a simple calculator which is able to perform arithmetic computations. The Incubator is fed a string (in the form of a ligand) which contains numbers and arithmetic operation symbols (`+`, `-`, `*` and `/`, along with parentheses), and *reduces* the string to its result by the action of multiple arithmetic processing units, if the string is well-formed. When reduced to a single number, the string is output as the result. Many processing units participate in the computation. We list them here:

- There is a processing unit type which detects **multiplications**. It simply matches expressions of the form *<num> * <num>*, and reduces them to the product.
- There is a processing unit type which detects **additions** and **subtractions**. Its matcher is a little bit more complex, because of precedence rules: it matches expressions of the form *[+|-|(|BEGIN]<num>[+|-]<num>[+|-|)|END]*, and reduces them to the result.
- There is an **end detector**, which recognizes a single number by matching *BEGIN<num>END* and signals the end of the computation to the harness, with the result being equal to the number matched.
- There is a **parenthesis eliminator** processing unit, which simply eliminates parentheses around simple number tokens, matching *(<num>)* and reducing it to *<num>*.
- FIXME: Currently we don't allow division — we want to stay in the realm of integers, for fun, and do rational arithmetic, reducing an expression to its simplest form. But this requires a lot more processing units, and some thinking, to be frank... We'll get there later.

Note that the processing units above only include ligand matching projections, and never match between themselves.

The harness required for the calculator is very simple. It first instantiates the Incubator and populates it with the appropriate processing units. It then prompts a human user for a string to evaluate, attaches *BEGIN* and *END* markers to the string, and feeds it to the Incubator. Finally, it waits for the end of computation to be signaled by the **end detector**, or for a specified timeout. It then cleans up and loops again from the beginning.

For example, when fed with the string

```
(2 + 2) * (3 + 2 * 6 + 5)
```

it will output 80. "We've come so far!", you exclaim. Indeed, this calculator probably wins the prize for the most lines of code written to perform simple arithmetic computations. However, one should not lose sight of the fact that the entire Incubator apparatus is independent of the application. The "calculating" part of the program is indeed quite small, and consists in the specifications of the individual processing units and a harness for user interaction. This specification may be compared with giving an input file to a `yacc`-type program to specify the calculator.

There are three main differences, however, with how such a "classical" calculator would be programmed. Namely: the Incubator-based calculator goes through an arguably unpredictable sequence of states; it is intrinsically parallel; its behavior in the face of an invalid input string is to go in a loop. We examine each of these characteristics in turn.

Consider the example above. A possible reduction sequence is the following:

```
(2 + 2) * (3 + 2 * 6 + 5)
(4) * (3 + 2 * 6 + 5)
4 * (3 + 2 * 6 + 5)
4 * (3 + 12 + 5)
4 * (15 + 5)
4 * (20)
4 * 20
80
```

However, another, slightly different reduction sequence is possible as well:

```
(2 + 2) * (3 + 2 * 6 + 5)
(4) * (3 + 2 * 6 + 5)
4 * (3 + 2 * 6 + 5)
4 * (3 + 12 + 5)
4 * (3 + 17)
4 * (20)
4 * 20
80
```

(There are other orders as well.) As far as the processing units are concerned, which sequence is executed first is impossible to predict. For any classical implementation of the Incubator (on traditional PC hardware, for instance), the order is well-determined, but it is buried in the Incubator — even deeper, in fact, since the Incubator relies on the OS's random number generator to determine the matching order.

Obviously, since the execution order is impossible to predict (beyond the clues provided by matching), it needs to be immaterial. This requirement applies to the writer of the processing units. It can be verified easily that the processing units described above satisfy this criterion for any input ligand. We discuss this further in the next section.

Consider the following initial sequence:

```
(2 + 2) * (3 + 2 * 6 + 5)
(2 + 2) * (3 + 12 + 5)
```

There are three possibilities to consider as the next step. Furthermore, some of those possibilities are *independent* in the sense that they can take place at the same time without affecting the result. In fact, if we consider that two processing unit interactions take place

at the same time, we can go in a single step from the last state above to either one of the following two states:

```
(4) * (15 + 5)
```

or

```
(4) * (3 + 17)
```

Of course, parallelization and sequence unpredictability are closely related in that parallelizability also implies an awareness on the part of the resident programmer. In fact, both are manifestations of the same principle: that there is no well-defined time line in the Incubator, and no need for one. Rather, there is a *partial ordering* on the various events that can take place.

Finally, let's consider the behavior of the above-defined Incubator in the face of an invalid input ligand. For instance, suppose we feed the string

```
5 + 3 * + 2 + 1
```

The Incubator can perform one reduction to

```
5 + 3 * + 3
```

but no processing unit matches any fragment of this new string. In this case, the Incubator simply stalls — the ligand, in some sense, is not *digestible* (we'll come back to this analogy later when we discuss the Cytoplasm). However, all is not lost — there are different ways to make this situation explicit. One is to add further processing units which detect invalid conditions and abort the computation, reporting an error. For instance, all of the following patterns are invalid:

```
+ END
- END
* END
BEGIN +
BEGIN *
* +
+ *
- *
<num> (
) <num>
...
```

This makes for a lot of extra processing units (or for a single, compound one which matches different patterns), and it's difficult to figure out if we've really exhausted the entire list of possibilities without going through a formal analysis of the reductions and possible inputs. Furthermore, this solution is biologically improbable, a subject to which we will return.

Another possibility to help detect an invalid ligand is to have the Incubator notify some resident when further matching is impossible - to have some kind of *default* behavior when all the registered matches fail. This solution is akin to what one would do with a traditional `yacc`-type input file. However, one difficulty with this scheme is that instead of having a situation where matching is impossible, we may have an infinite *cycle*. It can be shown easily that there are no cycles possible in the above Incubator (the effective string length diminishes with every processing unit application — FIXME: This may change when introducing rational division), but we can envision easily a situation where they are possible.

Imagine that we add a processing unit to the Incubator which embodies *commutativity of addition*. It detects patterns of the form *[+|-|BEGIN|()<anything1>+<anything2>[+|-|END|)]* and switches the order of the operands (note that *anything* is not quite anything — needs to contain balanced parentheses, for one). Then the above sequence can quickly change into

```
3 * + 5 + 3
3 * + 8
```

and thence oscillate between the latter string and

```
8 + 3 *
```

In this case, matching will always be possible, which makes the situation difficult to detect, but no "progress" is made. Of course, this situation is very much akin to an infinite loop in traditional programming. Detecting these oscillations is in fact a sub-problem of the halting problem for general Turing machines, hence is theoretically intractable. It is up to the resident programmer to ensure that the set of residents in the Incubator does not encode such cycles. Note that the cycles can take place even if the input ligand is valid (as is the case with the above commutation processing unit, for instance).

2.3.2 Another example

FIXME: Insert here another simple example with snippet matching, to introduce the idea of *regulation*.

2.3.3 Some Definitions

Let us make a number of definitions. We call a set of processing units a *program*. (Duplicate processing units in the same program very well *may* affect the outcome of the computation - they will certainly affect performance.) An Incubator equipped with a program but containing no ligand is called a *primed* Incubator. An Incubator is said to be *at rest* if there is no matching possible. It is said to be *grounded* if there are no bindings. A program is *well-grounded* if and Incubator primed with the program is at rest and grounded when there are no ligands present in the Incubator. We concentrate initially on well-grounded Incubators because their dynamics are simpler to explain.

The notion of *input* for an Incubator is easy to define, at least for a well-grounded Incubator. The input to an Incubator is simply a ligand which is fed to the Incubator. However, the *output* is somewhat more ambiguous to define. When fed with an input, a primed Incubator may perform operations indefinitely (we will relate this to the Halting problem in the next section). Hence we must make a few additional definitions.

A ligand is said to be *terminating* with respect to a certain program if an Incubator, primed with the program and initially grounded and at rest (if possible) eventually reaches rest again after being fed the ligand at input. Even in this case, however, it is not necessarily the case that there is an identifiable output to the Incubator: to identify an output is a role of the program, and it might fail to do so. Note that we can't simply refer to "the transformed input ligand", since that ligand might be transformed beyond identification (it might have been split into many fragments, it might be bound, it might simply be gone).

To identify an output to the Incubator is a collaborative task of the program and of the harness. The program must send some kind of signal to the harness, and allow the harness to extract the output (and, possibly, reset the Incubator to its ground state). A

ligand is *semantically terminating* with respect to a program and a certain harness if it is terminating and if it explicitly designates an output to the harness, as described above. We won't explore this notion further in the present chapter.

A program is said to be *behaved* if it is well-grounded and if *all* ligands are terminating with respect to it.

Even if a program is behaved, as above, it can act very badly: it can give different results at different times for the same input ligand, unpredictably, because of the probabilistic nature of matching in the underlying Swarm. For instance, consider two behaved programs `P1` and `P2`. It is easy to construct a new (behaved) program `P` which, when fed any ligand, will compute either `P1` or `P2` applied to it, unpredictably (FIXME: Show how).

It is important to realize that this non-determinism is a fundamental property of the Incubator — or rather, of the underlying Swarm. It is independent of the implementation of the Swarm layer. Even if the Swarm is implemented using completely deterministic algorithms — as it currently is, using a standard random number generator — the processing unit programmer can not rely on any determinism. The Incubator is completely decoupled from the program inasmuch as matching is concerned, which means the non-determinism *must* be assumed even if it is illusory. (In other words, "you can't rely on something you can't rely on".)

Rather, determinism becomes a *property* of the program, if we make the following definition: a behaved program is *deterministic* if it always reaches essentially the same end state at rest when fed, from the ground state, the same ligand. (FIXME: We need to characterize exactly what "essentially" means. If the program is semantically terminating, it means that the output is the same. But otherwise it's a little bit more fuzzy.)

Hence, this situation contrasts sharply with regular programming of modern computers, where determinism is never in question (except in case of breakdown). We claim that the non-determinism introduced by the matching process participates in a profoundly essential way in the other non-classical properties exhibited by Monod — adaptability, pattern recognition, etc., as we hope to show later. A so-called *trade-off principle* — between deterministic programmability, efficiency and adaptability — has been discussed elsewhere:

> "A system cannot at the same time be effectively programmable at the level of structure, amenable to evolution by variation and selection, and computationally efficient."

[Michael Conrad, quotation in *Sienko et al. 2003*, p. 146].

The attentive reader will have noticed that we have skirted a subtle point: is the property of being behaved actually deterministic? As we've defined it, there's no ambiguity: a behaved program always (deterministically) reaches rest. However, it is easy to conceive of programs which are *probabilistically behaved* but otherwise deterministic — that is, they terminate with a non-zero probability, but when they do, they always yield the same result. Indeed, for any deterministic program `P`, consider the alternative program `P'` which consists of `P` with the addition of a simple processing unit which matches any input ligand in its entirety and releases it immediately without modifying it. It is for this new program to stall by repeatedly binding with the new processing unit and never invoking the original program `P`. However, as soon as the original ligand is modified by `P` — which may happen at any time — the new processing unit is out of the loop, and the Incubator will reach the resting state prescribed by `P`.

We have avoided another issue, this one more important: how do we account for the possible role of the harness in releasing bindings? This role might be crucial. We have up to now assumed that the processing units are entirely responsible for the binding releases (as was the case in the calculator example above). At the other extreme, however, the harness could be responsible for all releases if the individual processing units *ask* it to perform the releases, for example. This makes all the above definitions useless — unless we recast them relative to a given harness. Hence, we introduce the notions of an *h-behaved* program, *h-deterministic* program, etc., all relative to a certain harness. Note that a program can be h-deterministic with respect to two different harnesses, yet yield different outputs for each harness — it can do so even if it is deterministic! The theoretical situation can thus occasionally be difficult to understand. The use of harness-controlled release will be made apparent when we discuss the Cytoplasm layer later on.

Let us consider the calculator example given in a previous section. The first incarnation of the calculator program, where the program `Pcalc1` consists of operational processing units only, is certainly well-grounded. All valid ligands are terminating (and in fact, are semantically terminating with respect to the harness provided) — however, invalid ligands are not terminating, so that the program is not behaved. Adding the processing units to handle all the invalid cases, as we do later, makes the new program `Pcalc2` behaved. It is also easy to verify that it is in fact deterministic - as it should be if it is to be a reliable calculator.

FIXME: Is determinism truly a desirable property, always?

FIXME: When we have another example, we have to discuss it as well.

2.3.4 Turing Soup

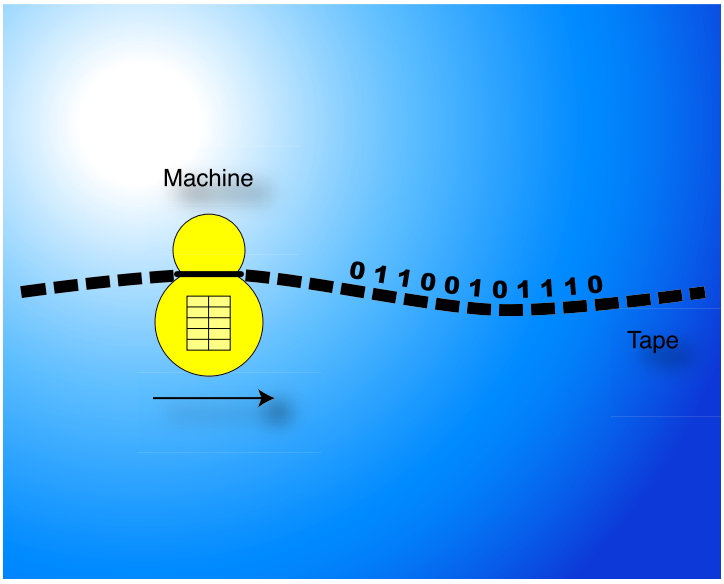
The Monod computational model can be described most easily if we show how it is different from the Turing / von Neumann computational model. We do this in several steps, keeping track of the relevance of each step to the end goal of the project: to provide a computational model naturally well suited to evolutionary algorithms.

In the abstract Turing machine model of computer execution and storage, a *tape* is read and modified by a *head* according to the *state* and the instructions stored in an *action table* on the machine.

The traditional implementation of this abstract model today is the von Neumann architecture, with storage for both data and the specification of the action table (the *program*), and a processing unit which executes the action table.

Despite many efforts, some of them even fairly successful, the Turing / von Neumann computational model does not lend itself naturally to techniques through which the program can be evolved using well-known algorithms adapted from biology. For instance, it is difficult to define *mutations* and *recombinations* of programs. This situation has left humans as

essentially the only *programmers* to date. However, programming is an inherently inhuman task.

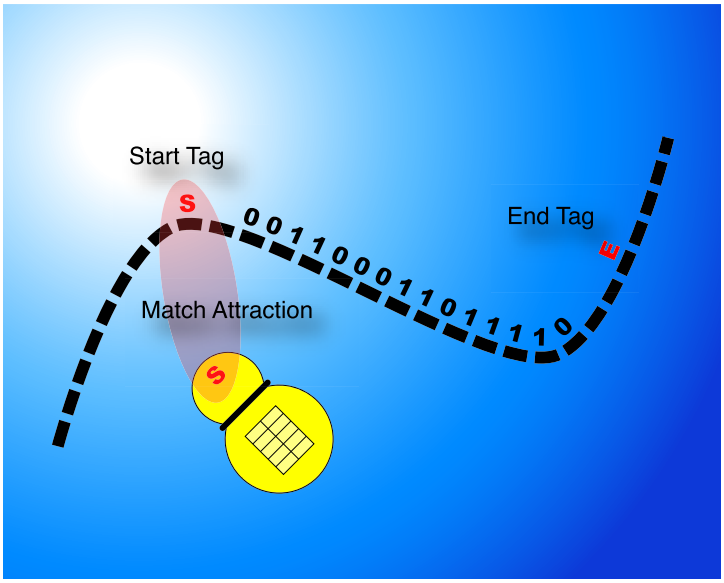


Imagining ligands and processing units may evoke the traditional picture of a Turing machine, with the ligand taking the role of the "tape" and processing units taking the role of the machine proper. One can either imagine a single processing unit *sliding* across the ligand in either direction (using the sliding commands described earlier FIXME: they don't exist yet), or a sequence of processing units matching and releasing the ligand and modifying it to insert a unique *mark* where appropriate, with the processing units always matching the mark. Hence, at the very least, the Incubator can simulate a Turing machine. Woohoo!

It's easy to imagine that the machine and the tape start out in a separated state. The machine is idle in this unattached state. It possesses a matching element which may or may not correspond to a specific matching site on the tape. Through some agency, the matching element and the matching site are bound, triggering processing of the tape by the machine, as above. The machine can be separated from the tape again, according to its action table, for example when encountering an end signal on the tape.

There is no functional novelty introduced here. This step is merely necessary to pave the way for the next one. However, a crucial implementation aspects should be pointed out. Finding appropriate matchings that can trigger bindings is, properly speaking, a computational task. However, it is not a task that is performed by the machines that are posited to be part of the Monod model. It can be thought of as being performed by the ambient medium — in any case, outside of the machines. A matching element is merely an indicator. Monod-the-computational-model, where matching can be taken for

granted, should thus be distinguished from Monod-the-implementation, where matching is a significant part of the program.



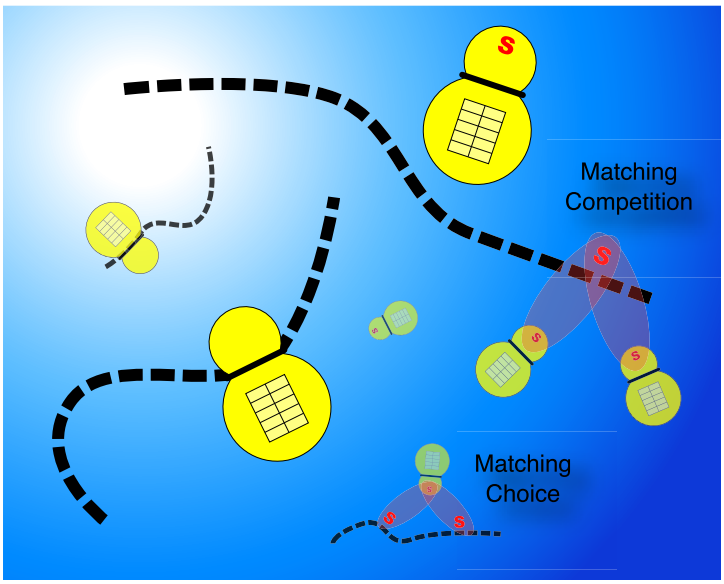
We now simply imagine that many different tapes and many different machines participate in communal execution and matching. There can be many different matching elements and matching sites. Many matching elements may compete for binding with a specific matching site on a tape, but only a single machine can be bound to a given site at any time. Conversely, many sites can compete for the attention of a single matching element on a machine. In such cases of competition, decisions are made stochastically.

The overall computation executed by the set of machines is not confined to any particular program, but is now distributed among the various *programlets* of each machine. Their action is coordinated through the matchings: a machine can create a matching site for another machine by changing the tape it is attached to; a machine can alter its matching element; the probabilistic nature of matching implies that the course of execution is not necessarily entirely determined by the set of individual programlets of all the machines; etc.

A most notable property of this program execution environment is that *conditional branching* can be (though does not *need* to be) eliminated from the repertory of instructions available to program each individual machine. Introduced very early on in its modern intent, the first description of conditional branching was fairly murky (Turing described it as an unconditional branch preceded by a computation which modified the code!). It is notoriously difficult to deal with branching in contemporary programming, because of the complexity it introduces, yet is indispensable — because of the complexity it introduces. Think of all the effort, both practical and theoretical, expended in white box tests, black box tests, debugging, dead branches, etc. The Monod computational model offers a very different way to think about conditional branching: indeed, a conditional branch can always be replaced by a tape/machine separation followed by the conditional binding of one of two machines depending on the state of the tape at the time of separation. FIXME: impact? On reusability, modularity, encapsulation — and more?

Another notable property of the environment is a natural suitability to parallel program execution. Many machines can be running at the same time on the same or different tapes.

The implications of this step on the evolvability of programs are significant. Conditional branching is a notoriously difficult instruction to deal with through evolutionary techniques — perhaps the difficulty can be mitigated by treating this instruction differently, as described above. The subdivions of a program into many programlets introduces the possibility of natural homologous recombination, which is absent from most current models of genetic programming. Programs evolved in this computational environment have a fighting chance of being natural candidates for parallel execution.



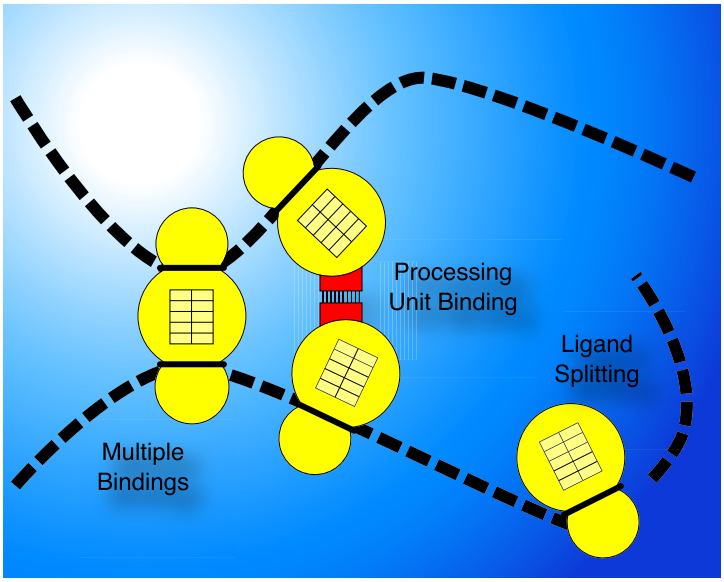
Parallelism is indeed apparent in the above mental image, since many "programs" can work at the same time on the same data set. However, the situation may appear rather unruly, with all sorts of thingies bumping against one another. Of course, this is what happens inside a real biological cell. But even more importantly, there's no reason that the Turing soup be fundamentally disordered. It needs to be "programmed" well, just like a regular Turing machine — that is, it needs an appropriate set of processing units — like the *deterministic* programs discussed in the previous section.

FIXME: Relate definitions in previous section with Halting problem for Turing machines.

In addition to the binding and releasing of tapes and moving along them, machines can be equipped with additional powers. For instance, tapes can be split and joined. Machines can be equipped with matching elements that match with other matching elements on other machines in order to affect both matchines' internal state. A machine can have more than a single matching element. And so on.

Wisely or unwisely, in order to define the operations allowed, we derive much inspiration from the molecular biology of cells. An analogy drives this inspiration, whereby machines are identified with proteins and tapes with polynucleotides (which we call ligands in the model).

From an evolutionary perspective, the operations available are a subset of the atomic instructions available to the programs in the Monod model. Having more complex operations available does not change the functional range of the model, but possibly significantly alters the operational characteristics such as speed of execution, speed of evolution and size of the programs.

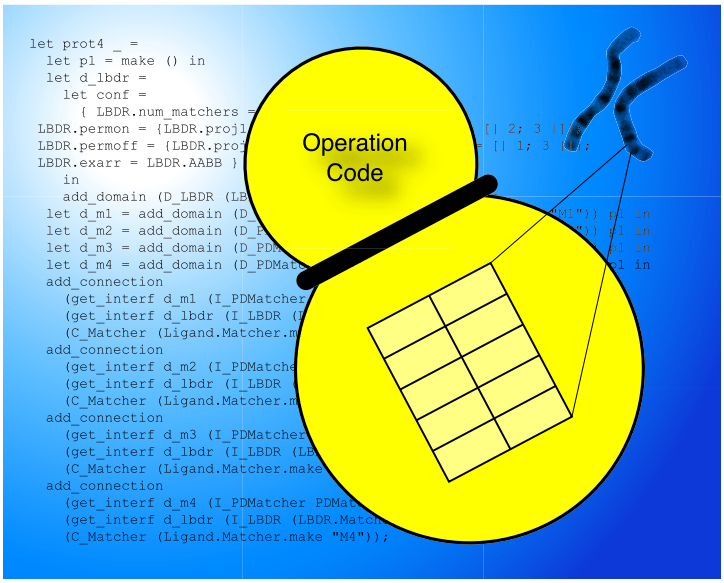


The specification of each machine includes the matching elements, the action table and how these two entities are related to one another. As with traditional Turing machines, there are different ways to define the specification of a machine in the Monod model: through an actual state table or through a program in a suitable language, for instance.

Each machine in the Monod model is assembled out of a finite set of specific domains, according to a blueprint for the machine. Each domain type has a specific function — for instance, a matching element domain, or an arithmetic operation domain — and can be connected to other domains through interfaces that it expresses — for instance, a matching element domain expresses a "I'm bound now" interface.

Constructing the machines out of domains provides yet another level of genetic flexibility to the Monod model. Each machine is described by a set of domains and a set of connections

between the interfaces of these domains. This description is akin to a gene and is highly amenable to controlled mutation and recombination.

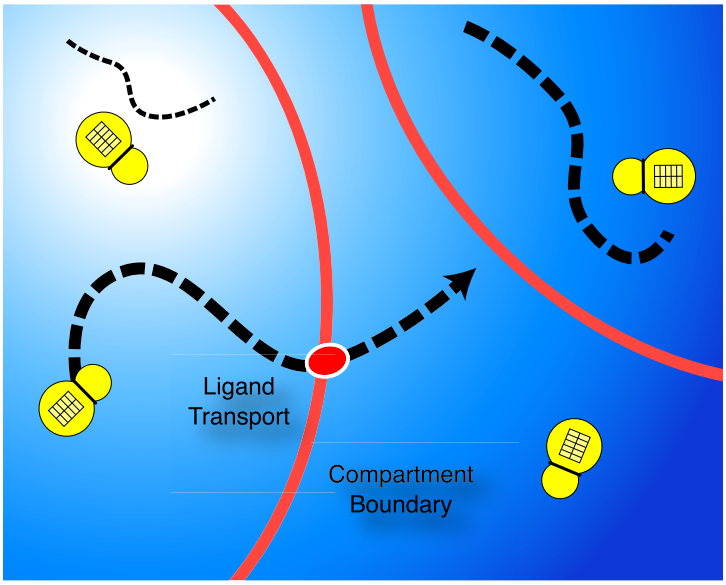


Binding and execution take place within bounded compartments. They are filled with a medium within which float the tapes and machines, which bump against one another, binding when matching is present.

Compartments may adjoin one another and tapes or machines may be transported from one to the other according to specific rules. Compartments may keep global computational variables, such as energy, which is depleted through certain operations and replenished by the presence of the end-products of successful computations.

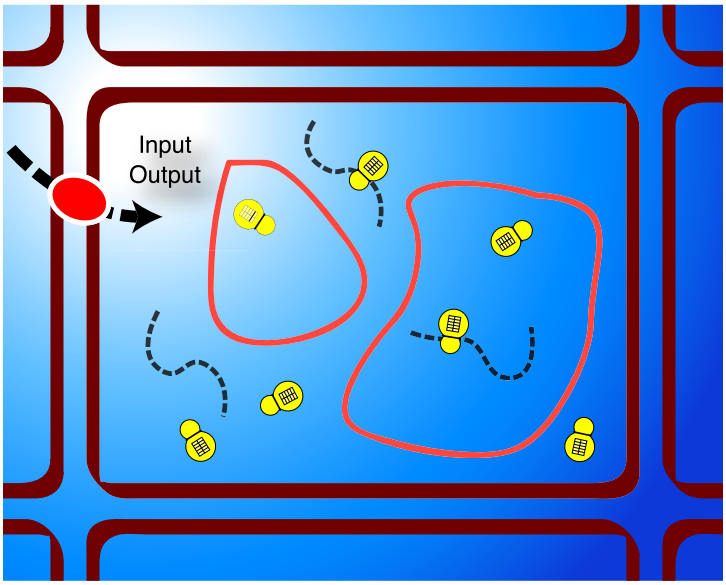
Compartments play a powerful role in the execution model by making it possible to segregate complex computations from one another. This segregation reduces side-effects, which may be brought about through uncontrolled evolution, and it accelerates computation

by increasing the probability of appropriate bindings. Both of these effects have a significant impact on the evolution of programs within the Monod model.



Monod cell is a complete autonomous system based on the Monod computational model. It includes one or more compartment, machine production capabilities (as in a cell nucleus) and input/output capabilities (as in a cell membrane).

A Monod cell has a genome, which contains the instructions to create the machines that run within its compartments and their expression conditions. A genome is a particular instance of a genotype, which is the class of all equivalent genomes. The genotype of a cell can be seen simply as "the program that is run by the cell". The cell itself can be seen as the corresponding phenotype to its genotype.



Does the Turing soup provide any benefit over the classical single Turing machine? It depends, first on what one cares to call a *benefit* and also on the particular implementation. Maybe. Ultimately, a Monod cell is really just a different way to do programming. We hope that the characteristics of the Monod cell lend the model naturally to evolutionary techniques. This alternative view of computation may have radical implications into how one perceives the task of *programming*. As long as the entire apparatus is implemented on a classical, Turing substrate (as it is today), it does not provide any *theoretical* advantage. We still can't solve the Halting problem any better, the notion of *effective computability* is unaffected, etc. However, the Monod computational model lends itself naturally to non-classical computational substrates. We cite three examples: widely distributed processing, which at least may provide a performance difference; truly stochastic matching and release mechanisms, possibly based on quantum effects, so that anything goes as far as effective computability is concerned; and molecular computing, which incorporates the above two examples and perhaps much more.

3 The Cytoplasm and the Monod Cell

The most significant aspect of Monod which has some biological inspiration has already been introduced in the lower layers of the design stack: it is the mechanism of *binding*, which plays the role of molecular recognition in cells.

The next layers introduce many more features of Monod which are biologically inspired, and hopefully computationally relevant. In the first section, we present aspects of cell biology which we want to incorporate in the Monod model. The next two sections introduce the next two layers of the design stack.

3.1 Biological Inspiration

In this section, we present those aspects of the biological cell which we find related to computation. A first implicit principle is indeed that *cells perform computations* through the physical and chemical activity that take place within them. At a grand scale, it is difficult to explain the nature or goal of these computations - life, survival, reproduction, etc. But at a small scale, it is easy to list a number of principles which seem to have a computational relevance. Unfortunately, it is very difficult to distinguish those mechanisms that may have computational relevance from those that don't. This is in great part due to the fact that biological processes are notoriously difficult to modularize into well-defined entities across well-defined layers with well-defined interactions, either in the biochemistry of the cell (with which we're concerned), neurological processes, ecology, etc.

In short, there are no epiphenomenological processes in biology! Everything counts.

Here is a list of principles that seem to have some importance within cells and that we have singled out as being potentially computationally relevant. The way we choose them is simple: we ask "what do we have to do to simulate the microbiology of a cell". Most of these points come from [*Alberts et al. 2002*]. Note that we have an explicit agenda in discussing these topics: we're looking for anything that may have to do with computation. Many points may seem farfetched. We point out, along with each principle, whether we adapt it as a *feature* within the Monod project and if so, in which layer.

A finite number of domains combine to create most proteins

Most proteins can be subdivided into a small number of *domains*. A domain is "a substructure produced by any part of a polypeptide chain that can fold independently into a compact, stable structure" [Ibid, p. 140]. Many domains — called *modules* have been identified which are found across different proteins, and "many large proteins show clear signs of having evolved by the joining of preexisting domains in new combinations, an evolutionary process called *domain shuffling*" [Ibid, p. 146]. The function of certain proteins can be analyzed by identifying the domains that it is made of, and domains are being classified on their own terms ("immunoglobulin module", "growth factor module", etc.).

It is not the case that a protein's function can be decomposed into the sum of the functions of the domains. For instance, "novel binding surfaces have often been created at the juxtaposition of domains" [Ibid, p. 145].

Nevertheless, the main new feature of the Cytoplasm layer is to *restrict* the functionality of the Incubator processing units: they can not be arbitrary programs anymore, but must be built out of a predefined set of domains. Processing units are then called *proteins*.

The question of what kinds of domains exist in the Cytoplasm layer is also driven by biology, through the following observations:

- **Proteins interact through binding and releasing.** The Incubator's purpose is to orchestrate these activities, so there must be domains which express projections — both with other proteins and with ligands. (A note to the reader: in biology, *ligand* refers to anything which binds to a protein — including other proteins. In here, it only refers to ASCII strings.)
- **Proteins have a state.** The state is often referred to as a *conformational state*, and the characteristic of having multiple state is called *allostery*. The state is determined, in part, by the bindings in which the protein participates. In turn, the state affects the affinity of all other bindings. Hence, domains must be able to carry a state and must be able to change the matchers and snippets of other domains.
- FIXME: What else?

Bindings are released stochastically

Proteins and ligands constantly bump into each other. Through collisions and the heat bath, bindings are broken. Bindings have highly variable strengths, depending on the match [Ibid, p. 160]. The strength can of a match between two entities be measured; the measure is called the *equilibrium constant*: it is the quotient, at equilibrium, of the concentrations of the bound entities to the product of their individual concentrations.

It is possible for programs in the Incubator layer to simply match and never release — blocking anything else that's might be useful. In the Cytoplasm layer, we introduce an external release mechanism analogous to the thermodynamic action within the cell.

FIXME: As we discussed in the Turing soup section above, this changes the analysis of *computability* and of the Halting problem (???).

FIXME: I don't understand this well enough, but it seems to be absolutely critical.

Interactions may or may not consume energy

Molecular recognition is *free*, in the sense that it is "powered" simply by the heat bath. Many other reactions also do not require explicit energy input (often in the form of ATP). However, many important reactions do. There is a correlation between reactions that require energy input and *irreversible* reactions. The consumption of energy gives the reaction a preferred *direction*. For instance, contractions of elastic proteins, movement of proteins along DNA, etc. [*Alberts et al.*, p. 183].

Energy consumption is a "fact of life". The laws of physics and the chemical reactions in a biological cell, as efficient as they may be, require the input of energy. Nevertheless, one may well ask what impact, if any, energy requirements have — or have had, in evolutionary terms — on the computationally-relevant aspects of cell biology. An immediate candidate principle is energy minimization: is it the case that evolution will choose less endothermic reactions among multiple otherwise equivalent candidates? This principle certainly seems plausible, and probably has an impact on most of the other facts presented in this section.

In an environment simulated on computer hardware, it might be tempting to establish an analogy between energy consumption and the consumption of CPU cycles. At the very least, this analogy makes the desirable property of energy minimization carry over. Note that we're talking about resource utilization minimization as an element of evolutionary

fitness. For instance, if an otherwise useful reaction which takes the system from state `A` to state `B` is reversible, it may lead the CPU to *thrash* wildly between `A` and `B`. Requiring that the reaction consume energy — and making it irreversible — may be a solution to the problem. If the reverse reaction is also available and also consumes energy, thrashing can still occur but will lead to early cell death, hopefully weeding out or regulating the problem reaction.

Unfortunately, the measurements required to use CPU utilization as a measure of energy are computationally difficult to handle. In the Cytoplasm layer, we introduce a related notion of energy.

FIXME: This is also fairly fuzzy for now...

Cells absorb and eliminate molecules

Cells both *need* and *process* external molecules — taking them in from the outside. They also *discard* and *forward* molecules — eliminating them from their interior. Transport across the cell membrane has both to do with maintaining the well-being of the cell and with it fulfilling its function — and sometimes, the line is fuzzy.

A Monod cell has the same dual relationship to the outside world. On the metabolic side, it can increase the amount of available energy by taking it certain ligands and reduce its energy consumption by eliminating other ligands. On the processing side, in order for a Monod cell to perform a computation, it is fed a ligand, and the output may be identified with an excreted ligand.

We even go a step further by identifying a successful computation with an energy-providing intake. More precisely, at the level of the Monod Culture, we can program a harness to identify the validity of a computation and tie it to the energy level of the cell. This makes the fitness function purely dependent on energy, simplifying computation and description. Input ligands may be seen as antigens entering the cell, and the result of a successful computation is the neutralization and elimination of the antigen, while an unsuccessful computation leads to the gradual poisoning of the cell, which ends up dying.

The cell has a topological structure

The levels of "topological" and "geometrical" structures present in a cell probably constitute one of the most puzzling aspects as far as computation is concerned. There are three levels we are concerned with: the coarse-grained subdivision of the cell into different, more or less impermeable *compartments* like the various organelles; the spatial extent of these compartments, which imply varying concentration gradients of all molecules; and the physical extent and geometrical structure of the molecules themselves, coupled with an exclusion principle which prevents different molecules from occupying the same space at the same time.

The subdivision of the cell into compartments is the most readily interpretable property, from a computational point of view: it substantially increases performance and efficiency, and reduces side effects. Consider two reactions which are desirable:

```
A + X -> B + X'
```

and

```
B + Y -> C + Y'
```

and one that is not:

```
A + Y -> D
```

Segregating `X` to a compartment and `Y` to another, while having a mechanism to shuttle the result `B` from the first compartment to the second has the effect of increasing the rate of both desired reactions (by increasing the relative concentrations of the reactants) while eliminating the occurrence of the undesirable reaction. This phenomenon is apparent in the multiple compartments of the Golgi apparatus, for instance [*Alberts et al.*, p. 736]. Of course, compartmentalization can be taken further by specializing different cell types, different organs, etc.

We introduce a notion of compartments in the Cytoplasm layer, as well as the suddenly necessary notion of compartment transport. Fortunately, there is no computational load implied by this feature.

The impact of the spatial extent of the various compartments is more difficult to analyze. For sure, it allows the existence of concentration gradients across the cell. These are manifest during many different cellular mechanisms, including cell division, development and differentiation, organelle development and movement, etc. However, the impact on the computational work performed by the cell is unknown (by us) at this point. Much research is focused on the computational properties of *reaction-diffusion systems* [*Sienko et al.*, Chapters 3 and 4], but its relevance to cell biology is not emphasized. For instance, what is the importance of the number of geometrical dimensions? Guess: two dimensions would not be enough, while four would be (self-evidently) more than enough.

Nevertheless, despite the uncertainties, we introduce in the Cytoplasm layer a framework with which to impart a topological notion of closeness to compartments. It is a *plugin*-type framework, so that we may choose different geometries for the cell. After all, the goal of Monod is to understand the impact of such biological characteristics on computation. Of course, the computational load imposed by simulating geometry can be very high.

Finally, molecules have a physical extent and structure, and they are subject to an exclusion principle which prevents two molecules from occupying the same space. It is in fact the geometrical structure of proteins (the *tertiary structure*) which gives them recognition capabilities. This aspect is completely abstracted away in the Incubator, where binding is a simple matter of traditional regular expression matching. A further consequence of the exclusion principle is that binding is always one-to-one (per site, that is). The Incubator already takes care of this attribute. Slightly more complex is what may be called *geometrical repression*, where a repressor may prevent further binding by simply *blocking the way*, even while the repressor binds far away from the sites it represses [*Alberts et al.*, p. 406]. There are also much more complex consequences. For instance, *DNA packing* is used as an active, very intricate and poorly understood means of regulating protein synthesis [Ibid, much of Chapter 7]. There are many proteins concerned with keeping tabs on geometrical consequences of extended molecules — for instance, DNA topoisomerases ingeniously prevent DNA tangling during replication [Ibid, p. 251].

At the very least, the regulatory aspects above point to computational consequences of molecular geometry beyond molecular recognition. Unfortunately, the load on any putative simulation is probably prohibitive — and the impact ultimately may be available through other means. At this point we are ignoring in Monod this aspect of cell biology. Biological evolution has to deal with the laws of nature - simulated evolution has to deal with the simulated laws of the Monod Cell, and there is arguably sufficient evidence to show that the

success of genetic algorithms is (at least qualitatively) independent of the particular laws at hand.

The impact of this omission is difficult to gauge a priori. Consider for instance the case of geometrical repression. The functionality may be recreated by making sure that the secondary activator must also bind to the repressor site in order to be active. However, there is a significant difference: in a biological cell, the geometrical repressor operates completely independently of the activator, may be reused across multiple sites, and, most importantly, may have evolved completely independently. This last point indicates that the Monod Cell is possibly missing an "evolutionary independent variable", so to speak.

All proteins are created according to precise instructions

Protein synthesis, the process of transforming genes into proteins, is comparable to a *compilation* process. At a certain level, that process may appear fairly simple: a sequence of codons is transformed in an unambiguous way into a sequence (the same sequence) of amino acids. Compilation in most computer languages is more complex, from a computational point of view! Of course, this simplistic description ignores the crucial aspect of *protein folding* — see below.

Also of note in this vein is the fact that the machinery which is responsible for protein creation is itself a product of the compilation process.

In the Monod Cell layer, we introduce Monod *genes* and the (very simple) process of conversion of the genes into proteins.

Proteins are created by folding sequences of amino acids in a very complex way

The amino acid sequence that makes up the primary structure of a protein is unambiguously determined by the gene(s) involved. Further, the secondary and tertiary structures — the physical folding of the sequence into a functional protein — is also determined, by the laws of physics and the context (temperature, presence of other catalysts, etc.). However, it is extremely difficult to *predict*, without a precise modeling effort, the resulting structure from the base sequence. In particular, there is no compact algorithm — other than a lookup — available to deduce the function of a protein from its genetic description. In computational parlance, the folding algorithm is *incompressible*: the best way to understand the result of folding is to simply do it! "...all structurally programmable architectures must have a highly compressible description in order to conform to formal rules specified in a simple user manual" [*Sienko et al. 2003*, p. 5]. This situation is in marked contrast with the process of compilation, with which an analogy was made above. The algorithm for compilation from source code to executable code is contained in a compiler, which is a very compact program. The quantum mechanics involved in folding are emphatically *not* a program.

How significant is it, computationally speaking, that the function of a "biological program" can not (easily) be deduced by examining the program alone? There are claims that the fact is central to molecular computation [Idem]. However, the reasons given are vague at best. For sure, philosophically, it emphasizes the claim that evolution is blind to the genotype and only acts through the office of the expressed phenotype.

Monod, currently, does not incorporate the equivalent of a folding process as a step from genotype to phenotype. In fact, quite to the contrary, the process used to synthesize

proteins from their Monod-genetic description is very simple. The initial maintainer of Monod has a sense that this is probably a most profound deficiency of the model. Are we missing the boat completely? Is this the single most important computationally-relevant fact of cellular biology? Nevertheless, before attempting to incorporate such a feature, two questions must be at least vaguely answered: how do we construct such a folding analogue? and why?

Most sequences of amino acids do not fold into valid proteins

There are 20^n possible amino acid chains of length n. A typical protein has about 300 amino acids — giving an incredibly large number. "Only a very small fraction of this vast set of conceivable polypeptide chains would adopt a single, stable three-dimensional conformation — by some estimates, less than one in a billion" [Ibid, p. 141]. In order words, using the compilation analogy above, most theoretical source files are not valid programs — which is not a surprise. "And yet virtually all proteins present in cells adopt unique and stable conformations. How is this possible? The answer lies in natural selection" [Idem].

Even further, we can draw a dichotomy between the *syntactic* validity of an amino acid chain, which has to do with it having a single, stable conformation, and the *semantic* validity of the chain, which refers to the resultant protein being able to fulfill its purpose in the context in which it is meant. Natural selection ensures the validity, both syntactic and semantic, of entire genomes.

In Chapter 4 [Monod Cultures], page 59, we subject Monod Cell genomes to such evolutionary algorithms.

Note that the cell possesses fantastically ingenious mechanisms to detect, correct and/or remove incorrectly created proteins (chaperones, the proteasom-ubiquitin pathway, etc) [Ibid, p. 357] [*Nature 426*, p. 883]. Such unstable chains can indeed be extremely dangerous (witness Alzheimer's, Huntington's and prion diseases). However, these mechanisms — or any other, thanks to the Central Dogma — do little good to genotypic evolution. In the Monod Culture, we create analogues of these mechanisms, but we are able to put them to work very early on in the chain. We call these violations of the Central Dogma *cheating*. Cheating offers a glimmer of hope in the face of the fundamental inadequacy of current hardware for running Monod.

The concentration of molecules in a cell is part of the state of the cell

FIXME: Concentration thresholds and triggers. Relative concentrations. Etc.

Certain cellular functions are quickly reprogrammable and possess long-term memory

FIXME: The immune system. Inspiration for certain not-quite-evolutionary mechanisms.

FIXME: The immune system always recognizes non-self by positively identifying non-self antigens. It never recognizes non-self by the *absence* of self-antigens, which would presumably be more powerful. Does this idea fit anywhere in Monod? Is this a limitation of the model, as it is a limitation of our immune system?

3.2 Proteins

All computation in the Monod model is accomplished by *proteins*. They are analogous to proteins in biological cells — hence the name! In turn, each protein is composed of possibly multiple domains, which are somewhat analogous to domains in biology as well.

The domains encode *what a Monod cell can do* at the most fundamental level. For example, there are domains for the following activities (which are described in detail further): binding, logical integration operations, changing ligands. From the point of view of the evolutionary algorithms which are part of Monod, the domains are the *terminal entities*, as we will see when we discuss Monod cultures.

The set of domains in Monod are Turing-complete, in that any Turing program can be created by a protein (FIXME: should be easy to show: have each protein insert a state marker on the tape at the mark position). However, they are not optimized along any other dimension right now, like efficiency, parallelizability, orthogonality. They have also not been *evolved*: they were designed in Monod directly. In order to reach certain levels of optimality, evolving them would probably be necessary. This would consist in *metaevolution*. (FIXME: References?)

We now proceed to describe in precise detail the Monod protein model. We do this in two stages. The first stage is a quick enumeration of all the salient definitions involved and the second stage is an elaboration of each definition along with examples. *The Monod protein model is not simple to explain.* It does not break down into neat stacked layers or independent modules. It is hoped that this situation is more than a failure on the part of the developers, a failure to explain properly. The model itself is complex, and, we presume, necessarily so. (FIXME: Why?)

Here is the quick breakdown of the Monod protein model. (FIXME: quickly review in light of the expanded version now written. Small changes are needed.)

- An *abstract domain*, or simply *domain*, for short, is a fundamental building block of proteins.
- There are different *flavors* of domains. These flavors are predefined in the Monod model. They have different uses and instructions.
- Domains have *interfaces*. There are *expressor* and *acceptor* interfaces, and also different interface *types*. Each interface type has an allowable range of *values*.
- Domains can be *connected* with one another by joining interfaces, one-to-one, an expressor to an acceptor of the same type.
- Certain domains also possess *projection sites*. There are two kinds of projection sites: *ligand binding projections* and *structural binding projections*, as for a processing unit in an Incubator. Those domains that have projection sites are called *boundary domains*.
- Projection sites accept *input operations* and emit *output operations*. The set of operations correspond to the possible actions of processing units in an Incubator. For instance, binding to a ligand, releasing another protein, changing a ligand are operations.
- Domains, each according to its flavor, transform input values on their input interfaces and input operations into output values and output operations through a process known as *conformation change*. Conformation changes are entirely specified by two entities, the *domain functional logic* and the *conformation engine*.

- A *domain assembly* is a set of domains along with valid connections between interfaces. There can be interfaces left *naked*.
- A *state* S of a domain assembly is a set of values for each connection in the assembly. A *state history* associates a state of the protein to a time variable.
- An *abstract protein*, or simply *protein*, is a domain assembly along with a distinguished state, called the *initial state*. By extension from a domain assembly, we can talk about the *state of a protein*, its *boundary domains*, etc.
- An *input history* for a protein is a series of input operations along with times at which they are presented to particular projection sites on the protein. Similarly for output history.
- A *behavior function* for a protein is a mapping from input histories to output histories. It describes how the protein *reacts* to input operations.

  An unqualified behavior function can thus associate *any* behavior with *any* protein. In subsequent definitions, we constrain allowed functions along different axes.
- Two properties can be stated easily in terms of the pure behavior functions.

  A behavior function is *causal* if an input operation can only have an impact on future output operations.

  A behavior function is *spacing independent* if it is causal and if different chunks of the input history can be shifted around, within limits, without substantially affecting the output history other than through shifting. In particular, a spacing independent behavior function is *predictable*, in that the response to an input function does not depend on the time at which the input function is presented.
- Given a spacing independent behavior function, we can extract its *behavior essence function*, which is equal to the output operations it returns in response to a suitably spaced out sequence of input operations. The behavior essence characterizes what a protein does pretty much most of the time.
- We define an equivalence relation among spacing independent behavior functions, declaring that two functions are *essentially equivalent* if they have the same behavior essence function.
- In order to relate behavior functions to the domain structure of the protein, however, we need to look for a refinement of the behavior function.

  A *realization function* for a protein enriches a bevahior function for the protein with another function which, for any input history, gives a complete, time dependent state history of the protein.
- As stated, the two parts of the realization function — the behavior function and the state history mapping — do not have to be formally tied to one another, or even to the decomposition of the protein into domains.

  Hence, we define a *valid* realization function as one that obeys certain rules about how state values are transformed by the conformation functions of the domains in the protein and how input and output operations are reflected in the state history function. These rules are fairly complex and are described in detail below.
- A behavior function for a protein which is equal to the behavior function of a valid realization function for that protein is called a *valid* behavior function for the protein.

Note that there can be many different realization functions for the same behavior function. If suffices that there be a single valid one for the behavior function to be valid itself.

- An abstract protein along with a valid behavior function for it is called a *concrete protein*. A concrete protein is called a *realization* of an abstract protein.

  We can talk of *a realization function* for a concrete protein as a valid realization function which extends the given behavior function.

  Concrete proteins are the focus of this chapter, and of the Cytoplasm layer of the design stack. The definitions, as complex as they may seem, are tailored to be flexible enough to allow for a complex *compilation step* from an abstract protein to a concrete protein while being rigid enough to allow the study of a protein from its purely abstract description.

  The remaining definitions introduce desirable properties of concrete proteins.

- A concrete protein is *well-grounded* if the state history corresponding to the empty input history is eventually constant. A concrete protein is called *stable* if, given any finite input history, the protein's state eventually becomes constant. A stable protein is well-grounded.
- An abstract protein is *stable* if all of its realizations are stable.
- An abstract protein is *well-defined* if all its realizations are behaviorally essentially equivalent.
- A protein is *well-behaved* if it is stable and well-defined. There are well-behaved proteins.
- FIXME: Introduce functional equivalence *across* proteins.

The definitions towards the end of the list above are important because they point to properties that we tend to expect from traditional programs. For instance, an unstable protein is such that it can be in a state where it is always working, but will never produce a result. A protein which is not well-defined will sometimes behave in one way, then in another, unpredictably. We can not completely restrict our attention to *well-behaved* proteins, because Monod cultures will certainly create way unbehaved proteins through random mutations and recombinations. However, well-behaved proteins provide the archetype of desired behavior.

Proteins which are not well-behaved may be seen as analogous to those polypeptide chains which do not have a unique stable conformation, or which are misfolded. Such chains are often biologically extremely deleterious. For example, many such chains expose hydrophobic areas and they tend to form clumps. Fortunately, the cellular machinery includes extensive protection to detect and neutralize these proteins (the proteasomes and ubiquitin binders).

Note that in biology, such invalid polypeptide chains are never called *proteins*. By definition, in biology a protein is well-defined, otherwise it's not called a protein. In the Monod world, a protein does not have to be well-defined, though sometimes we abuse language and assume that this is the case.

Note also that the functional representation of proteins we are currently describing is completely orthogonal to their *genetic* representation. It is not the case that the functional representation is easily derivable from the genetic representation, and vice-versa. This

situation is analogous to that in biology, where the tertiary structure of a protein, which is close to its functional representation, is difficult to relate algorithmically to its primary structure. We will explore this later when we discuss Monod Cells, in this chapter.

In the rest of this section, we present the details of the construction and properties of proteins, proceeding in the order that we laid out in the bullet overview above. We also lay out some examples of all the concepts introduced.

3.2.1 Domains

In the Incubator layer of the design stack, processing units can be arbitrary programs. In the next layer, the Cytoplasm layer, processing units are called 'proteins' and they are not arbitrary programs. Rather, they are created out of smaller building blocks, called *abstract domains*, or just *domains*, for short. These building blocks are connected to one another in precise ways to form proteins.

Domains come in different *flavors*. Each different flavor of domain has a specific use. No single flavor of domain does much on its own. Their power comes through the combinations that they can appear in. Here is a list of the different domain flavors, along with a word about their use.

- **Simple Ligand Binding Domain** (SLBD) — this domain creates a ligand binding Incubator projection.
- **Ligand Binding Domain with Remapping** (LBDR) — this is a more complex ligand binding domain, which exports two projections and allows the movement of chunks of ligand from one to the other. This is the domain to use to alter ligands.
- **Logical Integration Domain** (LID) — this domain performs various logical operations on its input.
- **Boolean Multiplexor Domain** (BMD) — a simple domain which is used for plumbing.
- FIXME: Structural matching domains; slider domains; synthesis domains; explicit state keeper domain; arithmetic operation domains; etc.

(FIXME: also need pseudo-domains for static string and static matchers. Call them, by abuse, domains as well. Need to code before documentation.) (FIXME: Need the structural matching domains too.) Note that, at the moment, the domain flavor definitions are still subject to change. Defining the domains adequately is a matter of delicate balance between flexibility and rigidity: we need enough flexibility to create arbitrarily complex proteins and sets of proteins, but enough rigidity for an appreciable fraction of all possible proteins to be somehow "useful".

Domains may be connected to one another to form more useful entities. They are connected through *interfaces*. Each domain may have many interfaces. Interfaces are characterized by three properties: a *direction*, a *type* and a *name*. The interface's direction is either *acceptor* or *expressor*. The direction of an interface usually refers to the flow of information along the interface, but it does not need to. As we will see later, interfaces must be joined in pairs, an acceptor to an expressor. This is the main property of the interface direction.

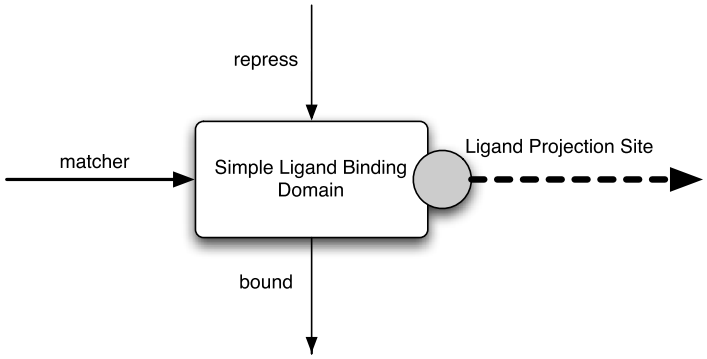
The type of an interface determines what kind of values may be transmitted along the interface. For instance, there are *boolean interfaces* and *string interfaces*. Interface joining is typechecked so that only interfaces of the same type may be joined.

The name of an interface identifies what the interface is used for. For example, an interface may have two boolean acceptor interfaces, one for triggering the function of the domain and one for repressing it entirely. Conceivably, these interfaces would have names like "trigger" and "repress". Certain interfaces may also be arrays of variable length. In this case, the name of the interface followed by an integer index refers to an element of that array.

Consider the Simple Ligand Binding Domain. It has three interfaces:

- A boolean acceptor interface named ***repress***
- A boolean expressor interface named ***bound***
- A matcher acceptor interface named ***matcher***

In addition to interfaces, a domain may have *projection sites* Ultimately, domains are the constituent parts of proteins, and proteins live in the Cytoplasm, which itself is an extension of the Incubator. In this view, the projection sites define the projections that the Incubator uses to look for bindings between the various proteins and the proteins and ligands. There are two types of projection sites, *ligand projection sites* and *structural projection sites*, as for processing units in the Incubator layer. Not all domain flavors possess projection sites. Those that do are called *boundary domains*. For instance, the Simple Ligand Binding Domain is a boundary domain, as it possesses a ligand projection site. This domain is shown below.



Projection sites interact with the Cytoplasm through well-defined operations, called *operations*. There are *input operations* and *output operations*. Both ligand projection sites and structural projection sites have their own private sets of both input and output operations. They correspond exactly to those operations that can be performed by the underlying Incubator. We now list all the possible operations.

A ligand projection site must accept the following input operations:

- **BindLigandT** — triggered when a matching is committed to a binding by the Cytoplasm.
- **ReleaseLigandInT** — triggered when a binding is released through the agency of the Cytoplasm.

It can also emit the following output operations:

- **ReleaseLigandOutT** — should be emitted when the ligand binding can be broken.

- **ActivateLProjT** — reactivate the projection if it is not active. That is, make sure it participates in match searching.
- **DeactivateLProjT** — deactivate the projection, and release any bound ligand if any.
- **ChangeLigandT** — change the underlying ligand to a new string. Used only by the LBDR.
- FIXME: Remapping operation (or splicing and joining), which is not yet implemented by the Incubator...

A structural projection site must accept the following input operations:

- **BindProteinT** — triggered when a protein-protein matching is committed to a binding by the Cytoplasm.
- **ReleaseProteinInT** — triggered when a protein-protein binding is released through the agency of either the Cytoplasm or of the other protein.

It may also emit the following output operations:

- **ReleaseProteinOutT** — should be emitted when the binding can be broken.
- **ActivateSProjT** — reactivate the projection if it is not active. That is, make sure it participates in match searching.
- **DeactivateSProjT** — deactivate the projection, and release any bound protein if any.

A domain D has a well-defined set of input operations O_i, which consists in pairs (b, p) where b is a projection site and p is an input operation of type appropriate to the site. Similarly for the set of output operations O_o.

What do all these interfaces and projection sites and operations actually *do*? They participate in the dynamic nature of domains. Domains transform input values and input operations into output values and output operations through a process we call *conformation change* in analogy with the physico-chemical transformations of proteins. Conformation changes are a gradual adaptation of the values on the output interface and the output operations issued by the projection sites in response to the input values and input operations. The graduality is important, as domains do not react instantaneously. In fact, in order to most flexibly define the domains, the reaction times are left unspecified.

We formalize these issues presently. We divide the description of a domain response in two separate entities: the *functional logic* of the domain, which prescribes the domain dynamics from a purely functional point of view; and the *conformation engine*, which supplements the functional logic with time dependent evolution functions.

The domain functional logic specification is a quadruplet (D, M, I, O) where

- D is a domain, specified with its projection sites and interfaces.
- M is a (usually small) state machine with a set S of states and a set T of transitions between the states.
- I and O are the *domain interface IO specification* and *domain operation IO specification* respectively, which relates the domain definition to the state machine in a way we describe below.

The *domain interface IO specification* associates to each state $s \in S$ in the state machine a function h_s^i from the space of possible input values to pairs consisting of an output value for each output interface and one of:

- nothing;
- a transition in the state machine M; or
- a transition in the state machine *and* a set of output operations

In other words, the domain interface IO specification completely describes the response to changes in the values presented on the input interfaces. This response can be manifested as changes in output values, changes in internal state and emission of output operations.

The *domain operation IO specification* associates to each state $s \in S$ in the state machine a mapping h_s^o from the domain input operations O_i to the machine transitions T. In other words, the domain operation IO specification describes how the domain responds to input operations. Note that the response is always in terms of state machine changes. This restriction simplifies the model. It requires that a domain with a projection site that acts in a non-trivial way have more than a single state.

This completes the description of the functional logic of a domain. We give a number of examples below — it's really much simpler than it looks!

The second part needed to fully describe domain conformation changes is the conformation engine. It adds to the functional logic of a domain a time dependent evolution function. FIXME: What's the best way to describe the conformation engine? Define in terms of a pair of a partial behavior function and a partial state history, so we can connect with those definitions for a domain assembly later on. In the end, it's not important, and it's not used. What's important is to show that the changes in output values and the emission of output operations *takes some time*. This time is bounded by an amount which may be specified. Otherwise, it is left flexible, up to the implementation of the domains proper.

Abstract domains are inherently attached with their functional logic but they do not have, by definition, a conformation engine. An abstract domain with a compatible conformation engine is a *concrete domain*.

Both the functional logic and the conformation engine are needed to fully describe how proteins made up of many domains will behave. As we will see below, the functional logic alone will leave the behavior of a domain assembly ambiguous and adding gluing the conformation engines together will resolve the ambiguities. However, introducing an equivalence relation on the possible behaviors will eliminate the need to know the details of the conformation engine. This simplification will prove very useful.

The framework for specifying domain conformation changes has been given. After a few short notes we present the functional logics for all the Monod domains in the subsections below.

The behavior of domains with respect to certain input values can have certain stereotypical values. Certain input interfaces are *optional*. This feature comes into play when the acceptor interface is not connected to an expressor interface — when it is *naked*, as we introduced earlier. In this case, a *default value* is clamped to the interface.

It is sometimes the case that the interface can not be left naked. This restriction must be specified as part of the domain specification. In this case, a default value does not need to be specified.

Finally, it is interesting to contrast the conformation changes in the simulated Monod world with those in the biological world. Biological proteins manifest all sorts of behavior, including allostery, which is the ability to exist in different states and even integrate different

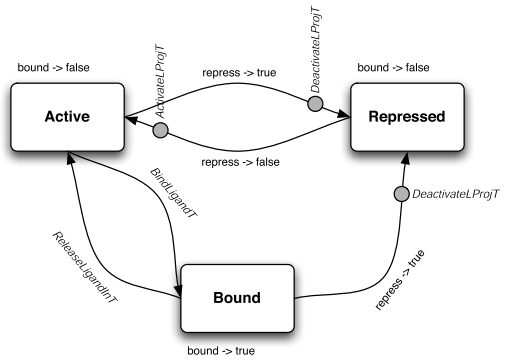
sources of logical input to compute a compound logical state; the ability to cleave ligands and join different strands; the ability to slide along a ligand. The Monod simulations also offer these abilities. A contrast between the real biological world and the simulated model is that physico-chemical conformation changes certainly qualify as complex quantum phenomena, while the Monod model has no pretense at incorporating such aspects. While this is certainly a notable difference, its computational significance is less clear. To the best of the author's knowledge, there are no clear examples of essential quantum protein function. Of course, our knowledge of protein interactions is still fairly basic so the situation is liable to change.

3.2.1.1 Simple Ligand Binding Domain

The SLBD is used to provide a simple indication that a ligand is bound. Binding can be repressed. No alteration can be made to the ligand. Hence, this domain is mostly useful as a regulatory domain.

The SLBD interfaces and projection sites have already been described earlier and an image presented. The `matcher` acceptor interface is mandatory. The functional logic of the domain is independent of the value of this interface. Hence, there is no need to refer to it further in the discussion below. The `repress` acceptor interface is optional and its default value is `false`. Hence, for all practical purposes, the input space is a simple boolean, corresponding to the `repress` interface, and the output space is also a simple boolean, corresponding to the `bound` interface.

The functional logic is described, if cryptically, in the following diagram.



This diagram shows that the state machine M has three states, named `Active`, `Bound` and `Repressed`. The transitions between the states are also shows.

Consider first the `Active` state. In this state, the ligand projection site is active and participates in matching at the Incubator level. The h_i function of the domain interface IO specification is such that the (only) output value `bound` is false. To $repress = false$, the function does not associate a state transition; to $repress = true$, the function associates a transition to the `Repressed` state and the emission of a **DeactivateLProjT** output operation. The h_o function only reacts to the **BindLigandT** input operation, triggering a transition to the `Bound` state.

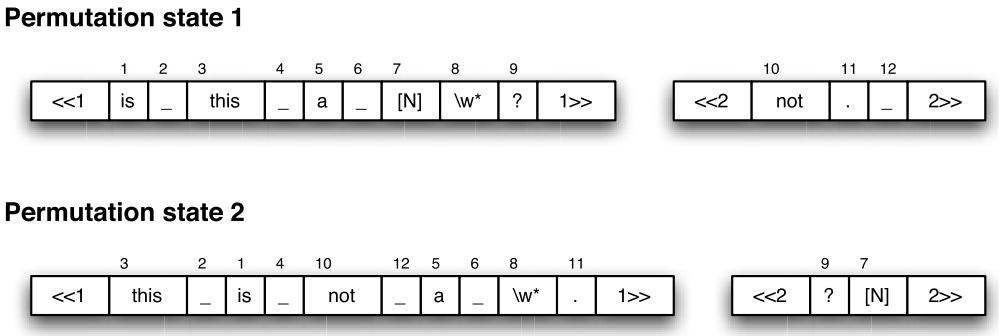
In both the *Repressed* and *Bound* states, the projection site does not participate in ligand matching in the Incubator. The domain IO specification is easily deducible from the figure.

3.2.1.2 Ligand Binding Domain with Remapping

The LBDR is used to bind to ligands and change them. Hence, this domain goes further than the SLBD. Ligand alteration is not arbitrary, however. Instead, two ligands must be bound and fragments may be exchanged between them. Ligand alteraction is thus *conservative*, in the sense that no ligand chunks may be created or destroyed by the LBDR. Other domains handle this task.

The exchanges are performed through permutations, as follows. First, the LBDR is configured with an array of matchers $[m_1, m_2, ..., m_k]$ and with two *permutation states* $S_1 = (P_1^1, P_2^1)$ and $S_2 = (P_1^2, P_2^2)$, each of which identifies an ordered partition of the matchers into two lists. Each P_i^j is a list of indices from the set of matchers $[0, 1, ..., k]$ and defines a *compound matcher*, which is a matcher which will match the concatenation of the corresponding matchers. The permutation states must be exhaustive, that is, for S_1, $P_1^1 \cup P_2^1 = [0, 1, ..., k]$ and similarly for S_2. Finally, the behavior of the end-points of the permutation states must be indicated. There are two possibilities: either the ligands are cleaved the tail end of the second one is affixed to the start of the first and vice-versa, or only the matching regions are rearranged.

An example of the permutation states is shown in the figure below.



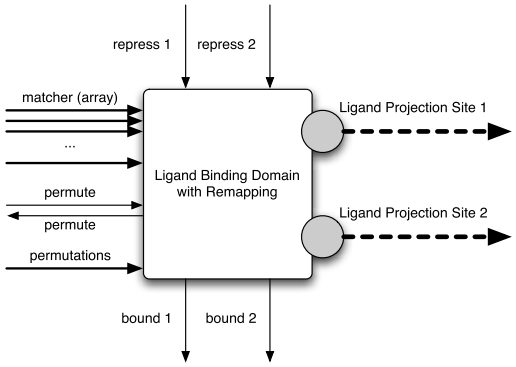
In this example, there are twelve matchers. The first permutation state lists them in the same order they are defined, though this does not need to be the case. The second permutation state lists the same matchers in a different permutation. The *<<1*-type tags indicate that there is no rearrangement of the extended ligands: only the matching regions are changed. This domain takes a ligand of the form "is this a [N]bird?" and transforms it into "this is not a bird". The "[N]" appears as a form of tag. The second projection site serves as a simple holder.

The LBDR clearly has two ligand projection sites. It is defined by the following interfaces:

- A matcher acceptor interface array called *matcher*.
- A mandatory final string acceptor called *permutations* which is interpreted in a specific manner.

- A pair of boolean acceptor interfaces called ***repress1*** and ***repress2***.
- A pair of boolean expressor interfaces called ***bound1*** and ***bound2***.
- A boolean acceptor and a boolean expressor each called ***permute***.

The domain definition is pictured in the following figure.

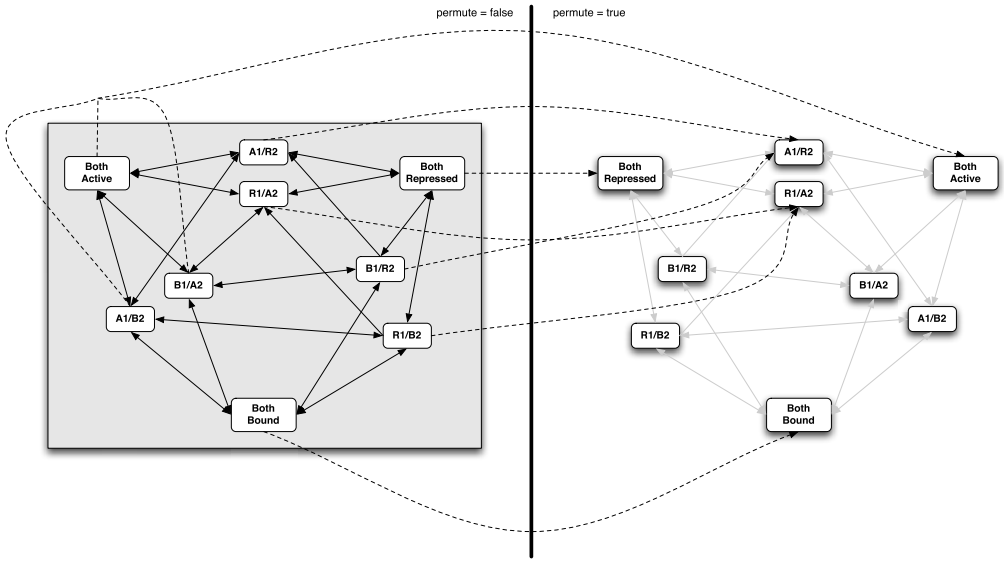


The matchers in the ***matcher*** array are indexed $0, 1, ..., k$ as described earlier. The ***permutations*** string acceptor must follow a simple pre-defined format. If it does not, the domain is invalid and will be rejected. The format simply lists the matcher indices, separated by colons, with both compound matchers separated by "`|`" and both permutation states separated by "`||`". For instance, the example shown in the figure above would read as follows (the string is broken with a `\` to make it fit):

```
<<1:1:2:3:4:5:6:7:8:9:1>>|<<2:10:11:12:2>>||<<1:3:2:1 \
:4:10:12:5:6:8:11:1:1>>|<<2:9:7:2>>
```

The ***repress1***, ***repress2***, ***bound1*** and ***bound2*** interfaces play the same role as the corresponding interfaces in the SLBD, but for the two projection sites of the LBDR. The ***permute*** acceptor interface flips the projection sites from the first permutation state (when ***permute***

= false) to the second permutation state (when *permute = true*). The default value of *permute* is *false*. The functional logic of the LBDR is shown in the figure below.



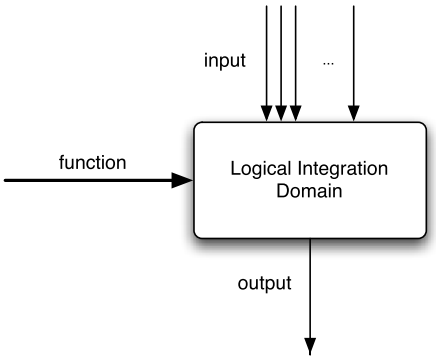
The *permute* expressor is set to the value of the *permute* acceptor after the flip is executed. This feature is needed in order to create a well-defined protein behavior, as we will show when we assemble multiple domains into domain assemblies. FIXME: Explain the functional logic further. Note how when a single one of the two projection sites is bound and the permutation state is flipped, the ligand is released. EXCEPT when the other matcher is the NULL string, in which case the permutation makes sense, and we do it. This makes for a somewhat complex logic... But it's useful, and it works.

3.2.1.3 Logical Integration Domain

The LID is used to compute a logical function of its boolean inputs and presents the result on its sole boolean output. It has the following interface:

- A mandatory final string acceptor called *function* whose value can not be changed over time and which is interpreted in a specific manner.
- An array of boolean acceptors called *input*.
- A boolean expressor called *output*.

The interfaces are shown in the following figure.

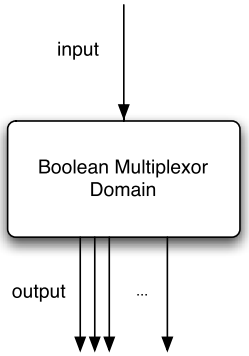


The *function* must be a string which defines a valid boolean function on the inputs *i[0]*, *i[1]*, ..., *i[k]*, where *k* is the index of the last matcher in the *input* array. The function definition can use the *&*, | and *!* operators. The domain does not have any projection site and hence no input or output transition. The domain state machine has a single state. The *output* value is equal to the result of applying the function to the input.

FIXME: Example

3.2.1.4 Boolean Multiplexor Domain

The BMD has a single state, a single boolean acceptor interface and a single boolean expressor interface array. It outputs the value of the input on all the members of the output array. This domain is useful because interfaces can only be connected one-to-one. FIXME: Expand.



3.2.2 Protein Construction and Properties

In this section, we describe how to assemble multiple domains to create useful entities. The end product are proteins. They will be placed in the Cytoplasm layer of the design stack and will act as the processing units of the underlying Incubator.

The first step is the definition of an *abstract domain assembly*, or *domain assembly* for short. A domain assembly A is a pair $(D = domains(A), C = conn(A))$ consisting of a set

D of abstract domains and of a set C of *connections* between domains in D. Precisely, a connection $c \in C$ is a pair (s, d) where s is an expressor interface of some domain $src(c) \in C$ and d is an acceptor interface of some domain $dest(c) \in C$ (possibly the same domain), which are such that both interfaces have the same type. We further require that any interface can not appear in two different connections. Not all interfaces need to be source or destination interfaces. Those that are not are said to be *naked*.

A *concrete domain assembly* is a domain assembly where the domains are conrete domains rather than abstract domains. If $\mathbf{A}$ is a concrete domain assembly and A is the underlying abstract domain assembly, we say that $\mathbf{A}$ is a *concretization* of A.

We define $proj(A)$ to be the set of projections of all boundary domains of $domains(A)$. If $p \in proj(A)$, then $domain(p)$ is the domain which contains p. We trivially extend the notion of operation from domains to domain assemblies by definition an operation of a domain assembly (either input or output) a pair consisting of a projection site $p \in proj(A)$ and an operation (either input or output) of p.

FIXME: Insert graphic example of domain assembly. List projection sites, identify naked interfaces, etc.

3.2.2.1 Behavior Functions

We now present a number of definitions concerning the behavior of domain assemblies. We start from a purely behavioral point of view, presenting these definitions in terms of the projection site operations that domain assemblies accept and emit.

An *input history* for A is a set of pairs (t, S_t) consisting of a time t and a set of input transitions S_t for A. Similarly for *output history*. Note that these notions clearly depend only of $domains(A)$ — in fact, only on $proj(A)$ — as does the following. A *behavior function* for A is simply a mapping from input histories to output histories. A behavior function describes how a domain assembly *reacts* to inputs. However the definition we have given is much too permissive. For example, it allows a domain assembly to never react or to react to nothing at all! In particular, as we have noted, the notion of behavior function is currently entirely independent from the connections of A. We make a few additional definitions in order to restrict behavior functions to make them slightly better behaved and give a few examples.

A behavior function F is *finite* if it maps finite input transitions to finite output transitions. Finiteness is a useful condition for proofs.

A finite behavior function F is *causal* if input transitions can only have an impact on future output transitions. More precisely, F is causal if, for any state input histories I_1 and I_2 which are equal up to a certain time t_e, the corresponding output histories O_1 and O_2 are also equal up to t_e. Causality is a desirable property. Well, put another way, the lack of causality makes everything rather unpredictable.

We can *shift* an operation history h by a time parameter τ to create another operation history h_τ by adding τ to the time member of each element of h. We can do this for any subset of an operation history as well. The *extent* of a finite operation history is the time interval from the lowest time appearing in the definition of the history to the highest. We can also make this definition for any subset of an operation history. A *cut* in an operation history is a time value θ which is not equal to the time of any operation in the history. Given an operation history h, a cut θ and a shift value τ, we can shift the parts of h which

are later than θ by τ to create a new operation history $h[\theta, \tau]$. This operation is called *spacing* an operation history.

A causal behavior function F is said to be *spacing independent* if any input history, suitably spaced, is then essentially invariant for F under the action of further spacing. More precisely, F is spacing independent if, for any input history h and for any cut θ of h, there exists a shifting value $\tau_0(\theta)$ such that, for any shifting value $\tau > \tau_0(\theta)$,

$$F(h[\theta, \tau]) = F(h)[\theta, \tau].$$

This definition is easily extended to a finite arbitrary number of cuts. The gist of the definition is that a spacing independent function has a well-defined, canonical *response* to any input history. Because of causality and finiteness, we know that if we wait long enough after the last operation of an input history, we'll eventually see the entire response. Spacing independence adds the desirable property that the response does not depend on the timing of the last input operation, if it occurred late enough to begin with. Note that in particular, if we choose the cut to be before any input transition, we can shift a spacing independent behavior function by any value and obtain the same response. This comforting property is called *predictability*. Finally, spacing independence has an impact when passing to a discrete time value, as we will see further.

Given a spacing independent behavior function F we can create the associated *behavior essence function* $\overline{F}$. This function accepts a finite ordered sequence of input transitions and returns a finite ordered sequence of sets of output transitions. The output is equal to the result of applying F to the input history corresponding to the input sequence, suitably spaced out so that the result will be independent of further spacing according to the definition above. It is easy to see that $\overline{F}$ applied to an initial sequence is equal to the initial sequence of the result of applying $\overline{F}$.

The behavior essence function characterizes what the behavior function does *most of the time*. What it does not include is what happens when input transitions are presented *too fast* that they can step on each other's functions. As an example, imagine that a binding operation is presented to a projection site of a domain assembly and that a release operation follows quickly, before the binding operation has had the time to "complete". The resulting behavior might be very complex, and it is even difficult to say even what it *should* be. It is certainly captured by the behavior function F, but not by the behavior essence function $\overline{F}$.

Two behavior functions F and F' which have the same behavior essence function $\overline{F}$ are said to be *essentially equivalent*. Hence, essentially equivalent functions differ only in they respond to closely presented input transitions. Of course, if they differ at some time t_0 then they may differ at any subsequent time even if further transitions are presented leisurely.

FIXME: Examples?

3.2.2.2 Realization Functions

FIXME: There is some minor confusion sometimes about concretizations versus realization functions. Clean up.

All of the definitions above concerning behavior functions are very much removed from the domain assemblies they concern since, as we mentioned, they depend only on the projection sites of the assemblies and not on the connections. In order to tie behavior functions

deeper with the assemblies, we need to introduce some more machinery. This machinery is related to the connections of the domain assembly.

A *state* of a domain assembly is an assignment $c \mapsto v$ of a value v for each connection $c \in conn(A)$ which is such that v belongs to the value set of the well-defined interface type of c. A *state history* for A is a time dependent assignment of a state, $t \mapsto c(t)$.

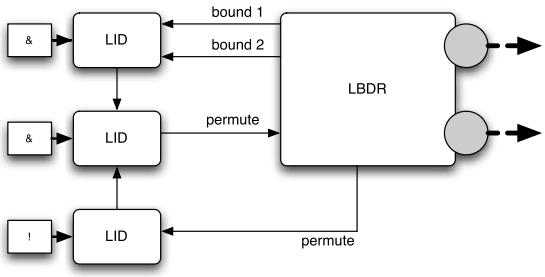
The *restriction* of an operation history h to a domain $d \in domains(A)$ is equal to those elements of the history whose operations belong to one of the projections sites of d. Similarly, the restriction of a state history of A to a domain d of A is the function which assigns values to each acceptor and expressor interface of d such that the value assigned at time t to an interface which is either the source or the destination of a connection in $conn(A)$ is equal to the value of that connection at the same time.

A *proto realization function* of a concrete domain assembly **A** is a pair (F, S) where F is a behavior function for A and S is a mapping from input histories to state histories of A. A *realization function* for **A** is a proto realization function which is such that its restriction to each concrete domain $d \in domains(A)$ is compatible with the conformation engine of d. FIXME: more details — we need to have defined the conformation engine more appropriately earlier.

FIXME: Any concrete domain assembly can be given a realization function: realization functions exist. In how many ways? What are the parameters?

The realization function is the glue that ties together all the components of the domain assembly. The behavior function associated with a realization function is definitely not arbitrary and must bear some relationship to the projection sites of the assembly, the domain types which exist in it and the connections between the domains. A behavior function of a domain assembly A which is associated with the realization function of a concretization **A** of A is said to be a *true* behavior function of A. Let's look at some properties and examples.

Note that a true behavior function is not necessarily finite. A non-finite behavior function will occur if we can find a cycle in the union of the state machines which does not need any external input operations to work. The following figure shows a domain assembly with a single LBDR domain which has such a cycle. (Note that only the relevant interfaces are shown.)

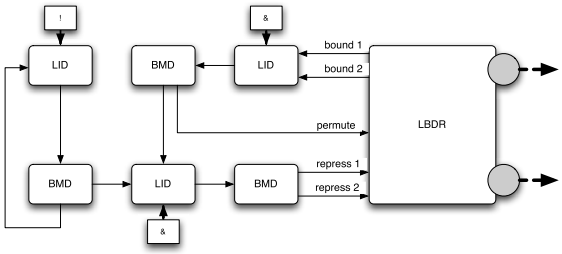


Whatever the realization function used to concretize the above assembly, the behavior function will go back and forth between the two permutation states as soon as both ligands are bound. Each permutation emits an output transition. Hence, the output history corresponding to the input history with two binding operations is infinite.

However, as soon as it is finite, a true behavior function is necessarily causal. Indeed, output transitions are always emitted by domains in response to input transitions and acceptor interface value changes.

FIXME: Does finiteness depend on the realization function? In the above example, it does not, but that does not need to always be the case.

Spacing independence is a bit more complex, as a true finite behavior function can be spacing dependent. As for lack of finiteness, spacing dependence is again caused by cycles in the global state machine, but this time the cycles are not manifested to the outside world: they are purely internal cycles. Consider the assembly shown in the following figure.



There are fundamentally different behavior functions associated with concretizations of this domain assembly representation. However, at least some of them have a conspicuously bad property: when both ligands are bound, whether they are immediately released or first undergo a remapping is unpredictable. Indeed, the two domains on the right-hand side of the figure, the negation logical integration domain and the boolean multiplexor, combine to produce an ever-oscillating true/false value which is fed to the rest of the system. If the values output from the top boolean multiplexor domain are slow enough, release will take place before the permutation occurs. Note that when both ligands are not bound, repression orders do not reach the LBDR so that the output history is indeed finite.

The choice of behavior is thus between a domain assembly which does nothing (it binds and releases) and one that does a ligand remapping. It may seem this choice is somewhat illusory, as we can simply ignore the domain assembly when it does nothing. However, the example illustrates an important property of domain assemblies and as we add more complex domains to the mix, a domain assembly can act in a much more confused fashion.

In order to address this eventuality, we introduce a property of realization functions. A state history is said to *stabilize* if all the values on all the connections are constant after a certain time. A realization function is *stabilizing* if the state history corresponding to any input history stabilizes. All domains individually stabilize. Hence, the only reason that a domain assembly would fail to be stabilizing is if there's a cycle in the assembly which creates an oscillatory behavior, as in the above example.

It is easy to see that a stabilizing realization function necessarily has a spacing independent behavior function. Indeed, choose the shifting parameter τ_0 to be equal to the time it takes for the state history to stabilize. FIXME: Is the converse true? It may be, but that's not totally obvious.

Some of the definitions above have been made in terms of concretizations of domain assemblies. This situation is unsatisfying as we would like to be able to look at a graphical

representation of a domain assembly and deduce its properties without having to consider particular realization functions. Hence, we make the following definitions. An abstract domain assembly A is finite if all its concretizations are finite. (FIXME: as noted above, is it sufficient to look at a single concretization?) It is stabilizing if all its concretizations are stabilizing and similarly for spacing independence.

One last property remains. The reader will have noticed that the behavior of a domain assembly can depend strongly on its concretization. Hence, a *well-defined* domain assembly is a finite spacing-independent abstract domain assembly such that any concretization of it has the same behavior essence function. It is still the case that the boundary behavior of a well-defined domain assembly can be unpredictable. It is difficult to control this boundary behavior.

3.2.2.3 Proteins, finally

A *protein* is a triplet $P = (A, S_0, F)$ where A is an abstract domain assembly, S_0 is a state for the domain assembly called the *initial state* and F is a true behavior function for A which is such that its associated realization function always has an initial state equal to S_0. Any abstract domain assembly can be made into a protein in potentially different ways.

FIXME: Also need to specify initial state for the state machine of each domain. Carry through rest.

A protein can be seen as a network of pipes and machines which carry and transform various types of liquid.

Proteins may have many properties, derived from properties of domain assemblies. There are finite proteins, spacing independent proteins, well-defined proteins. We don't recall all the details here. The reader is invited to read Appendix A [Combinations], page 77. This appendix contains an exploration of various proteins and describes their properties. Some of the properties discussed have not been introduced yet.

Later on we will see that the Monod Culture evolutionary framework can not avoid non-finite proteins, non-well-defined proteins, etc. Nevertheless, these properties are very much desirable. Indeed, any task that any protein can do can be done by one or a set of well-defined proteins, and probably better. (FIXME: Prove this!)

FIXME: Deleterious behavior. Non-productive behavior. Good behavior.

FIXME: Proteins inside an Incubator. Protein programs containing more than one protein.

3.3 The Cytoplasm

FIXME: Harness controlled stochastic release. In a biological cell, release is essentially controlled by the heat bath, and is thus a truly quantum phenomenon. Are we capturing this adequately here?

FIXME: Is the binding strength the same as the affinity which is used to calculate the matching?

FIXME: The rest of the chapter is pretty old...

3.3.1 Compartments

3.3.2 Release

3.3.3 Energy

Certain reactions in the Monod Cell consume or produce energy. Unlike in a biological cell, where energy is carried by dedicated molecules, the energy in the Monod Cell is maintained by as a simple tally for each component. When a reaction consumes energy, the tally is lowered, while when it is produced, the tally is increased.

To produce energy, a cell must successfully metabolize absorbed substrates, as will be shown later. Through the operation of proteins, ligands are gradually broken down. When tagged with a particular indicator by proteins, the sequences are recognized by the containing compartment's basic functions and categorized as either poisonous or energetic. A poisonous base sequence lowers the energy tally proportionally to its length, while an energetic one increases the tally.

Energy-consuming reactions can not take place within a cell with no energy left. Furthermore, such a cell will eventually die. Note that the energy of a cell is the sum of the energy of each compartment within the cell.

Energy within a biological cell serves to power certain reactions. In a Monod Cell, power is not an issue, of course. Also in a biological cell, energy is also used to give *direction* to certain reactions by making them irreversible. While irreversibility may be built into proteins without recourse to energy, a limited supply of energy helps drive the evolution of Monod Cell genomes away from *wasteful* cells which include lots of reversible operations - this will be explored in the next Monod book.

3.4 The Monod Cell

FIXME: A Cell absorbs a Snippet and must identify if the Snippet is a poison or if it is food. Its fitness depends on making the accurate recognition.

FIXME: Multiple Localities: Compartments

FIXME: Nuclear compartment may be implemented differently.

FIXME: Simplify. In brief, the fundamental inspiration for Monod is the consideration of biological proteins as small computational units. In Monod, all computations are done within a simulated *cell* by a large number of small, functionally distinct *proteins* which *react* with each other and with *ligands* which form the input/output to the cell to modify said ligands and thus perform computation. The reactions are simple string manipulation operations. The processing units have basic capabilities such as recognizing, binding and releasing target sites (either on ligands or other processing units), logically integrating their inputs and acting based on the result, splitting and gluing ligands, locally modifying them, etc. They are in fact Turing-complete — redundantly so — but the Turing machine image should be modified by imagining *lots* of them binding and detaching from the tape, instead of just one.

One positive attribute of Monod is that it is computationally realistic to simulate on existing computers, as it uses standard tools and is thus able to capitalize on pre-existing and optimized infrastructure: ligands are strings and binding is done through regular expressions, for instance.

Another nice attribute of Monod is that it is particularly well-suited to computational evolutionary methods. This is because the evolutionary terminals - the *domains* which

are the building blocks of the processing units - can be combined with great freedom, while providing enough flexibility to be Turing-complete.

A natural question to ask is whether the Monod approach is anything new, different from the other approaches mentioned earlier or others. Monod has been thought of and described in a context of biology, which is not quite so common in computer science; oftentimes, language differences can obscure otherwise evident similarities. FIXME: The answer is: I don't know...

4 Monod Cultures

FIXME: Nothing yet in this chapter. It will be the most interesting chapter, however, as it will deal with automatic programming.

FIXME: why do we think we'd be more successful than any other attempt to "evolve" universal computers? (Or do we?) It's a matter of finding the right balance. Also, cheating.

4.1 Evolution Driver

Fitness is a compound of many parts: ability to *digest* snippets, speed, parallelizability, etc.

Co-evolution and meta-evolution

Steady-state population model. Evolutionary Strategy (ES). Tournament matching. Brood crossover.

Homology is natural to the Monod genetic model, since programs appear natively divided into proteins. This does not mean that homology is necessary in the Monod model, but that its advantages can be used. This is in stark contrast to traditional approaches to Genetic Programming (GP), where the lack of homology causes, at least, great uneasiness about the crossover operator.

4.2 Cheating

What's cheating?

Genotype filter

Lamarckian incubator — not necessarily all it's cracked up to be. Refer to "Myths and Legends".

5 Results and Future Projects

FIXME: Nothing in here yet! The following are just draft ideas of the stuff we want to try with Monod. Arrange formally and clean up as "Future Projects" soon.

XOR learning — this is the first recognized benchmark learning problem, because it is the simplest non-linearly-separable problem.

Parenthesis matching. Calculator — both verification and computation.

Baldwin effect. Starting from a single complex unit and breaking it up into manageable pieces.

Encoding and measuring entropy of the encoding and minimizing entropy.

Evolutionary results:

Parenthesis matching. Calculator — both verification and computation. Evolution of the parenthesis matching: probably the global verifier is too complex to evolve through a random mutation — and it would absolutely have to, since intermediate results are not useful. However, a gradual world which offers lots of different snippets (of many *lengths*) may allow for a gradual evolution — coevolution. Symmetric and asymmetric coevolution. Examples: refer to "The Symbolic Species".

Metaevolution: evolution of the breeder.

Baldwin effect. Starting from a single complex Complex and breaking it up into manageable pieces.

Encoding and measuring entropy of the encoding and minimizing entropy.

6 Compilation and Usage

This chapter shows how to build, install and run the various executables in the Monod implementation, as well as the different parts of the documentation. We begin by describing in detail how to compile the Monod binaries from the source code, then we present the documentation creation process, and end the chapter with usage notes for the various Monod executables.

The current documentation describes the latest version available at the time of writing, which is version 0.02. Check the file '`VERSION`' in the top-level source directory to compare.

> **State of the Code: Version 0.02**
>
> At the time of writing, the Monod project is still in a early phase. The design stack is far from fully implemented. The bottom 3 layers (up to the Incubator) are "working". Additionally, the protein-definition part of layer 4 (the Cytoplasm) is there. The Swarm implementation is extremely inefficient and non-scalable (don't put more than, say, 10 residents in there!). No one cares about performance at this stage — the goal is to get to the Monod Culture as fast as possible. The the Cell is not yet implemented, not to mention the Culture. The only meaningful executable is the Incubator-based calculator example, which can't even perform division at this point. The test suite should run without a problem.

6.1 Compilation and Installation

Currently, the only way to run the Monod implementation is to compile from the source code, as there is no binary distribution. Hence, this section assumes that the reader has access to the source distribution of Monod, which can be found at http://sourceforge.net/projects/monod/. Monod is tested on Mac OS X and Linux.

We begin by giving details about the various prerequisites needed to build and run Monod. Then we introduce the various Makefiles and the targets used to create the executables.

6.1.1 Prerequisites

In order to build and run Monod, a certain number of programs need to be available on the computer. Namely:

- `make`, to process the Makefiles;
- The OCaml language, in the form of the compiler and various attendant programs;
- Texinfo, to process the manual; and
- Perl, which is used for a number of pre-processing steps.

6.1.1.1 Hardware

Currently, the Monod implementation runs on a single platform. The faster the CPU, the better — it is extremely CPU intensive, in part because of the matching which takes place in the Swarm layer (especially as it is extremely inefficient right now! FIXME: that). There

are plans to extend Monod to run in a distributed/clustered fashion across multiple platforms. And it is conceivable that Monod could accommodate non-standard computational platforms.

However, this is not the case right now.

6.1.1.2 make

The build process is controlled by `make`. GNU `make` version 3.79 was used during development, and any compatible GNU version should work. However, note that '`OCamlMakefile`' (see below) was used as the make template and it *requires* GNU `make`.

6.1.1.3 OCaml

Monod is written in the Objective Caml (OCaml) programming language, "the programming tool of choice for discriminating hackers". The OCaml website can be found at `www.ocaml.org`. OCaml combines the astounding performance characteristic of a low-level language with the "if it compiles, it works" attitude of a high-level language, with a special emphasis on syntactic manipulations of complex data structures, making it a good choice for the Monod project. OCaml is also eminently portable and Monod should run on Mac OS X, Linux or Windows.

You can download the OCaml source code from the web site above and compile it for your platform. The project was developed using version 3.06 of OCaml. The Monod distribution currently does not need any OCaml libraries and tools other than the ones in the standard distribution. (Precisely, we use the `ocamlc` and `ocamldoc` commands.)

We do use '`OCamlMakefile`', a very neat and useful assistant for every OCaml project, to help the build process, but the file is included in the distribution in a slightly modified form. The distribution for '`OCamlMakefile`' can be found at

```
http://www.ai.univie.ac.at/~markus/home/ \
ocaml_sources.html#OCamlMakefile.
```

The file is named '`ocaml.mk`' in the Monod distribution.

6.1.1.4 Texinfo

The book which you are currently reading is packaged as a part of the distribution, in the '`manual/`' directory. In order to create a local version of the documentation, you will need a certain set of build tools. However, the latest documentation can usually be found on the SourceForge site, at `http://monod.sourceforge.net/`.

FIXME: makeinfo, texi2dvi, latex, dvips, etc.

6.1.1.5 Perl

FIXME: Used for some preprocessing (for logging and test suite preparation) and for the log viewer. Using version v5.8.1-RC3.

Need the *`Tk`* package for the log viewer. It can be installed from CPAN. Note that if you use Mac OS X, as does the maintainer, you need to install Perl from scratch (say, in '`/usr/local`') in *dynamic* mode in order to use *`Tk`*. The instructions are included in the Perl source. Don't forget to start X before running a Perl/Tk program. But it works really well.

Also need the following packages: *Tk::Columns*. (Note that we need the version 0.03 of this package, which is not the version that CPAN installs, for some reason. The previous versions just crash, on both Mac OS X and Linux. So just install it manually.)

6.1.2 Compilation

We describe here the simplest way to obtain all the interesting Monod executables. For more information, the reader should consult the detailed Implementation section, Section 7.3 [Source Code Structure], page 70.

The main Makefile, located in the top directory as '`Makefile`', can be used to invoke all the main Monod end products. More precisely,

- `make all` will create all the main Monod executables, namely:
  - '`singlecell/singlecelltest`' — a non-interactive self-contained program which creates a single cell, loads it with a simple genome and goes through a few testing iterations. It is described in more details Section 6.3 [Usage], page 64.
  - FIXME: List here the other executables when we have them.
- `make examples` will create a few interesting stand-alone executables, some of which do not contain the entire Monod model. Namely:
  - '`examples/incubcalc/incubcalc`' — an interactive arithmetic evaluator, based on the Incubator layer of the design stack. It is described in more detail in Section 6.3.4 [Provided Examples], page 67.
  - FIXME: List here the other examples when we have them.
- `make testsuite` will create the full unit test suite, located in '`testing/alltests`'. It is described in more detail in Section 6.3.2 [Test Suite], page 65.
- `make doc` — described under Section 6.2 [Documentation], page 63.
- `make manual` — described under Section 6.2 [Documentation], page 63.

There is currently no installation step — all the binaries are left in the build directory. We describe their location in detail in the appropriate sections below. You can simply execute the executables from the command line. They are described in detail in the Usage section below.

6.2 Documentation

There are two major parts to the Monod documentation: the manual (which you are reading presently), and the automatically-extracted source code documentation. Both are generated using the source code Makefiles.

6.2.1 Manual

The manual source is a single Texinfo file, '`manual/monod.texi`'. It is processed using various documentation tools into either online HTML documentation or a printable Postscript book. Both versions of the documentation are created by running `make manual` from the top-level source code directory. The head HTML page created is '`manual/Monod/index.html`'. All the other pages can be navigated to from this page. The printable Postscript version of the manual can be found as '`manual/monod.ps`', and the printable PDF version as '`manual/monod.pdf`'. Note that PDF support in Texinfo is fairly rudimentary and you may

have to tinker significantly with the TeX setup in order to generate PDF output. I've had the best luck with teTeX version 2.0.2.

The manual contains some images. These images are not saved along with the documentation source, to conserve space. The effect is that it is impossible to build the documentation, either the online or printed form, without a little additional work. The graphics must be downloaded separately: on the SourceForge web site, go to the file release area and download the latest '`graphics.tar.gz`' file in the graphics package. In the '`manual/`' directory, create a symbolic link named '`graphics/`' to that directory. You may have to erase an old '`graphics/`' directory, if needed. Then you'll be able to build the documentation.

The manual contains a certain amount of mathematical formulas. It is fairly difficult to get texinfo to deal adequately with these formulas. The solution we have adopted is to prefer printed ouput to online output as far as the formulas are concerned. In the printed output, the full might of TeX will be used, while in online output, the raw commands are printed. For instance, consider T_i, which will appear neatly in the printed output but as `T_i` in the online output.

6.2.2 Source Code Documentation

The source code documentation is automatically extracted from the source code by the `ocamldoc` program. The output is an online, browsable repository of module signatures, object types, etc. which is mostly useful to implementers. To create the source code documentation, run `make doc` from the top-level directory. The head HTML page can be found at '`doc/html/index.html`'.

6.3 Usage

Here we show how to use the various executables produced from the Monod distribution. We first present the common command-line arguments, and then introduce each executable in turn. Currently, these executables are:

- `testing/alltests`
- `singlecell/singlecelltest`
- `examples/incubcalc/incubcalc`

6.3.1 Command-line Arguments

Currently, all the Monod executables are command-line programs. While each may have specialized command-line arguments, a few arguments are shared, which we now describe.

'`--help`'

Display the entire list of options.

'`--log-level <s>`'

Set the logging output level to `<s>`, which has to be one of *`Emerg`*, *`Alert`*, *`Error`*, *`Warning`*, *`Exceptional`*, *`Config`*, *`User`*, *`Info`*, *`Exec`*, *`Debug`* or *`Data`*. The default is *`Warning`*. For more information about logging, which is most useful to implementers, see Section 7.5 [Logging], page 73.

FIXME: That's it for now!

6.3.2 Test Suite

The first program one should run is the automated test suite, which will verify that the build is valid and that the platform is able to execute Monod. To build the test suite, run `make testsuite` from the main directory. The product is the executable '`testing/alltests`'. In the '`testing/`' directory, run `./alltests --help` to view the command line options. In addition to the options described in Section 6.3.1 [Command-line Arguments], page 64, the following options are available:

'`--verbose`'
: Display the description of each test run.

'`--runwait n`'
: Set the number of seconds to run each long test.

To run all the tests, simply run `./alltests` in the '`testing/`' directory. The output will look like this:

```
mathieu% ./alltests
.................
Delay for the swarm test 1
Second delay for the swarm test 1
Third delay for the swarm test 1
Fourth delay for the swarm test 1
.
First delay for the swarm test 2
Second delay for the swarm test 2
....
First delay for the incubator test 1
Second delay for the incubator test 1
Executed 4665 bindings in one second
.
First delay for the incubator test 2
Second delay for the incubator test 2
Executed 1005 bindings in one second
.
First delay for the incubator test 3
Executed 3663 bindings in one second
.
First delay for the incubator test 4
Executed 114 bindings in one second
.
First delay for the incubator test 5
Executed (299, 299) bindings in 0.1 second
.
Ran: 27 tests in: 4.95 Seconds
OK
mathieu%
```

The important piece of information is the *OK* on the last line, which indicates that all the tests were run successfully. This should be the case for any freshly obtained distribution. Any error in the test run will be very apparent, as in the following output:

```
mathieu% ./alltests
.................F
First delay for the swarm test 2
Second delay for the swarm test 2
....

     [...]

.
First delay for the incubator test 5
Executed (234, 233) bindings in 0.1 second
.
======================================================================
FAIL: Testing Util:17:Simple residents with no exposure

OUnit: Just created swarm should not be running.
not equal
----------------------------------------------------------------------
Ran: 27 tests in: 4.55 Seconds
FAILED: Cases: 27 Tried: 27 Errors: 0 Failures: 1
mathieu%
```

The *F* on the first line is an early indication of the test failure, while the details are given at the end of the entire run. The details are used by a developer to locate the precise test which failed and begin debugging.

Running this executable will create the file '`monod.log`', containing the logging output according to the log level.

Be aware that the test suite may succeed or fail depending on the value of the logging level set through the command-line! Indeed, certain tests check for time-dependent results. Since the log level affects the performance, some results can be skewed. In general, running with *Emerg* will provide the best results.

You can also increase the length of time each long test is supposed to run with the '`--runwait`' option. If you set the log level to log many entries, you can increase the runwait to make the test run longer to make sure it ends up passing. It is also often useful to simply increase this parameter to find possible deadlocks that happen only probabilistically. Note that sometimes deadlocks manifest themselves in ways that the test suite does not recognize, by simply stalling the test. Having a CPU activity monitor running alongside the test is often useful: if the activity drops while the test is still running, there's probably a deadlock somewhere.

Also, the success of the tests may depend on whether the executable was byte-compiled or compiled to native code, and whether the OCaml runtime is using native threads or not. Again, in order to smooth out differences, increase the '`--runwait`' parameter. It seems like the default parameter of one second may not be quite enough in all cases.

6.3.3 The singlecelltest Executable

FIXME: The `singlecelltest` command is currently completely broken. It will simply create lots of logging output to '`monod.log`'.

6.3.4 Provided Examples

Some standalone examples are provided which extend the testing provided by the unit tests in the '`testing/`' directory. The examples provided are

`incubcalc`
: An Incubator-based arithmetic-expression calculator.

To make the example executables, run `make examples` in the main directory.

6.3.4.1 incubcalc

The calculator has already been described in Section 2.3.1 [The Incubator Calculator Example], page 21 earlier. We show here how to use the implementation provided in the source code. The executable `incubcalc` is located in the '`examples/incubcalc/`' directory. The only command line options are given in Section 6.3.1 [Command-line Arguments], page 64.

From the '`examples/incubcalc/`' directory, run `./incubcalc`. The program will come up with a prompt:

```
mathieu% ./incubcalc
Please enter a string to evaluate:
#
```

Type in an arithmetic expression and press return:

```
mathieu% ./incubcalc
Please enter a string to evaluate:
# (2 + 2) * (3 + 2 * 6 + 5) <return>
```

The program will calculate the answer, print it, and loop to the beginning:

```
Done
The result is 80
Please enter a string to evaluate:
#
```

If you enter an invalid expression, it will stall (FIXME: Fix that soon!). Currently, no division is allowed (FIXME: Fix that soon!).

7 Implementation Details

We now present details of the implementation of Monod. This chapter should be used alongside with the source code proper, which can be downloaded from http://sourceforge.net/projects/monod/. The source code can be browsed through the web interface provided by `ocamldoc`, which is similar to that of `javadoc`. See Section 6.2.2 [Source Code Documentation], page 64. The reader should browse this documentation as needed throughout this chapter, as many references to the source code are included. Furthermore, the reader should be familiar with the previous chapter, Chapter 6 [Compilation and Usage], page 61.

Particular emphasis will be paid throughout the project to maintaining the documentation — in Texinfo format — adequately.

Significant attention is given to the large-scale architecture of the project to optimize maintainability, reusability and explanatory load. The upshot of this and of the documentation is that it should be easy to ramp up and become an active participant on the project.

7.1 Contributing to Monod

Contributions to the Monod project are welcome in all possible forms: discussions, critiques, documentation, support, coding, etc. The only rule is that the project should be kept as fun and ambitious as it is now! This section describes how to best contribute to the project.

The first thing to do is to get in touch with the project maintainer. Write to Mathieu Gagné at bourbaki@earthlink.net.

Monod is distributed under the GNU Public License (GPL). See the file '`COPYING`' for more information. In brief, this means that contributions to the Monod project must be released back to the community. Write to the maintainer for more information. The best way to do this is to submit them through the SourceForge site. See http://monod.sourceforge.net/projects/. FIXME: Why open source project?

Monod is still at a very early planning stage, so that there are still *lots* of things to implement! Many of them are described in the present manual — see Section 7.2.2 [Future], page 69; Chapter 5 [Results and Future Projects], page 60; Chapter 6 [Compilation and Usage], page 61 for quick pointers. Also, the '`TODO`' file in the main directory contains a brain dump of many features that need to be done. Finally, all interesting ideas are welcome!

Every project has certain attributes and practices that are particularly encouraged. Monod is no different. Here are some of the qualities that are most emphasized for the development of Monod.

- **Testing** is an important part of the development of Monod. Unit tests are kept in the '`testing/`' directory. It's good practice to add tests as often as possible when new functionality is developed and when bugs are found. The time spent on tests will actually *diminish* total development time in the long run if debugging, refactoring, etc. are taken into account.

  In addition to unit tests, more comprehensive tests (like the `incubcalc` example and the `singlecelltest` executable) help development by providing short-term directional goals.

- **Documentation** is central to the project, for all the reasons listed in the Section 1.1 [Orientation], page 2 section at the beginning. Good documentation has lots of internal redundancies, cross-references, a usable index, a lucid structure, etc. Also, one should never hesitate to add information even when it seems like it's too early. If need be, just prefix with a *FIXME:* tag.
- **Code quality** refers to much more than the absence of bugs. It encompasses **architecture** and a clear **large-scale design**; good and consistent programming practices; **testability**, which includes the use of logging directives in the source code (see Section 7.5 [Logging], page 73); etc.

In contrast, certain attributes of programming which (currently) take the back seat to the above include **performance** and **fault-tolerance**, for instance.

7.2 Development Timeline

In this section we collate the past events of the Monod implementation, and plan for its short-term future.

7.2.1 History

Coding on Monod was started on May 16, 2003, as a seriously part-time project, competing with the maintainer's work, social life and Slashdot reading.

The pre-release draft of Version 0.01 — the first version to be released on SourceForge — was finished on October 1st, 2003. Much cleaning up of the source code and documentation ensued.

7.2.2 Future

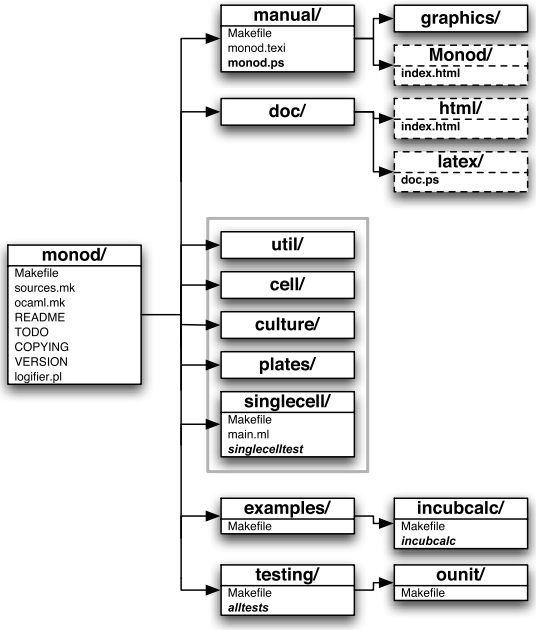
At the moment, the plan is to get to a fully-functioning Monod Culture prototype as soon as possible, while making sure that an acceptable architecture is laid out. Severe inefficiencies are tolerated in order to get to this state faster. The following Monod framework milestones are currently planned:

- v0.1: A single Monod Cell running a hand-coded genome that successfully performs arithmetic computations.
- v0.3: A Monod Culture is able to do polynomial symbolic regression with noisy input.
- v0.5: A Monod Culture able to co-evolve a single cell genome that successfully performs arithmetic operations on binary strings along with the gradual evolution of the environment by providing increasingly long strings.
- v0.8: A Monod Culture able to *learn* a given BNF grammar through co-evolution of a multi-cell genome in a priming environment of examples and counter-examples, and to probabilistically assess the validity of prompted strings.
- v1.0: Code and architecture cleanup, documentation up-to-date, etc.
- v2.0: One pass through the whole '`TODO`' file, which contains many postponed improvements to the program.

In parallel with these implementation milestones, many experiments are planned using the Monod framework. We do not list those here.

7.3 Source Code Structure

The directory structure of the source code is shown graphically in the following figure.



Each box represents a directory. The arrows indicate directory inclusion. Dashed boxes represent directories which are automatically created. Some of the important files are listed under certain directories. The files appearing in bold point to *product* files, that is, files created as the result of a `make` process — either executables, libraries or documentation. Files in bold and italic correspond to executables.

We ignore CVS-related files and directories, such as '`CVS`' directories and '`.cvsignore`' files.

The following table gives some details about each of the directories and files above.

'`monod/`' The main, top-level directory of the project. We have attempted to subdivide the project in meaningful directories and to reduce the clutter in this directory. The main files which can be found here are:

- '`Makefile`' is the main Makefile of the Monod project. It recursively calls the deeper Makefiles in order to create the main targets. However, it offers little control over the results — for this, the dependent Makefiles should be used directly. See Section 6.1.2 [Compilation], page 63 above for information about the main targets.
- '`sources.mk`' is the central repository of all the sources which form a part of Monod. It is kept somewhat structured so that different targets may access different sources. All the other Makefiles (almost!) include this file.
- '`ocaml.mk`' contains all the build rules related to OCaml targets. It is copied from '`OCamlMakefile`' and substantially modified to accommodate

our needs. See Section 6.1.1.3 [OCaml], page 62 earlier. This file is also included by all Makefiles which build OCaml targets.

- '`README`', '`COPYING`', '`VERSION`' and '`TODO`' contain the usual information contained in those files — mostly pointing back to the present manual. The '`TODO`' file is the most important, serving as a brain-dump.
- '`logifier.pl`' is a Perl script which is used pre-process all the OCaml source files to introduce logging messages. See Section 7.5 [Logging], page 73 below.

'`manual/`' Contains the source and build space of the manual which you are reading.

- '`Makefile`' is the manual's own Makefile. The main targets are *`html`* or *`ps`* for online or printed documentation, respectively; *`all`* for both; and cleanup targets.
- '`monod.texi`' is the single-file source of the whole manual, in Texinfo format. May be broken down into chunks later.
- '`monod.ps`' is the output of `make ps`. It is a printable manual.

In addition to the above files, the directory contains two subdirectories.

'`graphics/`' should be a symbolic link to a directory containing the graphics. This directory must be downloaded separately, since we do not save the graphics in the CVS repository (they're too large). See Section 6.2.1 [Manual], page 63 for more information.

The graphics are saved in many different formats. However, to edit them, or add graphics to the directory, at this point please contact the maintainer.

'`Monod/`' is an automatically generated directory and holds the generated HTML files. The head HTML file is '`index.html`'.

'`doc/`' This directory contains the source code documentation automatically extracted by `ocamldoc`. The directory itself is automatically generated when needed. It contains two subdirectories.

'`html/`' contains the HTML output, which can be access from the file '`index.html`' in that directory.

'`latex/`' contains the printable output, which is currently no used.

'`util/`' This directory, along with '`cell/`', '`culture/`', '`plates/`' and '`singlecell/`', contains the core source of the Monod project. The '`util/`' directory holds those sources which are generically useful tools, not necessarily tied directly to Monod. They range from logging to hash table extensions to architectural design patterns.

'`cell/`' Contains the implementation for the five lower layers of the Design Stack, up to the Monod Cell layer — that is, all the layers except for the Monod Culture. There is no explicit target associated with this directory. However, certain examples use these layers alone, which is why they are isolated.

'`culture/`'

Contains the implementation of the various features attached to the Monod Cultures, such as genetic algorithms, evolutionary strategies, etc. FIXME: Currently empty!

‘`plates/`’ The Monod architecture calls for applications to be divided in to *tectonic plates*, each embodying major parts of the functionality. The reusable plates and the code common to them is located in this directory. See Section 7.4 [Architecture], page 73 for more detail.

‘`singlecell/`’
This directory contains files that are very much in progress, and currently the only executable of the project. The files might change and/or be moved to the ‘`examples/`’ directory.

‘`examples/`’
Contains standalone executable applications demonstrating particular aspects of the project. The current examples include:

- `incubcalc`

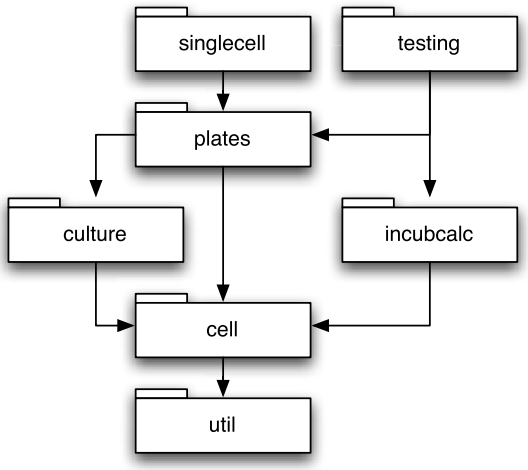
See Section 6.3.4 [Provided Examples], page 67 in the previous chapter for more details.

‘`testing/`’
This directory contains the unit test suite code and other tests. Run `make bytecode` or `make native` to create the unit test suite executable, called ‘`alltests`’.

The ‘`ounit/`’ subdirectory contains the (free-licensed) source code for an accessory unit test package developed independently from Monod, which we have included (and will perhaps modify) for convenience. OUnit is a unit test framework analogous of JUnit. The original distribution can be found at `http://home.wanadoo.nl/maas/ocaml/`.

Note from the above that each executable Monod target is controlled by a dedicated Makefile. This (small) restriction is imposed by limitations of ‘`OCamlMakefile`’.

While the source code is not formally divided into packages, there is a clear structure to the distribution and dependencies between the files. A coarse representation of the dependencies is shown in the following figure.



7.3.1 SourceForge Administration

The Monod project resides on the SourceForge collaborative site. The project home page, at http://monod.sourceforge.net, is not part of the source code distribution, and is currently only owned by the maintainer, along with a special-purpose distribution Makefile to package the entire web site and publish it to the SourceForge server. This is still done manually, as the frequency of changes is very low. Eventually, this section should contain information appropriate to maintain the site, once it becomes more complex and changes more rapidly.

7.4 Architecture

FIXME: Large-scale architecture of the project. Design stack in implementation, and tectonic plates starting at the level of the Monod Cell. Insert graphic. '`main`' file for every executable. Etc.

7.5 Logging

Logging is a cross-cutting aspect of the implementation of Monod. Logging messages are distributed throughout the source code. Depending on compile-time and run-time variables, the messages are streamed to a specified output as the program is executed. It's very much like spreading *`print`* statements through the code, but in a more controlled fashion.

The log messages are output to the file '`monod.log`' located where the executable is launched. FIXME: change the output using a command-line option and rotate the output files.

Log messages have the following components:

- A *logging level*, described below;
- A *thread number*, indicating the thread ID of the thread issuing the message;
- A *time stamp*, valid up to the second;
- The *filename* of the source file containing the message;
- The *line number* of the message is the source file;
- The message-specific *contents*.

All of these components are easily identifiable in the output. An example is shown below.

```
*** Info *** for thread 5 at 10/2 05:32:42 in swarm.ml line 387
Runner thread blocking
*** Info *** for thread 6 at 10/2 05:32:42 in swarm.ml line 423
Matcher thread waking up
*** Exec *** for thread 6 at 10/2 05:32:42 in swarm.ml line 178
Entering calculate_all_matches
*** Exec *** for thread 6 at 10/2 05:32:42 in incubator.ml line 434
Calculating the full marker for a procunit
*** Data *** for thread 6 at 10/2 05:32:42 in incubator.ml line 445
Procunit marker is matcher "BBB"
```

Each message consists of two lines, with the contents appearing on the second line.

The log levels are simply an ordered set of names which indicate the *importance* of a message. There are 11 possible log levels. Along with a description of their meaning, they are:

`Emerg` This event needs a human intervention to recover. The entire system is compromised and execution is aborted.

`Alert` Something went wrong and at least one thread of execution was not recovered.

`Error` Recovered error. Needs investigation.

`Warning` Something is fishy. Needs investigation.

`Exceptional`
A secondary code flow.

`Config` Configuration setup of change.

`User` Used-invoked alteration.

`Info` Informational message. Something significant has happened.

`Exec` Tracing the execution flow.

`Debug` Debugging information.

`Data` Parameter dumps.

The log level names which appear above can be passed as the parameter to the `--log-level` command-line argument, as described in Section 6.3.1 [Command-line Arguments], page 64 (capitalization is important). When set to a certain level, only those messages which have an importance at least equal to that level will be output. Currently, the default log level for all Monod executables is `Warning`.

To add log messages to the source code, simply write a single line of the form

```
LOG <level> "Message to log; %s is a string variable" astring;
```

where `level` is one of the levels given above, and the message can be anything, including standard `printf` arguments. However, the message *must* be contained on a single line and end with a semicolon.

All source files are pre-processed with a Perl script, '`logifier.pl`', which is located in the main directory. This script replaces all occurrences of the `LOG` keyword with the appropriate (and optimized) OCaml code. The Perl script is too simple to process multi-line messages.

The logging level has a very significant impact on the performance of Monod, to the point where certain tests can fail depending on the log level. For instance, the Incubator-layer unit tests check that a certain number of bindings are executing within a certain period of time. Running the test with a log-level equal to `Emerg` will make the assertion pass, while setting it equal to `Data` will make it fail. A bug which only appears under certain debugging conditions is called a *Heisenbug*. Be wary of the logging level...

7.6 Simulation-Specific Algorithms

The bulk of the description of Monod, in Chapter 2 [Three Design Patterns], page 15, Chapter 3 [The Cytoplasm and the Monod Cell], page 34 and Chapter 4 [Monod Cultures], page 59, is independent of the implementation. While the model may be described in detail, there are possibly many different ways to implements it. In particular, we have identified certain significant parts of the implementation which we call *simulation-specific algorithms*, because they are particularly tied to the von Neumann architecture of the substrate of the simulation. The details of these algorithms have no relevance on the Monod model proper — since they are purely in the real of implementation — other than on performance characteristics such as performance, but they are complex enough to warrant a separate section.

7.6.1 The Swarm Design Pattern Implementation

The Swarm design pattern, described formally in Section 2.2 [The Swarm Design Pattern], page 17, probably has the most important simulation-specific implementation. (Note that there is no separate Hive implementation - rather, the Swarm is the first layer in the source code.) Implementing the Swarm on a von Neumann machine is very much like fitting a (big) square peg in a round hole. The explicit parallelism of the Swarm screams for non-traditional computational substrates, but they're unfortunately not malleable enough today to accommodate the requirements. Plus the Monod developers don't have access to them. So there.

We propose below three different implementations of the Swarm. Having different implementations demonstrates the substrate independence of the Monod model. In fact, it allows the developers to debug the model more effectively, as this independence is a requirement.

7.6.1.1 Serialized Swarm

The current implementation of the Swarm is contained in the `SerializedSwarm` functor in the '`cell/swarm.ml`' file. `SerializedSwarm` is a particularly inefficient implementation. The two most important aspects of the implementation are: 1) how does the Swarm parallelize the execution of the residents? and 2) what is the strategy for finding matches between the projections?

The `SerializedSwarm` creates a single thread, the *runner thread*, for the execution of all the residents. A single list of all the active residents is kept, and the thread iterates over the list, calling the `click` method of each resident in turn. When a resident returns `false` from the call, indicating that it has no further work to do, it is taken off the list. It can be added to it externally or through a match being found.

When the active list is empty (even if there are many residents), the runner thread blocks so that the CPU does not get into a tight loop. Any change to the active list wakes up the thread.

Another thread is created, the *matcher thread*, whose single job is to find matches between all the projections and execute them. In the `SerializedSwarm`'s implementation, the matcher thread simply loops. At every loop, all the projections from all the residents are collected, the markers extracted, and the set of all pairs of markers is constructed. That set is then filtered by checking for matches, the ordered, and then bindings are attempted. The algorithm is completely memoryless, in that the entire set is reconstructed at every

turn even if no residents are added or projections changed. However, the matcher thread will block if no matches are found, to prevent a tight loop. The thread will awaken if there is any change to the list of active projections.

7.6.1.2 True Threaded Swarm

The true threaded Swarm implementation creates `n` pairs of threads. Each pair has a matcher and a runner thread, like in the serialized Swarm above. However, here the pairs execute concurrently with one another. The main difficulty in this model is the preservation of the Swarm design model, which requires a single thread of execution for each individual resident. As each resident may be called from many different modalities — namely, clicking, binding and releasing — this requirement takes some work.

FIXME: This implementation does not exist yet.

7.6.1.3 Distributed Swarm

The distributed Swarm expands on the true threaded Swarm by allowing the various pairs of threads to run on different computers. These *Swarm chunks* communicate with one another using remote method invocation. The effect is a truly parallelized Swarm.

FIXME: This implementation does not exist yet.

7.6.2 Cytoplasm Topology Implementation

FIXME: Nothing here yet.

Appendix A Combinations

This appendix presents examples of proteins. This exercise is useful to settle the terminology, to validate the definitions and to introduce potentially useful proteins.

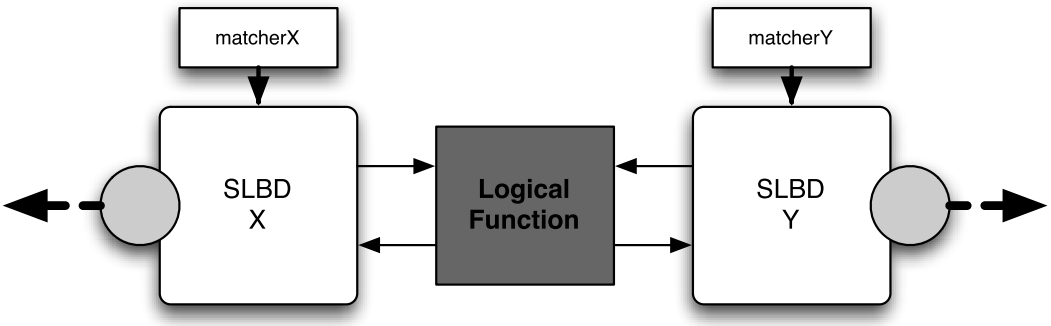
A.1 Two Ligand Binding Sites

This sections explores all the possible proteins which have precisely two ligand binding projection sites, no structural binding projection site and which are made out of the following domains flavors: SLBD, LBDR, LID and BMD (see Section 3.2.1 [Domains], page 43). We further require that all matcher inputs be static.

We can first divide this set into those proteins which perform remapping and those that don't.

A.1.1 Without Remapping

A protein with two ligand projection sites and without remapping must have exactly two SLBDs. Since we require that matching inputs be static, all we have left is to determine the mapping between the two boolean expressors and the two boolean acceptors, as shown in the following figure.



In order to complete the picture, we need to determine the logical function in the middle and to decide on whether the two matchers are the same or are different.

A.1.1.1 Truth-table Driven Logical Functions

One class of logical functions is especially simple to classify: those which settle immediately to a predictable output that depends on the input according to a simple truth table. As there are two inputs, the binding states for the SLBDs X and Y, and two outputs, the repress states for the SLBDs X and Y, there are 256 possible tables, symbolized in the figure below.

All the proteins definitions in this category are realizable as finite, stabilizing and well-defined proteins. (FIXME: Argument?)

We can arrange the 8 variables as a vector $(R_1^X, R_2^X, R_3^X, R_4^X, R_1^Y, R_2^Y, R_3^Y, R_4^Y)$ in order to classify all the possible tables. As the system is clearly symmetrical with respect to the interchange of the X and Y SLBDs, we can omit those tables which are "under the diagonal", which is what we do in the table below. This listing thus has 136 entries. This listing is certainly overkill, but it was useful in order to think more clearly about the issues involved. There is certainly a representation of the problem which takes into account the semantic relationships between the various entries of the truth table but sheer laziness made us go through the raw listing.

$(0, 0, 0, 0, 0, 0, 0, 0)$
: This protein binds to two ligands and never releases them. It is thus deleterious.

$(0, 0, 0, 1, 0, 0, 0, 0)$
: This protein can not release any ligand bound to Y. Deleterious.

$(0, 0, 0, 1, 0, 0, 0, 1)$
: Will bind to either the X or Y ligand individually, and release both when both are simultaneously bound. This protein is potentially **useful** in order to conditionally control the concentration of ligands in the Cytoplasm. However, its symmetry makes it difficult to harness.

$(0, 0, 1, 0, 0, 0, 0, 0)$
: Can't release Y. Deleterious.

$(0, 0, 1, 0, 0, 0, 0, 1)$
: Will bind to X but release it immediately. Will also bind to Y, and release it when further bound to X, at which point it will also release X. This protein can also be used to control the concentration of the Y ligand by binding to it until X is released. Hence it is **useful**. However, the repeated binding and releasing of X is wasteful. The same functionality is exhibited more optimally by the $(1, 0, 1, 1, 0, 0, 1, 1)$ protein.

$(0, 0, 1, 0, 0, 0, 1, 0)$
: This protein will never release a ligand bound to Y. Indeed, if X is also bound, then both ligands are bound and nothing will happen. Deleterious.

$(0, 0, 1, 1, 0, 0, 0, 0)$
: Can't release Y. Deleterious.

$(0, 0, 1, 1, 0, 0, 0, 1)$
: Same as $(0, 0, 1, 0, 0, 0, 0, 1)$ but slightly more efficient since when bound ligands are bound, they are released together immediately. **Useful.**

$(0, 0, 1, 1, 0, 0, 1, 0)$
: Can't release Y. Deleterious.

$(0, 0, 1, 1, 0, 0, 1, 1)$
: Pretty much the same as $(0, 0, 1, 0, 0, 0, 0, 1)$.

$(0, 1, 0, 0, 0, 0, 0, 0)$
: Can't release Y. Deleterious.

$(0, 1, 0, 0, 0, 0, 0, 1)$
: When bound to Y, will repress X. But if X is bound first, it can't be released. Deleterious.

$(0, 1, 0, 0, 0, 0, 1, 0)$
: Can't release X. Deleterious.

$(0, 1, 0, 0, 0, 0, 1, 1)$
: Can't release X. Deleterious.

$(0, 1, 0, 0, 0, 1, 0, 0)$
: Can't release X. Deleterious.

$(0, 1, 0, 1, 0, 0, 0, 0)$
: Can't release Y. Deleterious.

$(0, 1, 0, 1, 0, 0, 0, 1)$
: This protein looks like it could be used to control the concentration of the X ligand. X will bind, until the moderator Y appears to release both. However, when Y is bound alone, X is repressed and Y will never be repressed. Hence, this protein is deleterious.

$(0, 1, 0, 1, 0, 0, 1, 0)$
: Can't release Y. Deleterious.

$(0, 1, 0, 1, 0, 0, 1, 1)$
: Won't release if Y binds first, as X will be repressed. Deleterious.

$(0, 1, 0, 1, 0, 1, 0, 0)$
: This protein can be used to control the concentration of the X ligand. X will bind and be released by the moderator Y. Y will then alone be bound, at which time is will be released. The functionality of this protein, however, is more optimally accomplished by another protein. **Useful.**

$(0, 1, 0, 1, 0, 1, 0, 1)$
: Same as above, but slightly more efficient. Still, Y will bind and be released needlessly if it is present without X in the Cytoplasm.

$(0, 1, 1, 0, 0, 0, 0, 0)$
: Can't release Y. Deleterious.

$(0, 1, 1, 0, 0, 0, 0, 1)$
Y won't be released if bound first. Deleterious.

$(0, 1, 1, 0, 0, 0, 1, 0)$
Can't release Y. Deleterious.

$(0, 1, 1, 0, 0, 0, 1, 1)$
Y won't be released if bound first. Deleterious.

$(0, 1, 1, 0, 0, 1, 0, 0)$
Neither the X or the Y ligand will stick alone to this protein. While it won't hog any ligand, it won't do anything useful either. Useless and wasteful.

$(0, 1, 1, 0, 0, 1, 0, 1)$
Won't bind to anything. Useless and wasteful.

$(0, 1, 1, 0, 0, 1, 1, 0)$
Won't bind to anything. Useless and wasteful.

$(0, 1, 1, 1, 0, 0, 0, 0)$
Can't release Y. Deleterious.

$(0, 1, 1, 1, 0, 0, 0, 1)$
Control the concentration of Y ligand. However, X will bind and be released continuously, impairing efficiency. **Useful.**

$(0, 1, 1, 1, 0, 0, 1, 0)$
Can't release Y. Deleterious.

$(0, 1, 1, 1, 0, 0, 1, 1)$
Control the concentration of Y ligand. However, X will bind and be released continuously, impairing efficiency. **Useful.**

$(0, 1, 1, 1, 0, 1, 0, 0)$
Won't bind to anything. Useless and wasteful.

$(0, 1, 1, 1, 0, 1, 0, 1)$
Won't bind to anything. Useless and wasteful.

$(0, 1, 1, 1, 0, 1, 1, 0)$
Won't bind to anything. Useless and wasteful.

$(0, 1, 1, 1, 0, 1, 1, 1)$
Won't bind to anything. Useless and wasteful.

$(1, 0, 0, 0, 0, 0, 0, 0)$
Can't release Y. Deleterious.

$(1, 0, 0, 0, 0, 0, 0, 1)$
Can't release X. Deleterious.

$(1, 0, 0, 0, 0, 0, 1, 0)$
Can't release X. Deleterious.

$(1, 0, 0, 0, 0, 0, 1, 1)$
Can't release X. Deleterious.

$(1, 0, 0, 0, 0, 1, 0, 0)$
Can't release X. Deleterious.

$(1, 0, 0, 0, 0, 1, 0, 1)$
Can't release X. Deleterious.

$(1, 0, 0, 0, 0, 1, 1, 0)$
Can't release X. Deleterious.

$(1, 0, 0, 0, 0, 1, 1, 1)$
Can't release X. Deleterious.

$(1, 0, 0, 0, 1, 0, 0, 0)$
Can't release X. Deleterious.

$(1, 0, 0, 1, 0, 0, 0, 0)$
Can't release Y. Deleterious.

$(1, 0, 0, 1, 0, 0, 0, 1)$
Control the concentration of Y. X will behave well when Y is not present. One of the optimal representations of the functionality. **Useful.**

$(1, 0, 0, 1, 0, 0, 1, 0)$
Can't release Y. Deleterious.

$(1, 0, 0, 1, 0, 0, 1, 1)$
Control the concentration of Y. X will behave well when Y is not present. One of the optimal representations of the functionality. **Useful.**

$(1, 0, 0, 1, 0, 1, 0, 0)$
Won't hold on to Y. Useless and wasteful.

$(1, 0, 0, 1, 0, 1, 0, 1)$
Won't hold on to Y. Useless and wasteful.

$(1, 0, 0, 1, 0, 1, 1, 0)$
Won't hold on to Y. Useless and wasteful.

$(1, 0, 0, 1, 0, 1, 1, 1)$
Won't hold on to Y. Useless and wasteful.

$(1, 0, 0, 1, 1, 0, 0, 0)$
Can't release Y. Deleterious.

$(1, 0, 0, 1, 1, 0, 0, 1)$
Both ligands repressed from the start. Useless.

$(1, 0, 1, 0, 0, 0, 0, 0)$
Can't release Y. Deleterious.

$(1, 0, 1, 0, 0, 0, 0, 1)$
Control the concentration of Y. X will behave well when Y is not present. However, it will be released in two steps after it binds to catalyze the release of Y. **Useful.**

$(1, 0, 1, 0, 0, 0, 1, 0)$
Can't release Y. Deleterious.

$(1, 0, 1, 0, 0, 0, 1, 1)$
Control the concentration of Y. X will behave well when Y is not present. However, it will be released in two steps after it binds to catalyze the release of Y. **Useful.**

$(1, 0, 1, 0, 0, 1, 0, 0)$
Won't bind to anything. Useless and wasteful.

$(1, 0, 1, 0, 0, 1, 0, 1)$
Won't bind to anything. Useless and wasteful.

$(1, 0, 1, 0, 0, 1, 1, 0)$
Won't bind to anything. Useless and wasteful.

$(1, 0, 1, 0, 0, 1, 1, 1)$
Won't bind to anything. Useless and wasteful.

$(1, 0, 1, 0, 1, 0, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 0, 1, 0, 1, 0, 0, 1)$
Both ligands repressed from the start. Useless.

$(1, 0, 1, 0, 1, 0, 1, 0)$
Both ligands repressed from the start. Useless.

$(1, 0, 1, 1, 0, 0, 0, 0)$
Can't release Y. Deleterious.

$(1, 0, 1, 1, 0, 0, 0, 1)$
Control the concentration of Y. X will behave well when Y is not present. One of the optimal representations of the functionality. **Useful.**

$(1, 0, 1, 1, 0, 0, 1, 0)$
Can't release Y. Deleterious.

$(1, 0, 1, 1, 0, 0, 1, 1)$
Control the concentration of Y. X will behave well when Y is not present. One of the optimal representations of the functionality. **Useful.**

$(1, 0, 1, 1, 0, 1, 0, 0)$
Won't bind to anything. Useless and wasteful.

$(1, 0, 1, 1, 0, 1, 0, 1)$
Won't bind to anything. Useless and wasteful.

$(1, 0, 1, 1, 0, 1, 1, 0)$
Won't bind to anything. Useless and wasteful.

$(1, 0, 1, 1, 0, 1, 1, 1)$
Won't bind to anything. Useless and wasteful.

$(1, 0, 1, 1, 1, 0, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 0, 1, 1, 1, 0, 0, 1)$
Both ligands repressed from the start. Useless.

$(1, 0, 1, 1, 1, 0, 1, 0)$
Both ligands repressed from the start. Useless.

$(1, 0, 1, 1, 1, 0, 1, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 0, 0, 0, 0, 0)$
Can't release Y. Deleterious.

$(1, 1, 0, 0, 0, 0, 0, 1)$
Can't release X. Deleterious.

$(1, 1, 0, 0, 0, 0, 1, 0)$
Can't release X. Deleterious.

$(1, 1, 0, 0, 0, 0, 1, 1)$
Can't release X. Deleterious.

$(1, 1, 0, 0, 0, 1, 0, 0)$
Can't release X. Deleterious.

$(1, 1, 0, 0, 0, 1, 0, 1)$
Can't release X. Deleterious.

$(1, 1, 0, 0, 0, 1, 1, 0)$
Can't release X. Deleterious.

$(1, 1, 0, 0, 0, 1, 1, 1)$
Can't release X. Deleterious.

$(1, 1, 0, 0, 1, 0, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 0, 1, 0, 0, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 0, 1, 0, 1, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 0, 1, 0, 1, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 0, 1, 1, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 1, 0, 0, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 1, 0, 0, 0, 1)$
When Y is bound, X can't bind to release it because it is repressed. Deleterious.

$(1, 1, 0, 1, 0, 0, 1, 0)$
Can't release Y. Deleterious.

$(1, 1, 0, 1, 0, 0, 1, 1)$
When Y is bound, X can't bind to release it because it is repressed. Deleterious.

$(1, 1, 0, 1, 0, 1, 0, 0)$
Won't hold on to Y. Useless and wasteful.

$(1, 1, 0, 1, 0, 1, 0, 1)$
Won't hold on to Y. Useless and wasteful.

$(1, 1, 0, 1, 0, 1, 1, 0)$
Won't hold on to Y. Useless and wasteful.

$(1, 1, 0, 1, 0, 1, 1, 1)$
Won't hold on to Y. Useless and wasteful.

$(1, 1, 0, 1, 1, 0, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 1, 1, 0, 0, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 1, 1, 0, 1, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 1, 1, 0, 1, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 1, 1, 1, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 0, 1, 1, 1, 0, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 0, 0, 0, 0, 0)$
Can't release Y. Deleterious.

$(1, 1, 1, 0, 0, 0, 0, 1)$
When Y is bound, X is repressed. Deleterious.

$(1, 1, 1, 0, 0, 0, 1, 0)$
Can't release Y. Deleterious.

$(1, 1, 1, 0, 0, 0, 1, 1)$
When Y is bound, X is repressed. Deleterious.

$(1, 1, 1, 0, 0, 1, 0, 0)$
Won't bind to anything. Useless and wasteful.

$(1, 1, 1, 0, 0, 1, 0, 1)$
Won't bind to anything. Useless and wasteful.

$(1, 1, 1, 0, 0, 1, 1, 0)$
Won't bind to anything. Useless and wasteful.

$(1, 1, 1, 0, 0, 1, 1, 1)$
Won't bind to anything. Useless and wasteful.

$(1, 1, 1, 0, 1, 0, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 0, 1, 0, 0, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 0, 1, 0, 1, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 0, 1, 0, 1, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 0, 1, 1, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 0, 1, 1, 0, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 0, 1, 1, 1, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 1, 0, 0, 0, 0)$
Can't release Y. Deleterious.

$(1, 1, 1, 1, 0, 0, 0, 1)$
When Y is bound, X is repressed. Deleterious.

$(1, 1, 1, 1, 0, 0, 1, 0)$
Can't release Y. Deleterious.

$(1, 1, 1, 1, 0, 0, 1, 1)$
When Y is bound, X is repressed. Deleterious.

$(1, 1, 1, 1, 0, 1, 0, 0)$
Won't bind to anything. Useless and wasteful.

$(1, 1, 1, 1, 0, 1, 0, 1)$
Won't bind to anything. Useless and wasteful.

$(1, 1, 1, 1, 0, 1, 1, 0)$
Won't bind to anything. Useless and wasteful.

$(1, 1, 1, 1, 0, 1, 1, 1)$
Won't bind to anything. Useless and wasteful.

$(1, 1, 1, 1, 1, 0, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 1, 1, 0, 0, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 1, 1, 0, 1, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 1, 1, 0, 1, 1)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 1, 1, 1, 0, 0)$
Both ligands repressed from the start. Useless.

$(1, 1, 1, 1, 1, 1, 0, 1)$
: Both ligands repressed from the start. Useless.

$(1, 1, 1, 1, 1, 1, 1, 0)$
: Both ligands repressed from the start. Useless.

$(1, 1, 1, 1, 1, 1, 1, 1)$
: Both ligands repressed from the start. Useless.

It is easy to see that all these proteins exhibit one of a few possible types of behavior. We can categorize the various possible behaviors as follows. (FIXME: The numbers don't add up - count by hand later.)

- **Deleterious** behavior. This characterizes proteins which bind to one or more ligand and can never release it. This behavior is considered deleterious because it corresponds to a dead end for entities in the Cytoplasm. 56 of the 136 entries above, or about 41%, exhibit deleterious behavior.
- **Useless and wasteful** behavior characterizes those proteins which ultimately do nothing, but which spend cycles binding and later releasing ligands. The waste has a definite negative impact on the performance of the system and as such should be considered deleterious as well, but we isolated it for reasons of precision. 32 entries above, or about 24%, exhibit useless and wasteful behavior.
- Those proteins which do not bind to anything and do nothing are **useless**. They do consume space in the Cytoplasm but little energy is expended keeping track of them so that they are less dangerous than the wasteful proteins above. 39 entries above, or about 29%, exhibit useless behavior.
- Some proteins are **useful but non-optimal**. These are proteins which have a putative function but which are nevertheless wasteful of the resources of the Cytoplasm. The possible functions that have been encountered are:
  - *Control of the concentration of the X ligand.* This function is represented by 2 non-optimal proteins.
  - *Control of the concentration of the Y ligand.* This function is represented by 5 non-optimal proteins.

  We do not count as non-optimal the staggered release shown by certain proteins, which does not cost much. Hence, about 5% of the proteins show useful but non-optimal behavior.
- Finally, a few proteins exhibit **useful, optimal** behavior. One of them performs symmetric moderation of the concentrations of the X or Y ligands, while 6 of them perform moderation of the Y ligand exclusively, but very well. About 4% of the proteins show useful and optimal behavior.

FIXME: Discussion about how the percentages above can not be interpreted as *probabilities* that a randomly generated protein is useful or whatever, for two reasons: the listing above does not reflect the choice of genetic encoding, which is how proteins are created to begin with; and we have only listed those proteins which are simply truth-table driven. The next section shows a lot more proteins composed of the same domains, which have different short-term behavior.

A.1.1.2 Other Logical Functions

FIXME: Show examples of finite proteins which *almost* follow a truth table pattern but which arrive to it after a few oscillations. Also show non-finite proteins.

A.1.2 With Remapping

A.1.2.1 Truth-table Driven Logical Functions

Appendix B References

[Alberts et al. 2002] **Molecular Biology of the Cell**, 4th edition; B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter. Published by Garland Science. 1463 pages.

The classic cellular biology textbook, and the book that started it all. Surprises on every other page. The focus is on biology, not on computation, of course, but there is a definite "cybernetic" tinge at times. The book is a wonder to read even for non-biologists (like me). It takes the time to build up from the absolute basics, and has a lucid structure. It is also filled to the brim with images.

[Banzhaf et al. 1998] **Genetic Programming: An Introduction**; Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, Frank D. Francone. Published by Morgan Kaufmann. 470 pages.

An introductury text on genetic programming, with an overview of different techniques and results. Superficial and often repetitive, but presents limitations of the field.

[Beyer 2001] **The Theory of Evolution Strategies**; Hans-Georg Beyer. Published by Springer. 380 pages.

A very useful synoptic introductory chapter, and then a whole lotta math leading to some surprising conclusions. The faulty english is never altogether confusing.

[Calvin 1996] **The Cerebral Code**; William H. Calvin. Published by MIT Press. 256 pages.

[Dawkins 1982] **The Extended Phenotype**; Richard Dawkins. Published by Oxford University Press. 313 pages.

[Deacon 1997] **The Symbolic Species**; Terrence Deacon. Published by W. W. Norton and Company. 527 pages.

Another primary inspiration for Monod. The book argues for the close coevolution of the human brain and of language, principally the semiotic aspects of it, and argues that most of what makes us human derives from this history. A principal goal of Monod is to provide an environment suitable for making experiments on coevolution.

Now whether one wishes to invoke the "Baldwin Effect" is a story we will stay away from!

[de Castro and Timmis 2002] **Artificial Immune Systems: A New Computational Intelligence Approach**; Leandro N. de Castro and Jonathan Timmis. Published by Springer. 357 pages.

A book whose focus is very close to that of the Monod project: to investigate biologically motivated computation. It tackles the properties of the immune system, which are among the most exciting: memory and quick adaptation. However, the area of reseach is clearly very young, with few interesting and original results yet.

[Edelman 1988] **Neural Darwinism: The Theory of Neuronal Group Selection**; Gerald M. Edelman. Published by Basic Books. 371 pages.

[Monod 1970] **Le Hasard et la Nécéssité**; Jacques Monod. Published by Éditions du Seuil. 243 pages

The one book that I would have liked to read back in a college biology class - or in philosophy, for that matter. A lucid and far reaching synthesis of the new direction of biology taken after the first discoveries of molecular protein pathways within the cell. Everything is there, from the second law of thermodynamics to materialistic dialectics.

[Sienko et al. 2003] **Molecular Computing**; edited by T. Sienko, A. Adamatzky, N. Rambidi, M. Conrad. Published by MIT Press. 257 pages.

I discovered this book when the first draft of the documentation was mostly finished (10/22/2003). Reading the introduction and the first chapter, my heart stopped because of the close similarities to Monod. Its stated goal, however, is very different from that of Monod: it is to investigate the appropriateness of non-traditional substrates (molecular, chemical, biological) for computation. Monod is not concerned with substrate. Nevertheless, many sections are concerned with the more theoretical questions of the impact to the notion of computation.

[Turney 1996] **Myths and Legends of the Baldwin Effect**; Peter Turney. 13th International Conference on Machine Learning (ICML96), Workshop on Evolutionary Computation and Machine Learning, Bari, Italy, July, pp. 135-142. (NRC #39220).

This article can be found online on the author's website:

http://members.rogers.com/peter.turney/.

The Baldwin Effect was one of the reasons the author embarked on the Monod project. This article is a very short, tremendous diatribe about its real meaning — and one of the few places where it is described adequately.

Concept Index

A

abstract domain . . . 43
acceptor interface . . . 43
active marker . . . 17
affinity . . . 17
allostery . . . 47
architecture . . . 69

B

behaved program . . . 25
behavior essence function . . . 53
behavior function . . . 52
behavior function — causal . . . 52
behavior function — finite . . . 52
behavior function — predictable . . . 53
behavior function — spacing independent . . . 53
binding . . . 17
biology . . . 34
BMD . . . 51
boolean multiplexor domain (BMD) . . . 51
boundary domain . . . 44
Brownian search . . . 18

C

calculator based on Incubator . . . 21, 67
causal behavior function . . . 52
Clarke, Arthur C. . . . 4
`click` method . . . 15
command-line arguments . . . 64
command-line arguments — `alltests` . . . 65
compartment . . . 37, 56
compilation . . . 63
concrete domain . . . 46
conformation change — domain . . . 45
conformation engine — domain . . . 46
CPU thrashing . . . 36
cut in an operation history . . . 53
Cytoplasm . . . 9

D

design patterns . . . 15
design stack . . . 8
determinism . . . 25
deterministic program . . . 25
direction — interface . . . 43
documentation . . . 2, 63, 69
domain . . . 43
domain assembly . . . 51
domain assembly — stabilizing . . . 55
domain assembly — well-defined . . . 56
domain assembly state . . . 54
domain conformation change . . . 45
domain conformation engine . . . 46
domain functional logic . . . 45
domain interface . . . 43
domain projection sites . . . 44
domains . . . 34

E

energy . . . 35, 57
error handling . . . 23
essentially equivalent behavior functions . . . 53
evolutionary operators . . . 11
execution threads . . . 15
exposure . . . 18
expressor interface . . . 43
extent of an operation history . . . 52

F

finite behavior function . . . 52
FIXME . . . 14
flavors of domains . . . 43
fragment of ligand . . . 20
fun . . . 14
functional logic — domain . . . 45

G

geometrical structure . . . 36
goals . . . 3
grounded Incubator . . . 24

H

h-behaved program . . . 26
h-deterministic program . . . 26
hallucinatory flash . . . 3
hardware . . . 61
harness . . . 15, 18, 21
Heisenbug . . . 74
history . . . 69
Hive . . . 8, 15
homologous crossover . . . 12

I

Incubator . . . 9, 20, 67
Incubator — at rest . . . 24
Incubator — grounded . . . 24
Incubator — primed . . . 24
Incubator — program . . . 24
Incubator input . . . 24
Incubator output . . . 24

input history . . . 52
interface — domain . . . 43
interface direction . . . 43
interface name . . . 44
interface type . . . 43
invalid amino acid sequence . . . 39
irreversibility . . . 35

J

Jacques Monod . . . 2

L

LBDR . . . 48
license . . . 2, 68
LID . . . 50
ligand . . . 20
ligand — semantically terminating . . . 25
ligand — terminating . . . 24
ligand binding domain with remapping (LBDR) . . . 48
ligand binding projection . . . 20
lock and key . . . 18
logging . . . 73
logical integration domain (LID) . . . 50

M

Makefiles . . . 63
marker . . . 17
matcher . . . 20
matching . . . 17
membrane transport . . . 36
milestones . . . 69
molecular computing . . . 12
molecular recognition . . . 18
Monod Cell . . . 9
Monod Culture . . . 10
Monod, Jacques . . . 2

N

name — interface . . . 44

O

OCaml . . . 62
`ocamldoc` . . . 64
operation — domain . . . 44
operation — domain assembly . . . 52
output history . . . 52

P

Pandemonium . . . 18
parallelization . . . 23
predictable behavior function . . . 53
primed Incubator . . . 24
probabilistic matching . . . 17
processing unit . . . 20
program — behaved . . . 25
program — deterministic . . . 25
program — well-grounded . . . 24
program for Incubator . . . 24
projection . . . 17
projection sites — domain . . . 44
projection, ligand binding . . . 20
projection, structural binding . . . 20
protein folding . . . 38
protein synthesis . . . 38

R

randomness . . . 16, 17
reaction-diffusion systems . . . 37
release . . . 35, 56
resident . . . 15
reversibility . . . 35

S

Selfridge's Pandemonium . . . 18
semantically terminating ligand . . . 25
shelter . . . 15
shifting an operation history . . . 52
Shortcomings of Contemporary Programming . . . 4
simple ligand binding domain (SLBD) . . . 47
Simulation of a Turing machine . . . 27
Slashdot . . . 69
SLBD . . . 47
snippet . . . 20
social life . . . 69
source code . . . 64, 70
SourceForge . . . 2, 68, 73
spacing an operation history . . . 53
spacing independent behavior function . . . 53
stabilizing domain assembly . . . 55
stack, design . . . 8
state — domain assembly . . . 54
state history . . . 54
structural binding projection . . . 20
Swarm . . . 8, 17

T

terminating ligand . . . 24
testing . . . 65, 68
thrashing . . . 36
threading . . . 15
topological structure . . . 36

Turing / von Neumann model 4
Turing machine — simulating 27
Turing soup 29
type — interface 43

W

well-defined domain assembly 56
well-grounded Incubator program 24