PhD Dissertation

International Doctorate School in Information and Communication Technologies

DIT - University of Trento

Scaling up Systems Biology: Model Construction, Simulation and Visualization

Lorenzo Dematté

Advisor:

Prof. Corrado Priami

Università degli Studi di Trento

March 2010

Abstract

Being a multi-disciplinary field of research, Systems Biology struggle to have a common view and a common vocabulary, and inevitably people coming from different backgrounds see and care about different aspects. Scientists have to work hard to comprehend each other and to take advantage of each other's work; however, they can provide unexpected and beautiful new insight to the problems we have to face, enabling cross-fertilization among different disciplines.

However, Systems Biology scientists all share one main goal, in the end: comprehend how a system as complex as a living creature can work and exists. Once we really understand how and why a biological system works, we can answer other important questions: can we fix it when it breaks down; can we enhance it and make it more resistant, correct its flaws; can we reproduce its behaviour and take it as inspiration for new works of engineering; can we copy it to make our everyday work easier and our human-created systems more reliable.

The contribution of this thesis is to push ahead the current state of art in different areas of information technology and computer science as applied to systems biology, in a way that could lead, one day, to the understanding of a whole, complex biological system. In particular, this thesis builds upon the current state of art of different disciplines: programming languages theory and implementation, parallel computing, software engineering and visualization. Work done in these areas is applied to Systems Biology, in the effort to scale up the dimension of the problems that is possible to tackle with current tools and techniques.

Keywords
[Systems Biology, Process Algebra, Simulation Algorithms, Visualization, Parallel Computing]

To my family, past, present and future.
You are my strength. I love you all.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis reflects my own journey into the land of systems biology, from a computer science, programming language designer and developer perspective. Being a multi-disciplinary field of research, systems biology struggle to have a common view and a common vocabulary, and inevitably people coming from different backgrounds see and care about different aspects; this is both the limitation and the beauty of this field of study. Scientists have to work hard to comprehend each other and to take advantage of each other's work; however, they can provide unexpected and beautiful new insight to the problems we have to face, enabling cross-fertilization among different disciplines.

In the end, systems biology scientists all share one main goal: comprehend how a system as complex as a living creature can work and exists. Once we really understand how and why a biological system works, we can answer other important questions: can we fix it when it breaks down; can we enhance it and make it more resistant, correct its flaws; can we reproduce its behaviour and take it as inspiration for new works of engineering; can we copy it to make our everyday work easier and our human-created systems more reliable. These questions, coming for medicine, biology, biotechnology, informatics, all spring from the same basic goal, and all depends on being able to reach that goal.

The contribution of this thesis can be seen as an effort of pushing ahead the current state of different areas of information technology and computer science as applied to systems biology, in a way that could lead, one day, to the understanding of a whole, complex biological system. In particular, this thesis builds upon the current state of art in programming languages theory and implementation, parallel computing, software engineering and visualization and applies it to systems biology, in the effort to scale up the dimension of the problems that is possible to tackle with current tools and techniques.

1.1 Title dissected

Thesis titles tends to be obscure, too short and general or too long and detailed. However, a thesis title is important, as it is the sum of the contribution done by the candidate to the world of research. In order to introduce the concepts expressed in this thesis, I will dissect the title and examine its parts, giving a short preview of the areas my work touched.

1.1.1 Scaling up

Scaling is a pretty common world in computer science. Many algorithms, methods, languages were invented or substantially improved because the previous ones did not scale up well. An algorithm, method or language *scales* when can seamlessly be used to undertake problems of increasing complexity, even across different orders of magnitude.

The term itself is not used in rigorous way: even if the underlying idea remains the same, its exact meaning varies when applied to different concepts.

As an example, consider programming languages: all general purpose programming languages are considered *equivalent* from a computational point of view. More precisely, every programming language that computes exactly the same class of functions as do Turing machines is said *Turing-equivalent*. All general-purpose languages in wide use can simulate, and be simulated by, a universal Turing machine, and therefore are Turing-equivalent. Simply put, the computational power of all general-purpose languages is the same: any computation that can be done in one general purpose language, can be expressed in any other general purpose language. However, in terms of *scaling*, they are different.

In theory is not impossible to build every known software in assembly language or machine code; however, for all practical purposes, nobody will ever dream of building a web application or a word-processor entirely in assembly, because it requires a huge effort, astonishing programming skills, almost infinite time; it lacks flexibility, composability - for example, existing parts cannot be reused in a simple and standardized way. Finally, it would not be possible to build the software as a team effort, because assembly is mostly a "write-only" language, hard to understand even for the original author.

Assembly is still useful for core system components, or pieces of software that require terrific speed of execution, but it is really bad for building big systems or to produce software as a team effort. In other words, programming in assembly does not *scale*.

The same can be said for algorithms -for which however the definition of scalability is strictly related to the formal notion of complexity, and therefore is more rigorous- for methodologies, and for patterns and practices.

1.1.2 Systems biology

Systems Biology is an inter-disciplinary field of study focusing on the comprehension of the complex dynamics regulating biological systems, using tools and concepts derived from other scientific fields like mathematics and computer science. Systems Biology studies the interactions between the components of biological systems to understand how these interactions give rise to the function and behaviour of that system.

The main difference between systems biology and more traditional disciplines lies in the world *system*; in Denis Noble words "is about putting together rather than taking apart, integration rather than reduction" [161]. Adopting a systemic view, scientist may discover new emergent properties and understand better the processes happening in a biological system:

> The reductionist approach has successfully identified most of the components and many of the interactions but, unfortunately, offers no convincing concepts or methods to understand how system properties emerge. [...] The pluralism of causes and effects in biological networks is better addressed by observing [...] multiple components simultaneously. - Uwe Sauer, Matthias Heinemann, Nicola Zamboni [201]

It is important to point out that Systems Biology do not fight against reductionism; admitting that it is necessary to develop a new way of thinking and a new method to understand what emerges from the interaction of fundamental components by no ways implies that the reductionist approach is flawed; on the other hand

> The ability to reduce everything to simple fundamental laws does not imply the ability to start from these laws and reconstruct the universe. [...] The constructionist hypothesis breaks down when confronted with the twin hypothesis of scale and complexity. [...] At each stage, entirely new laws, concepts and generalizations are necessary, requiring inspiration and creativity to just as great a degree as in the previous one. - P.W.Anderson [7]

Understanding how a whole systems works under the assumption that its behaviours is more than the simple sum of the parts is at the base of all Systems Biology efforts, and of all the efforts to understand Complex Systems in general. *Integration* becomes central to capture qualities and properties that cannot be observed, or explained, only from the basic parts. Quoting again Noble "It requires that we develop ways of thinking about integration that are as rigorous as our reductionist programmes, but different" [161]; on the same line Sauer et. al. [201] state that to understand complex systems "rigorous data integration with mathematical models" is necessary.

The importance of interconnections in living systems is also stated by S.E. Jorgensen [122]. Like the other system-level scientists, Jorgensen observes that many relevant living biological systems are too complex to be reproduced *in-vitro* in a laboratory, or even to be described down to their details. He advocates that the only way to try and comprehend them is to build models on the scale of the system itself. This does not mean scientists have to build a model based on phenomenological observation alone, nor a model entirely built in a completely mechanistic way that tries to explain everything in terms of fundamental laws. They have to find a way to express a system as a combination of factors from levels both higher and lower in the hierarchy, of observed behaviour and mechanistic modelling. Quoting Luca Cardelli "it is now evident that even when we are able to fully characterize a model from a mechanistic point of view, the model itself can express "emergent" phenomenological behaviour that is not evident from the parts list. Conversely, given a known behaviour and a long parts list, it is often difficult to identify the subset of the parts list that is responsible for the behaviour". The advent of molecular biology more than fifty years ago shifted biology from the classification and observation of species and components to the comprehension of how they work; now we are at the point in which we need a formal and precise way to map biological interactions, in order to learn how to "fix" them in a rational way, as expressed in an humorous way by Lazebnik [139] and Cardelli [32].

1.1.3 Model composition, Simulation and Visualization

The urgent need for a *formal* way to express interactions is a shared sentiment among the Systems Biology community. Indeed, there is much discussion and work around which language we should adopt to build *models*, but no one doubts that scientific modelling is a fundamental part of the scientific method as applied to Systems Biology.

I strongly agree with this view; even beyond Systems Biology, I cannot really say to have understood a principle, or even more a complex system, if I am not able to reproduce it. As Richard Feynman said "What I cannot create, I do not understand".

Generally speaking, the scientific method is a set of techniques, specifications and guidelines that are applied to a method of inquiry, used to acquire, integrate or correct knowledge about observable phenomena. The details and precise guidelines vary from one scientific field to another, but to be dubbed scientific, a method of inquiry must be based on some features: data has to be gathered in an observable, empirical and measurable way; hypothesis must be tested under the same constraints to gather evidence for its verification or confutation; the steps taken during this process must be repeatable in order to dependably predict any future results.

Scientific researchers apply the scientific method to their work; they build and propose

hypotheses as explanations of phenomena and current data, and design experimental studies to test these hypotheses. John Stuart Mill was one of the first to clearly outline what distinguishes a scientific method from other methods of investigation [154].

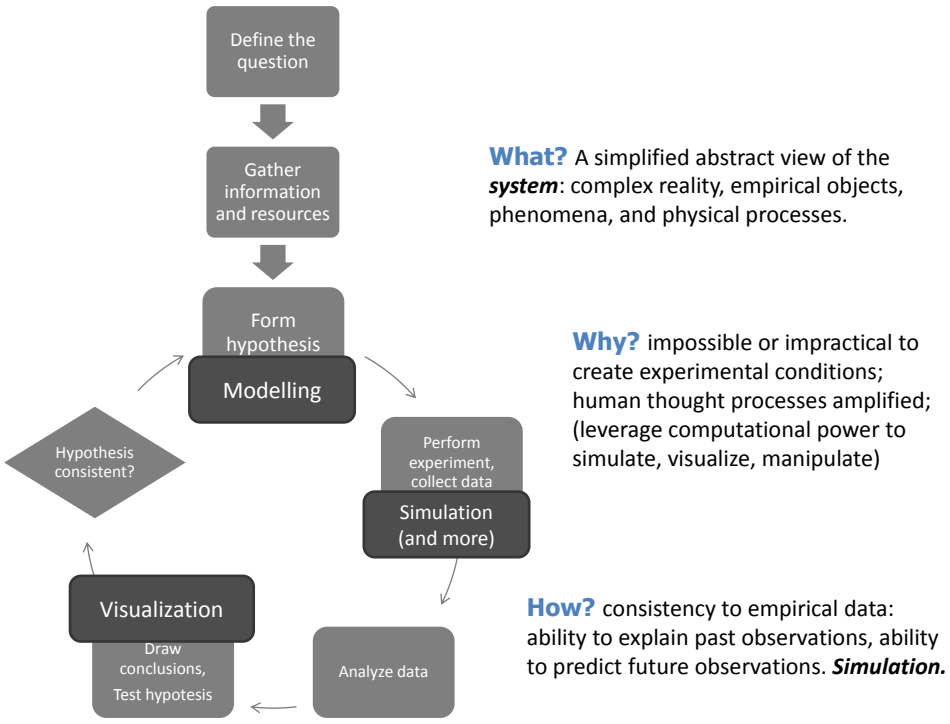


Figure 1.1: The scientific method applied to Systems Biology

Modelling is an essential part of the scientific method, and of almost all scientific activities; building a hypothesis about how a phenomena can be interpreted already implies the construction of a mental model of how existing data and knowledge could justify the phenomena subject of study. Many scientific disciplines have different ideas about specific types of modelling. In general, modelling is the process of generating abstract, conceptual, graphical or mathematical models.

In biology it is becoming incredibly difficult to build wet-lab experiments that elucidate the phenomenon at hand, especially for complex systems. Building models that explain all past experiments and predict the results of new experiments is the solution proposed by Systems Biology.

As we have seen, Systems Biology combines molecular biology with other field of study; in particular, mathematics at the beginning and now of computer science, enabled the possibility of *in-silico* experiments, done by building a model of the system and *solving* or *simulating* it on a computer. Eventually it is necessary to find real experimental validation, but modelling can make possible experiments of a complexity not manageable

before. Model composition, in-silico verification and predictions are part of the scientific method applied to Systems Biology (Fig. 1.1).

Simulation

Building a model gives by itself some useful insights on the process or system under study, as it forces the modeller to think about the system in a rigorous way; however, model construction is just the beginning. The scientific method requires the design of experiments to test hypothesis. Central to every experiment in systems biology is the *simulation* of the system under study; the use of computer based simulations for making predictions and testing hypothesis should come to no surprise, as it is now a common practice in many fields - from many-body physics to hydraulics, mechanical engineering and meteorology, just to cite a few ones.

In Computational Systems Biology, many techniques derived from formal verification and analysis of complex systems are under study for application to biological systems; these techniques can actually prove in a rigorous way some properties about a given model and the system it represents. However, stochastic simulation techniques that reproduce in an exact way the "chemical machine" on which models can run as a sort of "biological software" [75], are still the easiest, quickest and most used way to proceed and test the hypothesis underlying the simulated model.

Visualization

Finally, visual techniques for systems biology are a current topic of research, as they are useful at many levels. During my PhD I explored their use to design, organize and understand biological models. In particular, the latter application -visual techniques to *understand* data from a specific problem domain- falls into the area of *visualization* and may help to address the drawbacks and difficulties in understanding the results of simulations and analysis run on complex models.

The role of visualization is to extract information from raw data, the first step in the many-phase process that leads to *understanding* and ultimately to new knowledge [37]. Visualization takes basic data, composed of symbols taken from some grammar or language (e.g. series of doubles, graph structures, grid structures, DNA strings, ...) and processes them in order to present to the user *information*, using visual cues and graphical techniques. *Information*, in turn, is data processed to be useful, in a way that can provide answers to *who, what, where and when* questions. Ultimately, the correct mix of information and data help a scientist to understand the data and answer the most important question: the *how*, that gives new knowledge to the scientist.

1.2 Contribution

My PhD began three years ago with the rather general statement of doing *research on parallel techniques for simulation of biological systems.*

Even if this is a really interesting and challenging problem to face, as we shall see in due time, and even if despite some interesting progresses the problem is far from being "solved", during my first year I discovered that the size of a problem is not limited only by the simulation algorithm. The limit to the size of biological systems that can be simulated or analysed lies in a more fundamental reason: the inherent complexity of biological systems. Making progress in the area of parallel algorithms for computation biology is really important, especially in the light of the so called *concurrency revolution* that is taking part in these last years. But a parallel and scalable algorithm (for simulation, analysis, static checking, ...) does no good by its own, because we will be never able to use it and exploit its capabilities. When I started my PhD in 2006, I quickly discovered that it was not possible, given the present process-algebra based tools and techniques, to create a correct, big enough model to run on this hypothetical scalable algorithm. Nor it was possible to take the sheer amount of data produced by the hypothetical algorithm and interpret them in some meaningful way.

Scaling the simulation and analysis algorithms to take advantage of the computational power offered by today's and tomorrow's multi-core and many-core architectures is a great area of research, and it must be done in order to be able to understand not only single or isolated pathways, but also how tissues and cells works -especially if we want to overcome today's way of modelling systems from their macroscopic behaviour and deliver the promise done by systems biology of understand the whole by composing the parts-. But it is also very important to make progress in other areas, or we will be able to run, simulate and analyse big models, but we will not be able to compose them, or we will not able to understand the results of a simulation.

For this reason, the problem of scaling up systems biology spans over four main areas:

- **Scale at model composition level**: as in the assembler example, basic languages used for the first systems biology models are not able to scale. The main limitation is exactly the same found in all general purpose language: the lack of modularity, the impossibility of re-using modules and components written by someone else without having to rewrite them almost completely (easy *reuse*) and the lack of ability to build a complex software (or model, in this case) composing simpler modules, treating them in an (almost) black-box way. We advocate the use of a *programming language for systems biology*, and a programming discipline to model construction, in order to overcome these limitations. The contribution of the author in this area is its work

as part of the team that defined, refined and implemented the BlenX language.

- **Scale at algorithm level**: the Gillespie algorithm is the most widely used and known algorithm of systems biology: it made possible to perform accurate and physically correct simulations of realistic biochemical systems . However, it has some limits: as we try to build and simulate significantly more complex and vast models we are bound to collide with one of its assumption. The Gillespie algorithm, in fact, assume that space is *homogeneous*; clearly this assumption can hold at the level of a single pathway (even if even in this case it is necessary to make distinctions) but cannot be considered at a whole-cell level, or when dealing with inter-cellular interactions. The contribution of this thesis here is the design and implementation of a variant of the Gillespie algorithm for spatial, heterogeneous systems.

- **Scale at execution level**: the original goal of my PhD, design simulation algorithms to take advantage of the computational power offered by today and tomorrow multi-core and many-core architectures. This is particularly important, as the way in which new processors are built has changed dramatically in the last four years. Previously, programmers relied on Moore's law to deliver exponentially increasing performances to their users. This empirical law states that the number of transistors on a CPU, and so its computational power, is bound to double every 18 months. This law held since the transistor was invented, and it is still valid today. However, up to four years ago, an increase of computational power on a CPU meant a decrease in the number of CPU cycles necessary to complete the execution of an instruction (that changed from hundred to tens to multiple instruction per cycles, thanks to instruction level parallelism) and/or an increase in frequency (how many CPU cycles can be done in a second). The consequence is that existing programs could immediately benefit from new processors, running at roughly twice the speed (if they were CPU-bound) every new processor release. Around 2004, the last single core CPUs hit some physical barriers -power consumption, speed of signal propagation- and since then CPU frequency stabilized to 3-4 GHz and has not grown in an appreciable way. At the same time, instruction level parallelism continued to improve, but not in a very significant way, and at the price of great architectural complexity. So CPU manufacturer turned to multi-core architectures. The number of transistors still double roughly every 18 months, doubling the number of *cores*, while the complexity per core remains roughly the same. This lead to the start of the so called *concurrency revolution*: to obtain performance increments on new processors, software needs to be changed and become parallel.

  Parallel simulation of biological systems must be addressed, if we want to deliver the

promise done by systems biology to be able to understand a system as whole, where models could possibly be very large in order to capture the systems behaviour in a reliable way. Today models are at the scale of a single pathway; on a single CPU or core a sequential simulation algorithm can take hours to days to run the simulation of a complex pathway with hundreds of species and millions of entities. Consider now a (simple) cell in which tens of pathways can be active simultaneously, and the problem becomes clear.

The contribution of this thesis is an analysis of current simulation methods, how they can be revisited in terms of known parallel architectures and some practical considerations of the limits to the extent of parallelism that can be obtained on current architectures.

- **Scale at result interpretation level**: even today the simulation of a single pathway can produce a very complex output. Output generated from models expressed in a powerful language is particularly difficult to examine. This happens because the more a language is able to encapsulate and hide complexity from the modeller, i.e. the more powerful a language is, the more complex the model can become, and this is shown in the output of an analysis or simulation. Thousands of entities, intermediate configurations, complexes, can be produced during a simulation. Clearly, understanding what is going on and how they are produced, how they interact, ultimately why a particular behaviour is expressed can be daunting - but it is the reason of why the simulation was run and the model created in the first instance, so it must be addressed.

  The contribution of my thesis in this area is an analysis of the visualization techniques that can be applied to the simulation results to help the scientist to extract knowledge from the raw data. Furthermore, we present some ongoing work done with Larcher [134] on the use of classification combined with visualization to provide better results.

- **Scale at *experiment composition* level**: one of the goals of Systems Biology is to re-create an artificial laboratory in a virtual computing environment, creating *in-silico* experiments that can help biologists reproduce experiments that are too long or too expensive in a real lab. Sometimes, like in the case of experiments to test evolution theories, there are experiments that are not even possible in a real laboratory, and that must entirely simulated on a computer. Life scientists could use Systems Biology tools to reproduce in a virtual environment some experiments, test their hypothesis, and return to the lab with confirmations or new ideas on what they need to investigate to obtain answers to the problem at hand. In-silico experiments

are disparate and may different significantly one from another; model construction and simulation remain at the base of these in-silico experiments, but in many cases there is much more than simulation. Up to now, the design and execution of an in-silico experiment was almost completely deferred to the user. The contribution of this thesis is the design and first implementation of a framework to design, compose, run and reproduce in-silico experiments that require multiple tools and multiple steps to complete.

1.2.1 What is not part of this thesis

Many other areas of research deal with the problem of studying complex biological systems, in particular how to tackle problems of greater dimension and how to create more complex and elaborated in-silico experiments.

For example, there are whole conferences and journals dealing with the field of study of multi-scale and multi-level simulation [138, 67]. In particular, groups at the University of Rostock [226, 227] and at the University of Santa Barbara [27, 168] work on toolkits that allow for a semi-automatic choice of the algorithm to use for a simulation, considering and making trade-offs in both speed and accuracy. Similarly, Takahashi et al. [222] devised a method for multi-scale simulation that allows for whole cell simulations.

Another area of research that will not be included in this thesis is the usage and scaling of *formal analysis* techniques. A wide range of methods are grouped under this is a very large umbrella -basically, all computational analyses but simulation-. These very useful techniques, for which thorough research has been done in the area of performance and parallel computing, include markovian analyses and equivalences [13], performance and steady state analysis [129], model checking [43] (especially its stochastic variant [131]) and abstract interpretation [70].

Finally, this thesis will not face a very important aspect, which to the best of my knowledge has not yet been considered by the various systems biology workbenches and toolkits: versioning and reproducibility.

Versioning is the process of assigning version numbers to unique states of computer software. Also called *revision control*, this process is used to keep track of incrementally different versions of electronic information, typically source code, documents or other human-generated files.

Reproducibility and *documentation* are two of the basic ingredients of the scientific method. Both require to document, archive and share all data and methodology resulted from experiments, so they can be used to reproduce the experiment and made available for careful scrutiny by other scientists, allowing them the opportunity to verify results. This practice, called full disclosure, also allows statistical measures of the reliability of

these data to be established.

These two features are related, as they both need to keep track and store information about models, experiments, steps and parameters used for simulation and analyses. The systems biology community has put a good effort in the definition of standards dictating which information has to be recorded, that led to the creations of the MIASE (Minimum Information About a Simulation Experiment) standard with its formal representation SED-ML, and of the MIRIAM standard (Minimum Information Required in the Annotation of Models). However, software implementations of these standards are still scarce.

On the other hand, tools for versioning, tracking, integration and testing (with reproducibility, automated execution and reporting) are commonly found in the modern software developer toolbox[1].

The integration of ideas and functionalities from these software – in particular with respect to *versioning of models and results* and automatic *recording of changes, parameters and steps* of in-silico experiments – within the framework presented in Chapter 6 would be another step in the direction of seamless management of complex in-silico experiments.

1.3 Organization

This thesis starts with a general background and state of the art in the area of simulation and modelling formalism. The problems to tackle in order to scale up systems biology are disparate, even if they have multiple points of connection. In chapters from 3 to 6 each problem will be introduced separately, presenting for each of them a brief discussion of the state of the art, related work, both the how and why of the choices made, a small example and future research directions.

Before some conclusion remarks, chapter 7 presents a more detailed case study in which the techniques developed for this thesis where used to tackle a real size problem.

[1] As an example, consider source code repositories like CVS, SVN and Perforce; bug, features and change trakers like Bugzilla and Visual Studio TFS; unit test frameworks; etc.

Chapter 2

Background

This Chapter presents some background information on the basic concepts used in this thesis; in particular, I will present an expanded overview of Systems Biology, how it relates to simulation and stochasticity, and how the computational approach to Systems Biology opened new possibilities and new roads.

2.1 Why Systems Biology

The interdisciplinary field of study called Systems Biology was introduced in Section 1.1.2, at the beginning of this thesis; in that Section, we gave an overview of Systems Biology, mainly defining it as a *systems* science, highlighting the differences with other, more traditional disciplines in the life science domain of study.

Many distinguished scientist tried already to give a complete and precise definition of Systems Biology [127, 126, 114]; while almost all agree in the importance of defining it as a systems science, the precise definitions varies. This is another consequence of the interdisciplinarity of systems biology: people coming from different fields, with different backgrounds, inevitably focus on different subjects.

Here, following [184], we will introduce some central aspects of Systems Biology with a metaphor. When on vacation in some nice location, tourists like to take pictures, so that they can look at them later and remember the places they visited. Unfortunately, many times pictures are disappointing, because they are not able to convey the idea of the space. Especially with building, squares and landscapes it is difficult, if not impossible, to capture the image you have in mind and fix it on the film (or memory card). For example, look at a picture I took the last time I was in Firenze (Figure 2.1). The picture shows a particular of Palazzo Vecchio, with the statue of Michelangelo's David. The picture shows you a lot of particulars: you can admire the statue, with its wonderful proportions

and the smooth marble surface[1]; you can see the medieval palace, the door through which many rich and powerful rulers walked, how the door is shaped and the quality of the stone used for the building. However, despite all these details, you cannot admire the palace, or even more the square and the surrounding building with their statues; you cannot see where the palace and the famous Galleria degli Uffizi are, or were the Fiorentini used to burn witches and heretics in the Medieval Age.

Figure 2.1: A particular of Palazzo Vecchio in Florence

Surely, it is possible to zoom out, and with the help of wide lenses capture the whole square, but in this case you will lose all the details. Moreover, it is not possible to capture with a single picture how all the buildings are positioned, how the network of streets and alleys is laid out, so that you can get glimpses of the Arno river and Ponte Vecchio or of the Duomo. It is not possible to understand how people flow and flock through the square. In other words, the live dynamics of the square are not captured in a picture, no matter how detailed it is.

A video could help in this case, especially if a good director shoots long enough footage.

[1]To be precise, that statue is nowadays only a copy, as the original is located in the Galleria dell'Accademia museum.

But what if we only have a camera? Computer science and technology could come to the rescue.

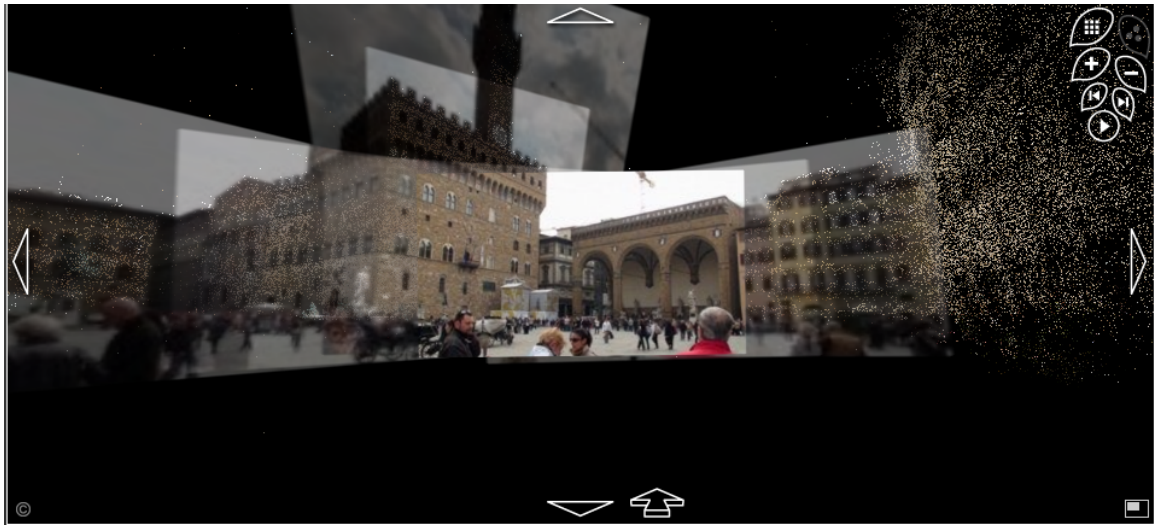


Figure 2.2: A snapshot taken during the navigation of the Palazzo Vecchio picture set in Photosynth

In recent years, advances in image processing research made it possible to automatically extract features, shapes and depth cues from images, so that a bunch of photos can be composed into one, harmonic 3D environment, using little guidance from the user. Microsoft Photosynth[2] (Figure 2.2) is one of these software. The static shoot in Fig. 2.2 cannot show how Photosynth composes static photos to recreate a 3D environment: it is possible to zoom in, switching form picture to picture in a smooth way, so that both the general picture and the details are preserved. Moreover, it is possible to relate together pictures that had nothing in common -for example, picture taken at the opposite sides of the square- and navigate through the environment, recreating the dynamics of a video.

The goal of systems biology is roughly the same: starting from detailed but static pictures, systems biology aims at composing them together, adding knowledge about the system to reveal dynamic properties and behaviour that was previously unknown or only hypothesized.

The 'pictures' in the analogy are the result *molecular biology*, the branch of science that studies biology on a molecular level, dealing with the formation, structure, and function of the macromolecules that constitute the basis of life, like DNA, RNA and proteins.

In the last sixty years, thanks to breakthrough discoveries and constant advances in technologies and experimental techniques, this field of research made huge progress, rising to a summit in the last years of the past century with the complete sequencing of the

[2] `http://photosynth.net/`

human genome.

Discoveries and advances in molecular biology led to a massive increase in the data available to scientists; the sheer amount of data available begun very soon to be beyond what it was possible to deal without information techniques, giving birth to *bioinformatics*, the application of informatics to store, manage and analyse molecular biology data.

Bioinformatics allowed for great improvements in handling of data, which lead to the success of the Human Genome Project and to the beginning of the post-genomic era. However, molecular biology and bioinformatics could not help us to understand the overall behaviour of a system; molecular biology shed light and analysed in detail how individual components (e.g. molecules and proteins) works, and bioinformatics helped in enabling these analyses at a genome scale, but this is not sufficient. As we claimed in Sec. 1.1.2, *systems* sciences need to go beyond the assumption that knowledge, however deep, of the single components could be used alone to explain the whole system, even in the extremely unlikely case of having a complete network of connections between the components.

Like in our analogy, while we continue to need to understand genes, proteins, mechanisms and structures at a molecular level, we need also the focus on the system's structure and *dynamics*; we need to shift from a single picture to a *network* of pictures, an environment that is much more than a simple catalogue or album. Quoting Kitano: "a system is not just an assembly of genes and proteins, its properties cannot be fully understood merely by drawing diagrams of their interconnections" [128]. Like in the case of Photosynth, we aim at reconstruct the system using tools and concepts from computer science.

2.2 Approaches: Stochasticity and Space

The first models of biochemical interactions pre-date the birth of computers; these models described interactions between the various entities in a *deterministic* way, following the *law of mass action*. This law assumes that the system which has to be simulated is homogeneous and that biochemical reactions are *continuous* and *deterministic*; also, it states that under these assumptions the reaction rate of any reaction is proportional to the quantities (or, better, concentrations) of its reactants.

Using this law it is possible to find expressions for the rate of change in concentrations of all the molecules of the system; hence any biochemical system can be expressed through a set of coupled, non-linear, first order differential equations. In general, these equations do not have an analytical solution, but they can be numerically integrated in order to find an approximation of the reaction dynamics of the system.

While in the past this approach was often used to model biochemical systems, in the last decades some major drawbacks of this method have emerged from a better under-

standing of physical phenomena. In fact it is now known that molecules undergo random collisions, some of which can trigger reactions if the released energy exceeds a certain threshold. Moreover, the dynamic evolution of biochemical systems is not a continuous process, because the quantities of the molecules can change only by integer amounts, that is, discrete steps.

Because of this intrinsic stochastic and discrete nature of biochemical reactions, the physical base of the deterministic model is not completely correct. In some cases this approximation is acceptable, whereas it becomes more and more unacceptable when systems are composed of a small number of molecules. In these systems, in fact, the deterministic approach is unable to describe the fluctuations in the quantities of the molecules: in these cases a stochastic modelling approach is required.

2.2.1 Monte Carlo methods and Monte Carlo simulation

The expression *Monte Carlo method* is very general. Monte Carlo (MC) methods are stochastic techniques; they are based on the use of random numbers, probability and statistics to investigate problems [123]. MC methods are used in everything from economics to physics and chemistry to traffic flow regulation. Of course the way in which they are applied varies from field to field; dozens of different MC methods are in use even within the same discipline, like chemistry. Loosely speaking, all you need to do to call a procedure a Monte Carlo experiment is to use random numbers to examine that particular problem.

The use of MC methods allows us to examine more complex systems: large-scale problems very often take the form of linear or non-linear algebraic, differential, and/or integral equations, which are not directly solvable (within a reasonable time). When the number of variables and parameters grows beyond a few hundreds, the computational difficulties become major obstacles and the usual methods begin to fail. It is at this point that Monte Carlo methods emerge as important problem-solving tools. Consider for example the field of biochemical kinetics -the field we are interested in-: even under the conditions in which the deterministic approach is valid (large number of molecules in a homogeneous system) and therefore a system can be represented in the form of a system of differential equations, often this system does not have an analytical solution. Even trying to solve it numerically is feasible only for simple sets of equations.

The Monte Carlo method in the case of linear algebraic problems usually consists in creating a suitable statistical situation (a probability space); the solution is obtained by computing an estimate obtained from a *random walk* in that space. This method is called *Random-walk Monte Carlo*. At every step in this random walk the path to follow is chosen raising a question and obtaining an answer in the form of the expected value of

some random variable. This is typically done using a computer model that makes use of so-called *pseudo-random* generators.

The Random-walk Monte Carlo method is widely used for both its efficiency compared to deterministic algorithms and its physically correct behaviour. It is worth noting that at many levels, nature exhibits a chaotic behaviour: besides quantum physics considerations, even if at microscopic level we could observe a deterministic behaviour, at a macroscopic level the number of variables and the complexity of the interaction, plus the inevitable influence from the environment, lead to an unpredictable evolution of the system.

2.2.2 The stochastic method

In fact, the *stochastic approach* to chemical kinetics has a stronger physical base compared to deterministic approach: early experimental studies (see as an example [213], [198]) have demonstrated that stochastic effects can be significant in cellular reactions.

More recent experimental studies showed the importance of noise in gene regulation: see [112], [152] and [66], just to cite a few ones.

> *The proliferation of both noise and noise reduction systems is a hallmark of organismal evolution* – Federoff et al.(2002)

> *Transcription in higher eukaryotes occurs with a relatively low frequency in biologic time and is regulated in a probabilistic manner* – Hume (2000)

> *Gene regulation is a noisy business* – Mcadams et al. (1999)

These studies, together with the success of Monte Carlo stochastic simulation techniques in the quantum physics simulation, have ignited widespread interest in stochastic simulation techniques for biochemical networks.

2.2.3 Representation of Space

In addition to stochasticity, spatial effects may also deeply influence many biological systems; just to mention a few, gene expression, mRNA movement and localization within the cytoplasm, morphogen gradients, protein transport; they all strongly rely on location and space. Moreover, diffusive effects are important also in the description of many pathways, including signalling pathways where sub-cellular compartmentalization and crowding can cause the signal weakening, or where co-location of molecules is fundamental for the pathway dynamics.

In [220], Takahashi et al. review the various approaches to spatial simulation. Spatial simulation methods can be classified using different criteria: whether and how they deal

Figure 2.3: The different levels of abstraction discussed in this chapter

with *spatial effects*, the *scale*, i.e. the granularity, at which they operate, the abstraction used for *time*, whether they are *stochastic* or not, and if they consider *weak interactions* or not.

Method	Space	Scale	Time	Stochastic
MD	Particle	Micro	Continuous	-
BD	Particle	Micro	Varies	+
CA	Discrete	Micro/Meso	Discrete (steps)	Varies
Spatial Gillespie	Discrete	Meso	Events	+
PDEs	Mesh	Macro	Continuous	Varies
Gillespie	Homogeneous	Meso	Events	+
ODEs	Homogeneous	Macro	Continuous	-

Table 2.1: (Spatial) simulation methods. The table is an adapted version from [220]

In Table 2.1 the various methods for spatial simulation are listed; each method in the list groups several different algorithms in a family; for example, BD encloses all the particle based approach at the Smoluchowski level of detail. The methods are listed in increasing level of detail, represented by the *Space* column.

Space is the spatial abstraction and level of detail used by the method (see Figure 2.3); in the 'particle' space, molecules are represented as individual particles with positions in a continuum space; in 'discrete' space, space is discretized in sub-volumes (or voxels), or a regular lattice is used. In the latter case, each lattice site can accommodate one or multiple particles. Usually, when only particle is allowed per lattice site, the method is still operating at the microscopic scale, while allowing multiple particles per site/per voxel bring the method into the so called *mesoscopic* scale, a scale at which the representation of particles is still discrete, but there is no need and no way to distinguish and treat individual particles. In other words, at the mesoscopic level there is the shift from individual molecules to populations, or species.

Time represent how time is treated. The time evolution in PDEs, ODEs and MD systems is obtained by integrating over time, considered as a *continuous* variable. Time is stored as a continuous variable also in Gillespie and its spatial variant; however, instead of integrating over a time variable, the method "skips" from one time point to another, which represents the next simulated *event*. Finally, BD methods use different time representations: methods based on Green Functions are event-driven, while Smoldyn, for

example, updates the system state at *discrete* time steps.

Many biochemical models need a level of detail and accuracy that requires simulations to deal with both spatial and stochastic aspects (for a survey on these biochemical phenomena and the computational methods used to simulate them, see [62]). For this reason, we will summarize and present only the techniques highlighted in bold in Table 2.1.

2.2.4 Gillespie SSA

The Gillespie SSA, and the family of algorithms derived from it, can be considered the *de-facto* standard simulation algorithms for systems biology. This algorithm, as displayed in Table 2.1, does not take into account space, operates at the *mesoscopic* scale -where quantities are discrete, but there is no single molecule details and hence the algorithm works at a species or population level- and it one of the first stochastic algorithms.

The stochastic approach to chemical kinetics -at a mesoscopic scale- was first employed by Delbruck in the '40s. The basic assumptions he made are that a chemical reaction occurs when two (or more) molecules of the right type collide in an appropriate way, and that these collisions are *random*. Whenever two molecules come within a certain proximity, they can react with some probability: collisions are frequent, but those with the proper orientation and energy, that is the collisions that allow molecules to react together, are infrequent. In [81] and [84], Gillespie introduced the additional assumption that the system is in thermal equilibrium. This assumption means that it is possible to avoid the difficulties generated by the procedure of estimating the collision volume for each particle; the system is considered as a well-stirred mixture of molecules, where the number of non-reactive collisions is much higher than the number of chemical reactions. It makes possible to state that the molecules are randomly and uniformly distributed at all times, and therefore that the mixture is *homogeneous*.

This *well-stirred mixture* assumption allow to derive an exact stochastic method, which is in charge of predicting collisions by estimating the collision volume of each particle, that is computationally lighter than other methods. This method, derived by Gillespie, is called the *Stochastic Simulation Algorithm* (SSA) [84].

We can observe that biological systems can be modelled on different levels of abstraction, but models at each level follow the same pattern:

- pairs *entity type, quantity*;
- interactions between the entities.

For example, in the case of biochemical models *entities* are molecules and *interactions* are coupled chemical reactions. Therefore, we can reduce the necessary parameters for describing a system to:

- the *entities*, usually referred to as *species*, present in the system $S_1, ..., S_N$;
- the number and type of *interactions*, called *reaction channels*, through which the molecules interact $R_1, ..., R_M$;
- the state vector $\mathbf{X}(t)$ of the system at time t, where $X_i(t)$ is the number of molecules of species S_i present at time t.

The state vector $\mathbf{X}(t)$ is a vector of random variables that does not permit to track the position and velocity of the single molecules.

2.2.5 Base rate and actual rate

For each reaction channel R_j a function a_j, called the *propensity function* for R_j, is defined as:

$$a_\mu = h_\mu c_\mu \text{ for } \mu = 1, \ldots, M \tag{2.1}$$

such that h_μ is the number of distinct reactant combinations for reaction R_μ and c_μ is a constant depending on physical properties of the reactants and

$$a_0 = \sum_{\mu=1}^{M} a_\mu$$

The c_μ constant is usually called *base rate*, or simply *rate* of an action, while the value of the function a_μ is called the *actual rate*.

Gillespie derives a physical correct *Chemical Master Equation* (CME) from the above representation of biochemical interactions. Intuitively, this equation shows the stochastic evolution of the system over time, which is indeed a Markov process.

Gillespie also presented in [84] an exact procedure, called *exact stochastic simulation*, to numerically simulate the stochastic time evolution of a biochemical system, thus generating one single trajectory. The procedure is based on the *reaction probability density function* $P(\tau, \mu)$, which specifies the probability that the next reaction is an R_μ reaction and that it occurs at time τ. The analytical expression for $P(\tau, \mu)$ is:

$$P(\tau, \mu) = \begin{cases} a_\mu \exp(-a_0 \tau) & \text{if } 0 \leq \tau < \infty \text{ and } \mu = 1, \ldots, M \\ 0 & \text{otherwise} \end{cases}$$

where a_μ is the *propensity function*.

The *reaction probability density function* is used in a stochastic framework to compute the probability of an action to occur. The way of computing the combinations h_μ, and consequently the *actual rate* a_μ, varies with the different kind of reactions we consider.

In the case of first-order reactions, h_μ is equal to the number of entities (the *cardinality*) of the one reactant, while in the case of second-order reactions, h_μ corresponds to the number of all possible interactions that can take place among the reactants.

2.2.6 Molecular Dynamics

Chemical and biochemical reactions can be simulated in a very precise and detailed way using molecular dynamics. Molecular dynamics is a form of computer simulation where atoms are allowed to interact under known laws of physics, giving a view of the motion of the atoms. Methods that simulate quantum mechanical and molecular mechanical dynamics have been applied to a wide range of problems of chemical and biological interest (see for example [92]), such as chemical reactions in solution and enzymes and solvent effects on electronic excited states; in these simulations, every detail of the chemical reaction, like formation and breaking of bonds between single atoms, and the position and energy of every atom in the system are explicitly simulated.

2.2.7 Brownian dynamics

Brownian dynamics (BD) methods operate at a slightly coarser level of detail, where molecules have an identity and an exact position in continuous space, but no volume, shape or inertia. Every molecule of interest is represented as an individual point, and those that are not of interest (water, non-reactive molecules, etc.) are not represented. Brownian Dynamics simulations are a stochastic simulation approach with continuous space; they are based on the solution of the Smoluchowski equation, which describes the diffusive encounter of molecules in solution.

Handling of time varies from method to method: some methods are realized as a numerical procedure to solve the Smoluchowski equation; in this case, time is continuous. Others methods, like GFRD (Green's function reaction dynamics) [228] and E-GFRD (Enhanced GFRD) [221], are based on the decomposition of the problem into a set of two-body problems and on the analytical solution for these two-body problems of the Smoluchowski equation by using Green's function; in this case, the simulation time is driven by events.

Finally, Smoldyn [8] is another approach to the numerical realization of the Smoluchowski model. In Smoldyn, a bimolecular reaction occur if two reactants approach each other within a binding radius, a radius that is different (typically smaller) than the physical radius of the molecules, and that depends on the diffusion coefficients and on the reaction rate constant. Simulated time is discrete, as reactions, computation of movements and update of the position are done at fixed time steps.

2.2.8 Lattice-based methods

At a slightly coarse level of detail we find lattice-based methods. In this case, the simulated space is divided into voxels (three dimensional elements); each voxel is assigned to a lattice site. In particular, we will focus on cellular automata (CA) based methods. These methods share two characteristics: space and time are discrete, and the evolution in time of the system is fully specified in terms of local interactions, instead of being obtained by solving for the global behaviour of a phenomenon.

In a standard cellular automaton each site has a finite number of states; the molecule can propagate from one site to another according to its diffusion rate, and then collide or react with other molecules. CA can be used to simulate reaction and diffusion at both microscopic scales, having single and multiple molecules at a site, respectively. In the latter case, the CA is called a *multiparticle model*.

The multiparticle diffusion model is more complex and more realistic. In this model, multiple particles per lattice site are permitted; particles move in a stochastic way by following independent random walks between positions in the lattice. Brownian diffusion is therefore modelled as a series of independent random choices for the movement of particles on a regular, uniform grid.

A third possibility is to use a coupled map lattices (CML). A CML is an extension of a CA where the discrete state values of CA cells are replaced by continuous real values. CML are a versatile technique for modelling a wide variety of dynamic systems and phenomena, including chemical reaction-diffusion systems, as the Gray-Scott model -described in [165]- or Turing pattern models.

2.2.9 Spatial Gillespie

The Gillespie algorithm, introduced in Sec. 2.2.4, allows simulating chemical reactions in an efficient way. Every collision that leads to a reaction is explicitly simulated, but collisions that do not lead to a reaction are not. The stochastic behaviour of the chemical system is preserved, as molecules are still represented as discrete quantities, but information on a single molecule, and with it any positional information, are lost. Moreover, the assumptions made by Gillespie explicitly rule out diffusion from the system: since the solution is in thermal equilibrium, it is assumed that diffusion is instantaneous so that each molecule has the same probability of reacting with every other molecule in the system. The algorithm works well locally, but cannot be used to represent complex pathways that span over a considerable extension of reactions taking place in an inhomogeneous medium.

A proposed extension is the *discretization* of the space by subdivision into logical

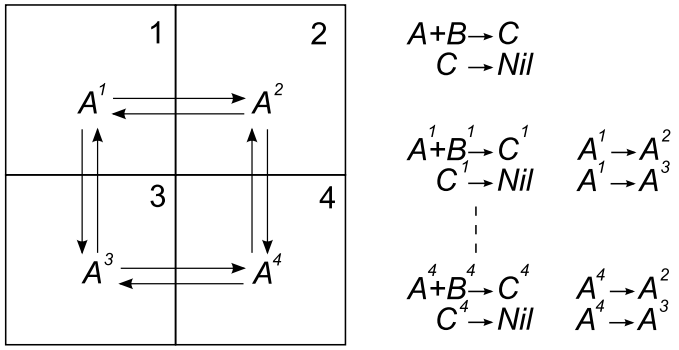


Figure 2.4: The extension to the Gillespie SSA proposed by Bernstein. On the left, a discretization of the space into four cells. On the right, the species and the reactions added to the system in order to deal with diffusion.

sub-volumes, often referred to as *Cells*. The dimension of a cell is chosen to be small enough for the sub-space to be homogeneous and for the enclosed entities to have almost instantaneous diffusion, so that the assumptions made by the Gillespie algorithm are valid inside a single cell; furthermore, spatial information is added to the system by duplicating every species S. New species with the same characteristics of S and with an index identifying its position on the grid are added to the system $(S_1, S_2, ..., S_n)$; diffusion is represented by first-order reactions among species. This method, proposed by Bernstein [14], is depicted in Fig. 2.4.

The advantage of this approach is that the algorithm in charge of simulating the reaction-diffusion system does not change; it is possible to add more species to model the molecules in different compartments and add reactions to "diffuse" between adjacent compartments, and then to use the existing tools and algorithms to simulate the modified system.

An efficient implementation for simulating reactive-diffusive systems by using spatial structures is used in the *next subvolume method* (Elf et al. [65]). The underlying theory is the same utilized by Bernstein, as both are based on the exact realizations of the Markov process described by the Reaction Diffusion Master Equation. The algorithm uses three data structures: (i) a *connectivity matrix*, (ii) an *event queue* and (iii) a *configuration matrix*, used to naturally partition reactions into sub-volumes. Instead of mapping movements of entities using different species, the *direct method* [84] is used on each sub-volume to compute the time for the next event, i.e. a chemical reaction or a diffusion event. Then, the *next reaction method* [80] is used to identify the sub-volume where the first event will occur. The event is simulated, then the reaction and diffusion times in the volume (or volumes, in case of a diffusion event) are updated using the *direct method* again.

2.3 Model definition

As we have seen in Section 1.1.2, Systems Biology put the focus on understanding the system's structure and dynamics; as a consequence, scientists needed a way to describe how a biological system behaves, how it will respond to stimuli, how it will dynamically evolve in time, transitioning from an initial state to a final one. In other words, Systems Biology leads to a *hypothesis-driven* kind of research [128], where *models* play a central role.

A biologist that would like to understand how a systems behaves, starts by making an educated guess of how it may work internally: based on the data and facts at hand, he builds into his (or her) head a model of the system. Then, he proceeds to verify that model, in order to validate or reject it, in the spirit of the experimental scientific process.

Generally speaking, a model could be expressed in many ways: form drawings on a blackboard to the cartoon-like figures so common in many textbooks and databases (see Figure 2.5), to a plain English description of the dynamics, to a set of equations describing the concentration of chemicals and molecules in time.

Soon, the problem of which formalism to use to create model arose. For example, free form descriptions (textual or graphical) have several disadvantages: they are not really precise -the scientist can make several assumptions that he or she may later forget-, it is difficult to share -again, assumptions and symbols that may be clear to a scientist may not be for another one- and does not give the ability to perform direct tests and validate or invalidate the model in a fast, reproducible way.

These shortcomings are all due to one reason: the lack of formality, or in other words, of a precise semantics.

The first biochemical models that tried to overcome these limitations were described by using mathematical tools, mainly PDEs (Partial Differential Equations) and ODEs (Ordinary Differential Equations). As seen previously in this Chapter, these sets of equations are used to build *deterministic* models of the system under study. Mathematical models are not ambiguous, as every observable aspect of the system is described using the precise and rigorous language of mathematics.

However, mathematical models present some limitations when they are used to express system's dynamics. First of all, composability. As shown in [30], a biological network represented as a set of differential equations consists of a large "flat" set of equations that unrolls the state space, where both the network structure and the discrete character of the components are lost.

Secondly, on a related note, a model based on a large flat systems of equation may lead not only to the loss of structure, but to models that have little or no *predictive*

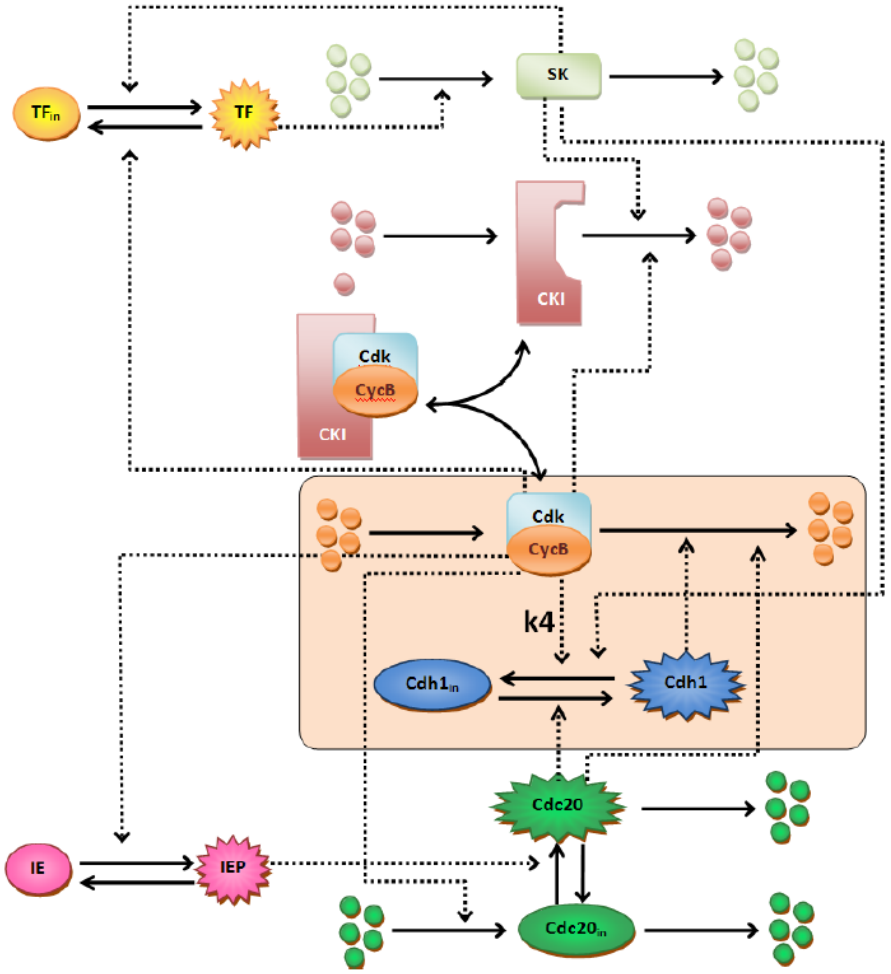


Figure 2.5: Representation of a biological process in present-day databases and textbooks; image taken from [164]

$$\begin{cases} \frac{d[S]}{dt} = -k_1[E][S] + k_{-1}[ES] \\ \frac{d[E]}{dt} = -k_1[E][S] + k_{-1}[ES] + k_2[ES] \\ \frac{d[ES]}{dt} = k_1[E][S] - k_{-1}[ES] - k_2[ES] \\ \frac{d[P]}{dt} = k_2[ES] \end{cases}$$

Figure 2.6: Representation of a biological process as a set of differential equations: the structure is flat and modularity is lost

$$\begin{cases} \frac{d[S]}{dt} = -k_1[E][S] + k_{-1}[ES] \\ \frac{d[E]}{dt} = -k_1[E][S] + k_{-1}[ES] + k_2[ES] - \mathbf{k_{fi}}[\mathbf{E}][\mathbf{I}] + \mathbf{k_{ri}}[\mathbf{EI}] \\ \frac{d[ES]}{dt} = k_1[E][S] - k_{-1}[ES] - k_2[ES] \\ \frac{d[P]}{dt} = k_2[ES] \\ \frac{\mathbf{d[EI]}}{\mathbf{dt}} = \mathbf{k_{fi}}[\mathbf{I}][\mathbf{E}] - \mathbf{k_{ri}}[\mathbf{EI}] \\ \frac{\mathbf{d[I]}}{\mathbf{dt}} = -\mathbf{k_{fi}}[\mathbf{E}][\mathbf{I}] + \mathbf{k_{ri}}[\mathbf{EI}] \end{cases}$$

Figure 2.7: Loss of modularity is clear when we modify kinetics in presence of an inhibitor: it is necessary to modify the equation for E to obtain the correct behaviour

power; in other words, models that are limited to describe the *observed* behaviour of the system, opposed to models that give insight on the underlying mechanics that produce that observed behaviour. As we have seen in Section 1.1.2, Systems Biology needs to do more than sticking together the description of single components to understand how components interact as systems to produce the observed behaviours. On the other hand, however, this does not mean that it should embrace an entirely *holistic* approach, relying only on high level descriptions of the system, but build on the reductionist approach and overcome its limitations.

For example, consider a signalling pathway where it is *observed* that a protein acts like a switch, by making the concentration of an entity grow as a hill function when it is active. A common way of modelling it with a system of equations would be to model this growth by using a hill function. The model will reproduce the exact observed behaviour; however this does not imply that we understood why, in the first place, the protein acts as a switch [108]. Again, it is possible to *observe* that an entity has a negative feedback on another one: when the concentration of first one grows, the concentration of the latter diminish. Using equations, biologists may be tempted to introduce a direct dependency between the two entities, because this is what he observed, even if the two entities never

come to direct contact in the real biological system. Of course, modelling of higher level behaviour it is sometime necessary; also, it is obvious that this is not a shortcoming of mathematics per-se, but it is partly consequence of the lack of composability, of the ability of building a model starting from parts, and partly due to its denotational nature. In Jasmin Fisher words "a mathematical model is a formal model whose primary semantics is denotational; that is, the model describes by equations a relationship between quantities and how they change over time. The equations do not determine an algorithm for solving them [...] for mathematical models, there is a gap between the meaning of the model and its implementation on a computer" [75].

Finally, stochasticity. It is indeed possible to include noise and stochasticity in PDE systems, in the form of the Langevin equation [82] (for a complete outlook on how different deterministic and stochastic methods are related, and how to translate between different methods, see [83] and [25, 30]). However, trying to preserve stochasticity may lead to odd equations, where entities that only exist in discrete quantity are related in differential form. Moreover, established stochastic methods acting on *discrete* quantities have the advantage of being proven to be exact and of being generally applicable, with no need to demonstrate the physical soundness of every model, and simpler.

This transition from deterministic to stochastic methods marked the transition from *solvable* models to *runnable* or *executable* models.

2.4 Computational and Executable Systems Biology

As soon as computers were available, mathematical modellers started to take advantage of their power to study mathematical models. However, these models were *solved*, using a choice of algorithms to analyse the mathematical relationship between elements. In the '70 a different variety of models, called *computational models*, started to appear; in particular, *boolean networks*, first introduced by Kauffman [125, 85].

A computational model resembles a computer program. Like a mathematical model, a computational model is a *formal*. The difference lies in its semantics; a computational model dictates a sequence of steps or instructions that can be executed by an abstract machine. This means that the primary semantics of the model is *operational*, and can be therefore implemented on a real computer. To study a mathematical model, an algorithm must be devised; a computational model instead is inherently *executable*, as it describes which steps must be taken by its abstract machine to reproduce its behaviour. This abstract machine can be implemented on a computer, and therefore the model can be *simulated* by running it inside an implementation of the abstract machine that follows one of the stochastic algorithms just introduced. This approach of executing biological

processes is at the basis of emergent schools of thought like *executable biology* [75] or *algorithmic systems biology* [184].

Computational models give various advantages: for example, when their execution engine implements one of the stochastic selection algorithms listed in the previous sections, their simulation is by-construction correct w.r.t. physical characteristics; depending on the underlying formalisms, a number of computer science analysis techniques (model checking, behavioural equivalence checking, Markovian analysis, ...) can be applied to them; finally, they are easier to compose, especially when they are written in an appropriate way.

2.4.1 Which language?

A consequence of this shift towards computational models was the need for a new *language*; mathematical models are naturally expressed using mathematics, but what about computational models? Which is the language of computation? Or, even more important, which is the language for biological computation?

A first, simple choice, initially adopted for biochemical models, is the use of *chemical equations*, a set of chemical formulas extended to represent biological reactions and interactions. Every reaction in a system is therefore represented as one or more chemical formulae:

$$E + S \rightarrow ES \tag{2.2}$$

$$ES \rightarrow E + S \tag{2.3}$$

$$ES \rightarrow EP \tag{2.4}$$

$$EP \rightarrow E + P \tag{2.5}$$

This approach proved to be very popular: it is very simple, looks familiar to biologists -chemistry belongs to their background- and fits nicely with stochastic methods; it is only necessary to label each reaction with a stochastic rate, and the method will take care of computing propensities for that reaction. Furthermore, this approach is very common: it is sufficient to consider that SBML, the Systems Biology Markup Language, uses this formalism at his core [110].

Chemical equations are a step forward in some aspects: for example, besides the aforementioned support for stochastic method, they are better than differential equations with respect to composability. In fact, chemical equations express naturally sets of parallel reactions; therefore, two independent pathways can be simulated together simply by stitching their models. However, if the two models need to interoperate, things get more complicated and more work is required. This problem arises from the *lack of formal-*

ity and from the lack of information about the *behaviour* of molecules and interactions. Furthermore, this approach makes nothing to help the modeller in dealing with the complexity inherent in biological systems[3]; the chemical notation is not compact. Again, this is due to the fact that instead of *describing* the behaviour of molecules and biological entities, this approach is limited to *listing* the possible behaviours. It makes it easy to add new behaviour, but leads to duplications and verbosity. As every programmer knows, duplications affect in a terrible way extensibility and maintainability.

To draw a parallel with computer science, think about trigonometric functions, like sine and cosine. It is possible to implement them with a lookup table, listing values of sine and cosine for various inputs. Or it is possible to use an *algorithm*, a method to compute values for sine and cosine for any possible input, like the power series developed by Newton. In this way, the function is compact, its precision higher (arbitrarily higher), and works for many different inputs. However, it requires a computer to be convenient to use.

As an answer to these needs, *formal languages* for systems biology began to flourish. Using a formal language to model biological interactions and dynamics allow us to write compact models, and let us build in a relatively simple way very complex models. Moreover, it is easy to see how models built using formal languages are inherently *executable*: formal languages have semantics, and so a computer program can consume them and, following the semantics rules, reproduce step by step their dynamic evolution in a simulation, or analyse in a rigorous way their properties using static analysis techniques.

In this way the complexity inherent in biological systems is not entirely erased, but partially transferred from the modeller to the software. The software, in fact, can make automated analysis of a model, but then the results of these analyses -the emerging dynamics of the system, important to understand how the system behaves- must be interpreted by a scientist. Understanding how a system behaves is one of the goals of modelling, and it is also important as a debugging facility to correct models with an erratic behaviour.

Many different formalisms were used as a language for describing the dynamics of biological systems; among them, the already mentioned boolean networks; Petri-nets [167], originally developed as a graphical formalism for chemical reactions; statecharts [98]; process algebras. We consider here process algebras, because their ability of handling large systems, concurrency, causality, nondeterminism, stochasticity and cooperation/competition for resources makes them the ideal candidate to tame the complexity of biological systems.

[3]In fact, the next, not yet finalized revision of SBML, version 3, will contain new concepts that will help to overcome its now apparent shortcomings

2.4.2 Process Algebras

Process Algebras (or *Process Calculi*) are a family of formalisms created to model *concurrent systems* in a formal way. Process Algebras provide a language for the description of interactions between a collection of concurrent *processes*, and algebraic laws that allow process descriptions to be manipulated and reasoned upon in a formal way.

A variety of process calculi originated from the precursors CCS [157] and CSP [105]; however, all of them have in common several features.

First of all, the notion of *process* is central to these formalisms. The exact term used to refer to processes may vary, and they can be referred to using some synonyms, like *entities*, *agents*, *boxes* and so on. Processes are first-class constructs in these languages, and they are the single most important one, in the same way functions are the base construct in functional languages and objects in object oriented programming. In this thesis we will introduce and focus on the BlenX language, so we will follow the BlenX literature and use the term *box* or *entity* to refer to processes.

Second, entities are separated, isolated objects: their state is not shared, nor any other entity in the system can modify it directly. Instead, they interact by communication (message-passing), exchanging messages on *channels*, names whose purpose is to provide means of communication.

The third characteristic is the description of entities and systems of entities as a combination of a small number of primitives by means of basic operators. The basic operators include input and output *actions*, used to exchanged messages; sequentialization of actions and interactions; parallel composition of processes; recursion or process replication; manipulations of interaction points (e.g. creation or hiding of channels).

A last common characteristic is the definition of algebraic laws over the operators; as we mentioned, they allow process expressions to be manipulated and permit formal reasoning about processes.

A composition of operators and primitives can be used to build what is called in computer science a *communication protocol*. The behaviour of a system is given by the ordered sequence of actions and communications that a system can perform. Despite of its simplicity, process algebras contains the crucial ingredients for the description of concurrent and cooperating systems.

It is easy to see how they can be useful to describe concurrent execution in the computer science domain; for example, a web-browser requesting a page to a web-server may be modelled in process algebra as such:

```
browser := Send(HttpChannel, RequestForPage).Receive(HttpChannel, Response);
server := Receive(HttpChannel, RequestedPage).(Choiche(
ExistPage(RequestedPage) -> Send(HttpChannel, Page),
not ExistPage(RequestedPage) -> Send(HttpChannel, Error404));
```

Send and Receive are the primitive input and output actions we were talking about; they are composed here using the sequencing operator '.', meaning that the action after the dot is executed only when the action before the dot is completed, and the 'Choice' operator (often represented with a '+' sign), meaning that either one or the other action is executed, but never both. Choice looks like an *if* statement for actions.

With this in mind, we can translate the two processes (or entities, as we shall call them) in plain English. The browser entity declare to do the following: "send a request on the Internet Http channel, *then* prepare to receive a response on the same channel"; the server entity is prepared to: "receive a request form the Internet Http channel, *then either* if the page exists send the page, *or* if it does not exists send an error".

It is important to note that a message exchange happens when two entities request two complementary actions on the same channel: for example, the `Send(HttpChannel, RequestForPage)` in the browser entity matches the `Receive(HttpChannel, RequestedPage)` in the server entity.

Despite their common underlying basis, the several languages and calculi proposed in the literature have all important distinctions and different features that makes them differently expressive, powerful, and simple to use and reason upon (in a formal way too). A distinction that is particularly important for us is the synchronous or asynchronous nature of the communication primitives - the input and output actions. In some calculi, actions -in particular the output or *send* action- can be *asynchronous*, which means that they are not-synchronized with a matching action; in this case, the action is non-blocking, and the eventual remaining part of a sequence is executed immediately, without waiting for the pending communication to complete. In most of them, however, the communication primitives are *synchronous*, which means that pairs of matching actions are rendezvous points; entities are allowed to progress only when both sides of a communications are ready. We will consider only synchronous calculi, as they are more apt to describe biological interactions.

2.4.3 Process algebras and Systems Biology

Historically, the first step towards the use of process algebra for computational systems biology was the use of algebras originally meant to model concurrent interactions in software systems (like CCS [155], the pi-calculus [156], PEPA [103] or the Mobile Ambients [34]). Process algebras started to be used to model abstractions of biochemical reactions in the last years of the 20th century; between 2001 and 2002 works in this area started to flourish [190, 191, 182], mainly using the pi-calculus as a language for biology; in 2002 Regev and Shapiro wrote a seminal paper [189] that cast the ground for the description of biological processes using concurrency theory and process algebras (see Figure 2.8).

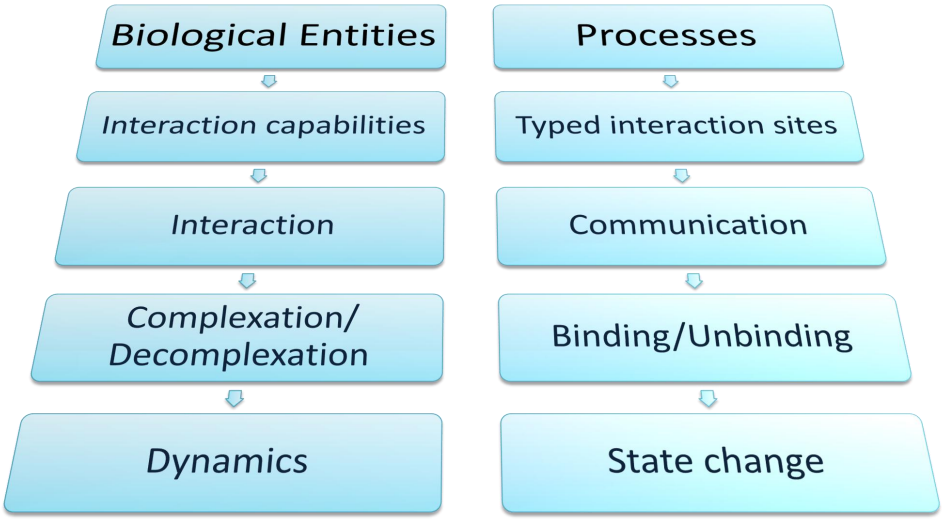


Figure 2.8: The parallel drawn between biology and concurrent and distributed computing. Cells and biological processes can be seen as computations [189]

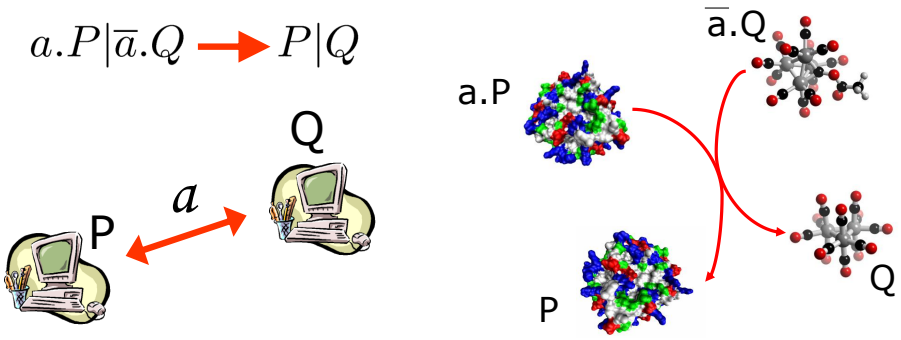


Figure 2.9: Languages of communicating interacting processes, used for modelling concurrent systems, can represent molecular interactions as well

Actions and co-actions, that are usually seen as input and output activities, can be the abstract view of any sort of complementarities. Actions could well correspond to the abstract view of requests sent by an operating system to a printer manager, or the conformational changes that take place in a receptor protein in response to its binding with the signal molecule (see Figure 2.9). What is crucial to notice here is that, whichever is the level of abstraction considered, by its own nature process algebras describe a system in terms of what its subcomponents can do rather than of what they are.

These first attempts encouraged the invention of new "biologically inspired" process algebras, designed to model in a better way some of the features specific to the biological domain [93, 47]; in particular, the experience in modelling biological systems in pi-calculus and Mobile Ambients lead to the design and creation of BioAmbients [187], a calculus suitable for representing various aspects of molecular localization and compartmentalization, including the movement of molecules between compartments, the dynamic rearrangement of cellular compartments, and the interaction between molecules in a compartmentalized setting.

Concepts in BioAmbients were developed to design and create the Brane Calculi [29], a family of process calculi with dynamic nested membranes, where active entities here are tightly coupled to membranes. In this way, the focus shifts from molecules to membranes, since membrane are the entities performing interactions, and hence play a central role in computation.

Experience with the pi-calculus also inspired the creation of Beta-binders [180], a calculus to reason about biological interactions where constructs called *binders* are added to wrap a process. Like BioAmbients, binders resemble membranes; however, they are a fine grained concept used to mimic biological interfaces at various levels (protein domains, membrane receptors, ...). A few operators were added to the pi-calculus kernel to describe the dynamics of those interfaces; communication semantics was modified to support both encapsulation and *affinity based communication*, useful to model realistically domain-domain interactions in proteins, ligand-receptor binding and in general all shape based interactions [177, 179].

Similarly, the K-calculus [49] introduced interactions modelled at the domain level, where bonds are represented using shared names; even if K-calculus has a different granularity, it can still be encoded in pi-calculus.

Finally, experience in using both PEPA [26] and Beta-binders [42] for building models of biochemical pathways recently lead to the creation of Bio-PEPA [41], which extended PEPA with features to handle biochemical networks, such as stoichiometry and different kinds of kinetic laws.

This list of algebras and calculi is by no means complete, but includes the most relevant

ones to the development of computational and executable systems biology.

In [172], Cardelli devises four distinct chemical *toolkits*, each combinatorial in nature, each abstracting from chemistry using a different abstract machine defined by *instruction set* and its own peculiar *interaction model*. These interaction models are not ones commonly used in computing: however, as we have seen, it is possible to map them to concepts derived from concurrent and distributed computing. In particular, Cardelli devises three machines, for different levels of detail; even if the interaction model for these abstract machines is very different (e.g. fast synchronous binary interactions for Proteins, slow asynchronous stochastic broadcast for Genes), he points out that the pi-calculus, with stochastic semantics and a synchronous synchronization model, is extraordinarily suitable for describing molecular interactions at a protein level, higher levels of organization and also asynchronous genetic networks. For this reason, the synchronous communication model is considered the model of choice for biological interactions.

Modelling reactions with a process algebra

Biological *species* (cells, enzymes, molecules...) are modelled as *entities* with interaction capabilities; a biochemical interaction becomes a message exchange (a *protocol*); the modification of a species, its transformation due to internal or environmental factors, becomes a change of state in the entity, or the transition from one entity to another one, according to the language.

As an example, consider the following chemical interaction:

$$E + S \rightarrow E + P$$

This is a classical chemical interaction in which an enzyme E act as a catalyst to ease the conversion of the substrate S into the product P.

If you think of E as an entity, box or agent, it has only one purpose: every time it finds a substrate, change it into a product. But we have seen that entities are separate objects, so the only way in which E can change S is indirectly, by sending it a message:

```
E := Send(Channel, ChangeYourself).E;
```

Once a message is sent, the entity continues to act as an E, so that an enzyme it represents can be re-used and interact with another substrate. On the other side, S simply has to wait for a message. When a message arrives, if it is a *ChangeYourself* message it change into a product P, otherwise it will continue to be an S:

```
S := Receive(Channel, Message).(Choice([Message == ChangeYourself]P
                                       [Message != ChangeYourself]S);
```

2.5 Summary

This chapter introduced some basic Systems Biology concepts that will be used throughout this thesis. In particular, we introduced modelling and simulation as fundamental steps for supporting the scientific method in Systems Biology. We then focused on different simulation methods and on modelling languages, with particular emphasis on the usage of process algebras to build executable biological models.

The next chapter will elaborate on these concepts, presenting how they lead to the introduction of BlenX, a process algebra-derived language with biological constructs.

Chapter 3

BlenX

This Chapter will introduce BlenX, a language derived from the Beta-binders process algebra, specifically designed to model different biological phenomena.

In the previous chapter we have seen how process algebras and, in general, concurrency theory, revealed to be apt to describe biological systems, and how new process algebras were specifically designed to include basic biological interactions and characteristics as primitives.

Now we will explore how these concepts drove the design of BlenX, explaining the rationale of a language derived from a *biologically inspired* process algebra. This kind of language adds to the common process algebras concepts derived from biology, in order to make the modelling process easier and reduce in some way the "semantic gap" between biology and programming languages. Our goal is to illustrate not only *how* the language works but also *why* it looks and works in this way, in a tour of the design choices that were made during its creation.

In fact, although BlenX builds on concepts derived from the Beta-binders process algebra, a real world language has to face several compromises: it has to be usable by non-experts, to be efficiently interpreted and executed on a computer, it needs to cope with incomplete and mixed models, it has to provide constructs that are not present in pure process algebras but are still necessary, and so on. Compromises are necessary: similarly to a living organism, and like many other software systems, evolutionary pressure is exerted on a language. As a consequence, language design is an evolutionary process.

3.1 Building BlenX

In the previous chapter, we introduced differential equations as a possible way of modelling biological interactions. However, we observed how differential equations are not easily

composable. Consider a slightly revised version of the example in Figure 2.6:

$$\frac{d[S]}{dt} = -\, k_1[E1][S] + k_{-1}[E1.S] \tag{3.1}$$

$$\frac{d[E1]}{dt} = -\, k_1[E1][S] + k_{-1}[E1.S] + k_2[E1.S] \tag{3.2}$$

$$\frac{d[E1.S]}{dt} = k_1[E1][S] - k_{-1}[E1.S] - k_2[E1.S]$$

$$\frac{d[P]}{dt} = k_2[E1.S] \tag{3.3}$$

If the product P is used in another reaction, we do not only need to add new equations, but also to modify the existing ones. For example, suppose that P is an enzyme that can be activated by phosphorilation, and that $E1$ is its kinase (i.e. the enzyme responsible for its activation). If we wanted to add the complementary reaction (i.e. its de-activation by an $E2$ phosphatase), we would need to add a couple of differential equations – to define the change in time of $E2$ and of its bound form, $E2.P$ – but we would also need to change Eq. 3.1 and Eq. 3.3:

$$\frac{d[S]}{dt} = -\, k_1[E1][S] + k_{-1}[E1.S] + \mathbf{k_3}[\mathbf{E2.P}] \tag{3.4}$$

$$\frac{d[E1]}{dt} = -\, k_1[E1][S] + k_{-1}[E1.S] + k_2[E1.S]$$

$$\frac{d[E1.S]}{dt} = k_1[E1][S] - k_{-1}[E1.S] - k_2[E1.S][E1]$$

$$\frac{d[P]}{dt} = k_2[E1.S] - \mathbf{k_4}[\mathbf{P}][\mathbf{E2}] + \mathbf{k_{-4}}[\mathbf{E2.P}] \tag{3.5}$$

$$\frac{\mathbf{d[E2.P]}}{\mathbf{dt}} = \mathbf{k_4}[\mathbf{P}][\mathbf{E2}] - (\mathbf{k_{-4}} + \mathbf{k_5})[\mathbf{E2.P}] \tag{3.6}$$

$$\frac{\mathbf{d[E2]}}{\mathbf{dt}} = -\mathbf{k_5}[\mathbf{E2}][\mathbf{P}] + \mathbf{k_{-5}}[\mathbf{E2-P}] + \mathbf{k_5}[\mathbf{E2.P}] \tag{3.7}$$

Similarly, if we wanted to add an inhibitor to the modelled system , it is necessary to rewrite part of Eq. 3.2:

$$
\begin{aligned}
\frac{d[S]}{dt} &= -\, k_1[E1][S] + k_{-1}[E1.S] \\
\frac{d[E1]}{dt} &= -\, k_1[E1][S] + k_{-1}[E.S] + k_2[E1.S] - \mathbf{k_6}[\mathbf{I}][\mathbf{E1}] + \mathbf{k_{-6}}[\mathbf{E1.I}] \\
\frac{d[E1.S]}{dt} &= +\, k_1[E1][S] - k_{-1}[E1.S] - k_2[E1.S] \\
\frac{d[P]}{dt} &= +\, k_2[E1.S] \\
\frac{\mathbf{d[I]}}{\mathbf{dt}} &= -\mathbf{k_6}[\mathbf{I}][\mathbf{E1}] + \mathbf{k_{-6}}[\mathbf{E1.I}] \\
\frac{\mathbf{d[E1.I]}}{\mathbf{dt}} &= +\mathbf{k_6}[\mathbf{I}][\mathbf{E1}] - \mathbf{k_{-6}}[\mathbf{E1.I}]
\end{aligned}
$$

Moreover, the two modifications cannot be composed: if the modeller wanted both the deactivation through a kinase and the competitive inhibition, he would have to write a third, different model. This is a known problem, pointed out by researches that worked on the equivalences between the various formalisms (process algebras, differential equations and chemical equations [25, 30]).

Using chemical equations, as we have seen in the previous chapter, makes composability somehow better: starting from a simple model (kinetic rates are omitted for simplicity)

$$
\begin{aligned}
E1 + S &\rightarrow E1S \\
E1S &\rightarrow E1 + S \\
E1S &\rightarrow E1 + P
\end{aligned}
$$

we can add de-activation through a phosphatase $E2$ in a very simple way:

$$
\begin{aligned}
E2 + P &\rightarrow E2P \\
E2P &\rightarrow E2 + P \\
E2P &\rightarrow E2 + S
\end{aligned}
$$

Similarly, it is possible to add competitive inhibition:

$$
\begin{aligned}
E1 + I &\rightarrow E1I \\
E1I &\rightarrow E1 + I
\end{aligned}
$$

Moreover, the approach is composable: the equations can be just stick together, and with

the help of a correct stochastic simulation algorithm, they will just reproduce the expected behaviour.

This approach, however, is not perfect. First of all, composability relies on naming: enzymes, molecules and in general entities must have precise names. Entities with the same name will be treated as one species; to avoid conflicts when combining modules, unintended name clashes have to be avoided. The second drawback is that this way of modelling is really verbose: behaviour is *listed* instead of *specified*; in case of multiple possible interactions, the number of equations "explodes" in an exponential way. Some models, like the construction of polymers of unbound length, cannot even be expressed with a finite set of chemical equations [30].

A way to overcome this limitation is to describe the behaviour of the single entities in a *computational* way: instead of saying what the molecule can do by listing the reactions in which it can participate, we describe what the entity does when interacts with another entity; in this way, the set of equations ' $E1 + S \rightarrow E1.S \leftrightarrow E1 + S \rightarrow E1 + P$' is transformed into a set of three processes[1], for $E1$, S and P:

> $E1$ waits for a message on its `e1bind` channel. Once it receives the message, it becomes "occupied"; from that point, it can either send an activation message or wait for an unbinding request;
>
> S sends a complementary message on `e1bind`, matching $E1$'s action. When a match is found, S change its state, becoming an S' process, where it can either wait for an activation message or send an unbinding request;
>
> P, at the moment, does nothing.

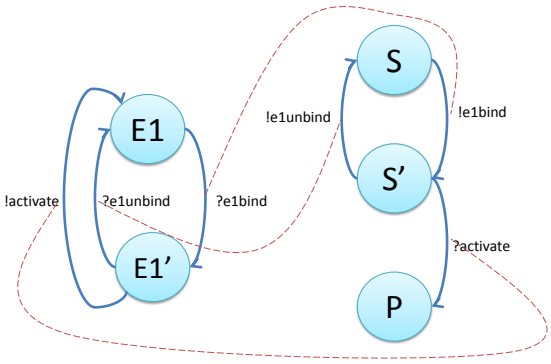


Figure 3.1: Interacting automata

[1]Processes, messages, channels and other concepts derived from concurrency theory were introduced in Section 2.4.2

Each process, representing one entity, can be represented graphically as two interacting automata [31] (see Figure 3.1), or alternatively with a more standard textual description. Recall from the previous chapter that in the textual description of process algebras the behaviour of each entity is represented as a set of actions (input '?', output '!', etc.) composed using operators. Sequencing of actions is obtained using the '.' operator; '+' is the choice operator, representing possible alternative behaviours (*or*), and '|' represents the parallel composition of behaviours (*and*):

```
E1 := e1bind?().(activate!().E1 + e1unbind?().E1);
S := e1bind!().(activate?().P + e1unbind!().S);
P := nil; //for now, P does nothing
```

The behaviour for P can be modified so that, for example, it acts like an enzyme, sending activation messages on a specific channel:

```
P := p_activate!();
```

Adding de-activation through a phosphatase $E2$ requires the specifications of a process for $E2$ and the addition (using the *choice* operator) of the correct behaviour to P:

```
E2 := deactivate!().E2;
P := previous_behaviour + deactivate?()
```

if the simpler $E2 + P \rightarrow S$ kinetics suffices, or

```
E2 := e2bind?().(deactivate!().E2 + e2unbind?().E2);
P := previous_behaviour + e2bind!().(deactivate?().S + e2unbind!().P);
```

if we wanted to reproduce the more complex behaviour of the initial model.

3.1.1 Biological inspiration

Although modular, this language risks being too "low level": for example, it is not possible to provide explicit representations for wrappers, membranes, and in general all the subdivisions at the base of living things, and of their respective points of interaction (ion channels, receptors, membrane proteins, protein domains). Communication is at the same time too broad and too specific: too broad, because there is no limitation in the *scope* of messages; every entity interacting on the same channel can exchange messages, with no notion of encapsulation or *information hiding*. Too specific, because entities must interact on a precise channel, know its name, and exchange messages at a fixed rate defined by the channel.

The first problem can be mitigated using a *restriction* operator, like in pi-calculus; the restriction operator limits the *scope* (i.e. the visibility) of messages to a sub-set of processes. However, its usage to codify physical boundaries can be cumbersome.

To solve the problem of too specific interactions we need to relax the typical *key-lock* mechanism used in message passing: classical process calculi assume a key-lock model for interactions, where a strict, exact matching on channel names is required in order to allow two parties to communicate. Reactions with non-exact matching, however, are quite common in biology [3] and they are relevant in the creation of drugs (consider, for example, competitive inhibition of receptors).

A possible solution to both problems is to "wrap" or enclose entities in a structure. In order to create a more general abstraction, this structure is logical: it can reflect either the physical boundaries of the object (e.g. when the entity is a protein, or a cell) or not. In this way, the structure defines a *scope*; every channel name, every interaction is by default isolated from the rest of the environment. If the modeller wants to expose some channel, it has to explicitly allow for that; for example:

```
E1 :=
(e1bind, e1unbind, activate) //List of public, exposed channels
[  // Square brackets are used to isolate a process
   e1bind?().(activate!().E1 + e1unbind?().E1)
];

E2 := (deactivate) [ deactivate!().E2 ];
```

A corresponding graphical notation could help in making the isolation clearer (see Figure 3.2). Each entity could be written in isolation, and there should be no worries about name clashes on internal channel names.

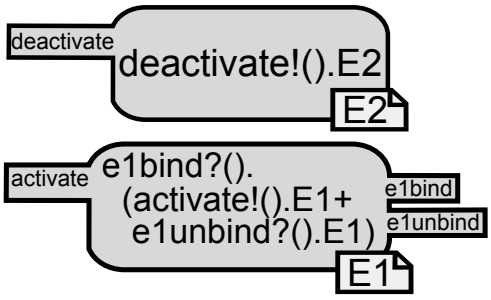


Figure 3.2: Encapsulation of processes

Building on this solution, it is possible to relax the strict matching on channel names. In biology, an interaction occurs because of the characteristics of the two interaction sites: for example, a ligand can interact with a receptor on the basis of its shape, charge, amino-acid sequence, and other physical or chemical *characteristics*.

We can assign a *name* to these characteristics; then the characteristics of an interaction site can be described by a set of names attached to it. Let's call this set of names the

type of the interaction site. Let's also call *box* the structure that enclose processes and *binders* the interaction sites (see Figure 3.3):

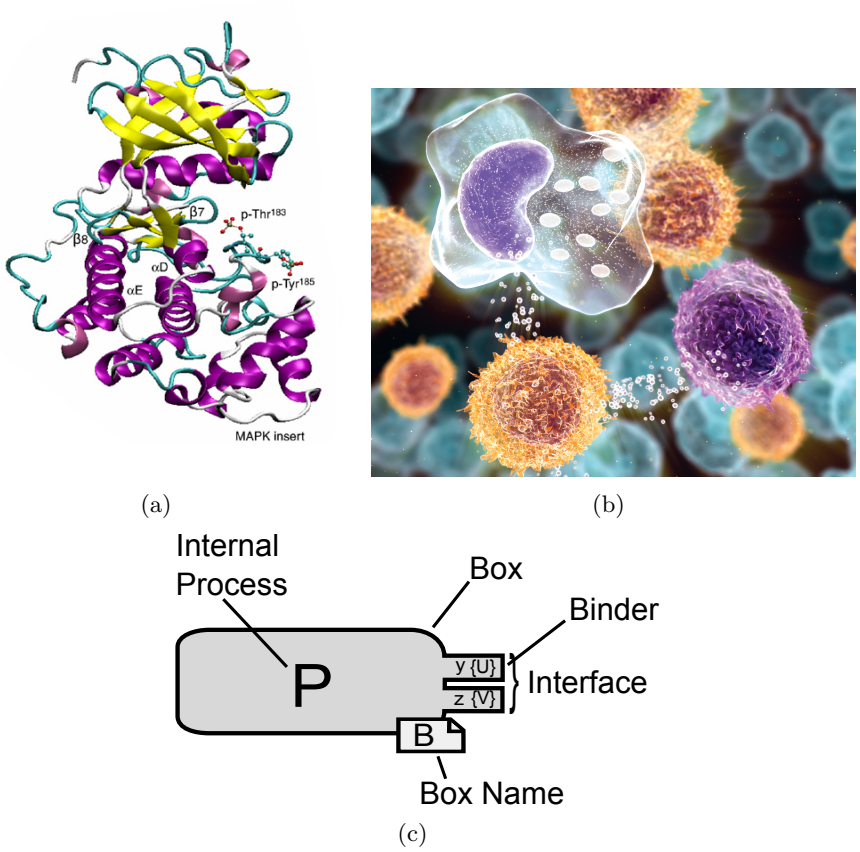


Figure 3.3: Following Regev and Shapiro [189], the language draws a parallel between processes and molecules. Both the protein in (a), with its domains, and the cell in (b), with its receptors, can be mapped onto the box in (c), with its interaction sites (*binders*, or collectively *interface*) and its internal behaviour (P). The process P reacts to messages on binders, like the living cell reacts to stimuli on its receptors.

```
E1 := (e1bind: {A, B}, e1unbind : {C, D}, activate: {E})
   [ e1bind?().(activate!().E1 + e1unbind?().E1) ];
S := (e1bind: {B}, e1unbind: {D})
   [ e1bind!().(activate?().P + e1unbind!().S) ];
```

Communication capability now can be expressed as a function of the characteristics of each interface: if they are compatible, the two can communicate and therefore interact. This compatibility can be expressed as the *intersection* between the two sets: if two of the interaction sites on their respective entities share some characteristic, communication is allowed (Figure 3.4).

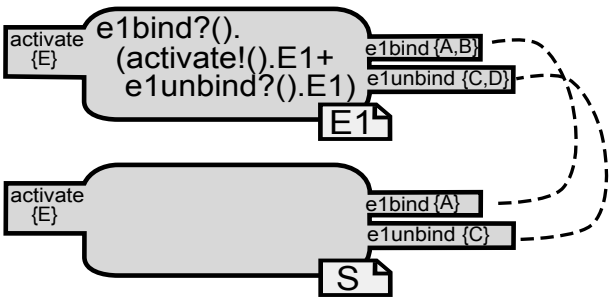


Figure 3.4: Processes with typed interaction sites can interact when types are compatible

Another important aspect of encapsulation is that it improves modularity of the language. Consider enzymatic interaction with inhibition, and try to modify our process algebra example to add an inhibitor:

```
E1 := e1bind?().(activate!().E1 + e1unbind?().E1);
S := e1bind!().(activate?().P + e1unbind!().S);
P := nil; //for now, P does nothing
```

Ideally, we should be able to add a process or entity describing the inhibitor to the system, without modifying the description of the enzyme or of the other entities in the system. Adding a process for the inhibitor requires us to re-use the existing "binding" channel in $E1$, in order to have competitive binding:

```
I := e1bind!().e1unbind!().I;
```

After binding, by sending a message on the `e1bind` channel, the inhibitor issues an unbind message and waits for a match to arrive. In this way, it occupies one enzyme $E1$, making it unavailable for interactions with S.

This model has several problems:

- there can be an undesired interleaving. Consider what happens after an enzyme becomes "bound" to a substrate, and another one becomes "bound" with an inhibitor. It is not possible to tell anymore which one was bound with another one. This should be not a problem in this simple model, but can possibly and easily lead to a non correct model in more complex scenarios;

- it is not simple to use different rates. Up to now we omitted to specify kinetic rates and we supposed that, in process calculi tradition, they were attached to channel names; however, a biological entity could have interactions with multiple partners, like in our enzyme-substrate-inhibitor example. It is possible to codify multiple interactions using different channel names and/or private channels, but this harms model composability.

In fact, a common solution is to use the shared channel to send a private channel that will be used to represent the real interaction; for example, in pi-calculus this is obtained using the 'new' and/or the restriction (μ) operators:

```
E1 := e1bind?(act_channel).(act_channel!().E1 + e1unbind?().E1);
S1 := e1bind!(new channel my_channel(rate(0.1))).
        (my_channel?().P1 + e1unbind!().S1);
S2 := e1bind!(new channel my_channel(rate(0.2))).
        (my_channel?().P2 + e1unbind!().S2);
```

In this way, $E1$ can interact with both $S1$ and $S2$ at different rates. However, if we want to use different rates for binding and unbinding, as required by the inhibitor example, we need to consider a more complex model, like the one in [31]:

```
E1 := (new channel k1(rate1), new channel k2(rate2)).
    a!(k1,k2).(k1!().E1 + k2!().E1);
S != a?(k1,k2).(k1?().S() + k2?().P);
```

This pattern can be generally used for complexation and decomplexation; however, it get across that this solution is not easy to use; furthemore, this pattern does not allow to add interaction capabilities in a simple a modular way.

In addition, when processes are encapsulated, like in our case, sending a private channel loses meaning: out of S scope, `my_channel` will have no connection with S[2]. So this pattern does not work well with our "new" language.

Encapsulation provides a simpler solution: it is possible to attach the communication rate to the capability itself. What we obtain is a communication model based on the *affinity* between interaction sites: the affinity gives to the modeller the possibility both to tell when two sites can interact and at which rate they interact:

```
E1 := (e1bind: {A, B}, e1unbind : {C, D}, activate, {E})
   [ e1bind?().(activate!().E1 + e1unbind?().E1) ];
S1 := (e1bind: {B}, e1unbind: {D}, activate, {F})
   [ e1bind!().(activate?().P1 + e1unbind!().S1) ];
S2 := (e1bind: {B}, e1unbind: {D}, activate, {G})
   [ e1bind!().(activate?().P2 + e1unbind!().S2) ];
affinity({A, B}, {B}) = rate(...);
...
affinity({F}, {E}) = //some computed rate
affinity({G}, {E}) = //some other computed rate
```

The problem with interleaving during complexation and decomplexation, however, remains. More in general this is a problem with boundaries, both logical and physical: models need to cross and modify them. Biology is full of examples where boundaries are modified, created or destroyed: mating, exocytosis, phagocytosis, secretion, vacuoles, mitosis, are only some of the processes in which creation of membranes are concerned.

[2]Technically, `my_channel` will be a free name in the receiving process

A possible solution is to provide explicit actions for dealing with membranes, like in [28] and [188]. However, these actions add complexity to the modelling language, from both a practical and a theoretical point of view.

Another alternative is to equip the language with a reduced set of primitives for handling boundaries modification and write more complex actions in terms of those primitives. The simplest possible primitives, in this case, are two: one for merging two enclosures, and one for dividing an existing enclosure in two parts. Let the first one be called f_{join}, and the second one f_{split}. These primitives are functions over boxes; f_{join} takes as input two boxes (each as an interface, process pair) and returns a single box; f_{split} takes as input a single box (defined as an interface and two processes in parallel) and returns two boxes:

$$f_{join} = (Box_1(i_1, P_1), Box_2(i_2, P_2)) = (Box(i, P_1|P_2), \sigma_1, \sigma_2)$$
$$f_{split} = (Box(i, P)) = (Box_1(i_1, P_1), Box_2(i_2, P_2), \sigma_1, \sigma_2)$$

were σ_1 and σ_2 are functions used to perform name substitution (required, for example, to avoid name clashes) inside P_1 and P_2 respectively.

Using f_{join} and f_{split}, our example of enzymatic interaction can be written in a simpler way:

```
E1 := (e: {E})
   [ activate!().E1 ];
S := (s: {F})
   [ activate?().P ];

join (Box_1, Box_2) =
   if Box_1 == E1 and Box_2 then
      (Box(e: {E}, s: {F}, activate!().E1 | activate?().P), id, id)
   else
      bottom;

split (Box) =
   if Box == Box(e: {E}, s: {F}, E1 | P) then
      (E1, Box(s: {F}, P, id, id)
   else
      bottom;
```

where complexation and decomplexation are treated as joining and splitting of boxes respectively.

When we examine carefully the model, however, we can see that some old problems reappear: when the processes for the enzyme $E1$ and the substrate S are joined, they can interact by communication through a channel that is now shared. The channel, which was previously private both to $E1$ and S, needs to have the same name.

In many cases it is possible to write a clever join function that takes care of renaming the channels appropriately, but the function would be very complex and *ad-hoc*, hence not usable in the general case. Another problem arises from the global nature of join and split functions: adding new rules for joining and splitting boxes requires changing the functions, adding more and more cases.

3.1.2 Towards the implementation

As we have seen in the previous chapters, one of the most relevant advantages, if not the most important, of using a formal language for building biological models is that models become *executable*. Process calculi for systems biology provide an *operational* description of the system; therefore, in order to correctly model and simulate biological processes, they need to offer a formal description of how a model can be executed.

This formal description requires the definition of the syntax and semantics of the language, and of an *abstract stochastic machine* implementing the semantics. The definition of the semantics and of the abstract machine for BlenX is not the focus of this thesis; Romanel et al. [185] addressed these topics, covering all the aspects of the language[3] that we are introducing in this chapter. The interested reader can refer to [57], [185] or [197] for further details.

The machine *executes* the model by reducing entities and processes, choosing among the possible concurrent semantic rules (*mono* actions involving one process or *bi* actions involving pairs of processes, like for communication) by means of one of the methods described in Section 2.2.

The primary goal of an abstract machine is to efficiently perform computations over the model on standard computer hardware. By computation, we mean simulation, generation of the underlying continuous time Markov chain and, in general, analysis of the model using an algorithm.

If we try to write a realistic abstract machine for our language, in order to implement an efficient stochastic simulator for it, some problems with the language pop up.

One of these problems is efficiency: when many entities are present in a system, the abstract machine should be able to handle them efficiently. Previous stochastic simulators for process calculi [4] used as internal representation lists of processes, where each individual was represented as a single process. Propensities were computed by maintaining a list of available inputs and outputs on a given channel; when the next reaction channel was chosen using a stochastic method (usually, the Gillespie SSA), the machine randomly chose one of the available reactions on that channel with equal probability, by choosing

[3] With the exception of continuous variables, which will be introduced in the next section

[4] We are referring to the scenario of late 2006, when BlenX was being designed.

an input through a uniform random distribution and then similarly selecting an output from the remaining list.

Clearly, when the number of individuals gets greater, the selection process becomes slower; moreover, this encoding, although correct, is distant in philosophy from the design principles of many stochastic methods, including Gillespie, which are based on species, populations and counting for the sake of efficiency.

Counting

A possible solution to this problem is to define a notion of equivalence, enabling the grouping of processes in classes that could be called *species* and treated as such by the various methods.

Various notions of equivalence for process calculi exist: among them, *structural congruence* seems a good choice. Some minor changes to the language, like removing the restriction operator (that has limited use in our case, as boxes already substitute it in most of the cases) and substituting recursion with a *replication* operator, make structural equivalence decidable in polynomial time, and therefore efficient to implement.

The introduction of the replication ('*rep*') operator makes necessary to add the possibility of defining internal processes independently, to avoid duplications of code:

```
let Ep : proc = e1bind?().(activate!().r!() + e1unbind?().r!());
E1 := (...)
   [ rep r?().Ep | Ep ];
```

Affinity

Another decision that has to be made regards affinities, in particular how a measure of affinity between two types can be computed. Previously we mentioned *set intersection* and *affinity computed on types characteristics* as possible ways of computing an affinity measure, but we did not specify how this computation can be done. Since this computation can differ from one scenario to another, a simple solution is to let affinity be "user defined": the user may list explicitly the pairs of compatible types, and how strong is their affinity (e.g. which is their interaction rate). This list can be specified separately, possibly in another file, so that it can be automatically generated by a pre-processor[5].

Using this solution each type, which we shall call *binder identifier*, can be identified by a single label. Our enzyme-multisubstrate example can therefore become:

```
let Ep : proc = e1bind?().(activate!().r!() + e1unbind?().r!());
E1 := (e1bind: EB, e1unbind: EU, activate: EA)
   [ rep r?().Ep | Ep ];
```

[5]This possibility will be used in Chapter 7

```
let S1p: proc = e1bind!().(activate?().r!() + e1unbind!().P1p);
let P1p: proc = //P1 behaviour...
S1 := (e1bind: S1B, e1unbind: S1U, activate: S1A)
   [ rep r?().S1p | S1p  ];

let S2p: proc = e1bind!().(activate?().r!() + e1unbind!().P2p);
let P2p: proc = //P2 behaviour...
S1 := (e1bind: S2B, e1unbind: S2U, activate: S2A)
   [ rep r?().S2p | S2p  ];
```

In a separate file, we declare the various affinities:

```
(EB, S1B, rate(...));
...
(E1A, S2A, rate(...));
```

3.1.3 Field tests

The best way to test a language is to write a *compiler* and a *machine* for it, and start writing programs. In the case of a language for systems biology, the simplest possible machine is a simulator, and the programs are models of biological interactions.

Suppose to be in charge of the development of a simulator for our language: very soon it becomes clear that another problem plagues the join and split constructs. For the maximum flexibility, f_{join} and f_{split} can be defined as lambda functions; this means that it is possible to express any kind of computation, including recursion using a Y-combinator. This is clearly not desirable: f_{join} and f_{split} are defined over the global state of the system, hence the evaluation of these functions had to be done at each step of the computation. This evaluation can heavily affect performance, and in the worst case it can even not terminate.

A better way of dealing with changes on boxes structures is needed.

Complexes

Complexation and decomplexation can be represented with f_{join} and f_{split}. However, doing it in a modular and correct way it is not as easy as it should be: creation of complexes is one of the most common scenarios in modelling biological pathways.

Many biological processes rely on the formation of bonds between entities; complexes and polymers have both a structural and a functional role. Even our simple enzyme-substrate example and its variants (multi substrate, inhibition) works by constructing complexes.

Since complexes are so important in biology, a reasonable choice is to consider them as *primitive constructs*, like entities or membranes. Complexes are molecules composed of

base structural units connected by chemical bonds (Fig. 3.5(a)); polymers can be thought as large complexes composed of repeating structural units (Fig. 3.5(b)). In our language, these structural units are represented by boxes, where each box has one or more interaction site (*binder*). We can use binders not only as interaction points, but also as *binding sites*: when two binders are *bound*, a private interaction channel is built (Fig. 3.5(c)); like in nature, this connection forms a complex that can have a functional role (by allowing an enzymatic reaction, for example), a structural role (formation of a biolpolymer like DNA or proteins themselves) or both. Technically, a complex is a graph, where entities are nodes connected through their binders (see Figure 3.5(d)).

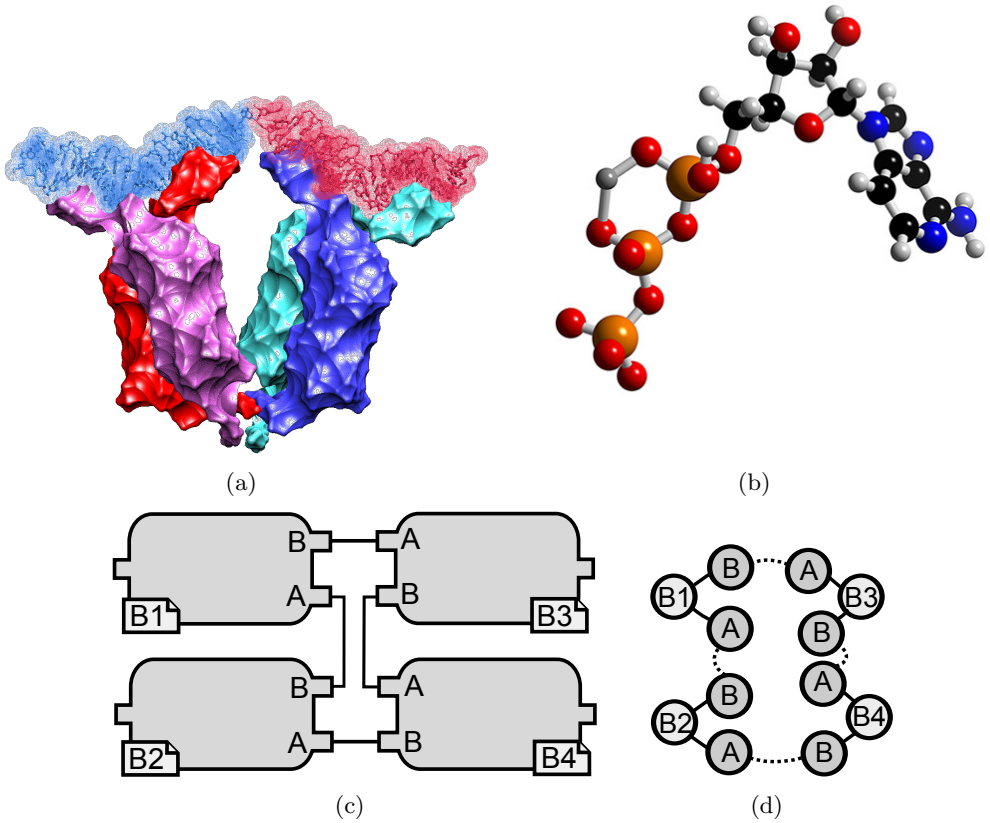


Figure 3.5: Complexes in biology: functional units or dimers (a), polymers (b). Complexes in our language: boxes represent functional or structural units, connected through binders (c); boxes connected to form a complex can be seen as graphs (d)

The introduction of complexes can be made with very little impact on the language: it suffices to modify slightly the affinity definition. When we want to allow for complexation through a binder, we specify rates for complexation and decomplexation in addition to the usual rate of communication:

```
(E1A, S2A, complexation_rate, decomplexation_rate, communication_rate);
```

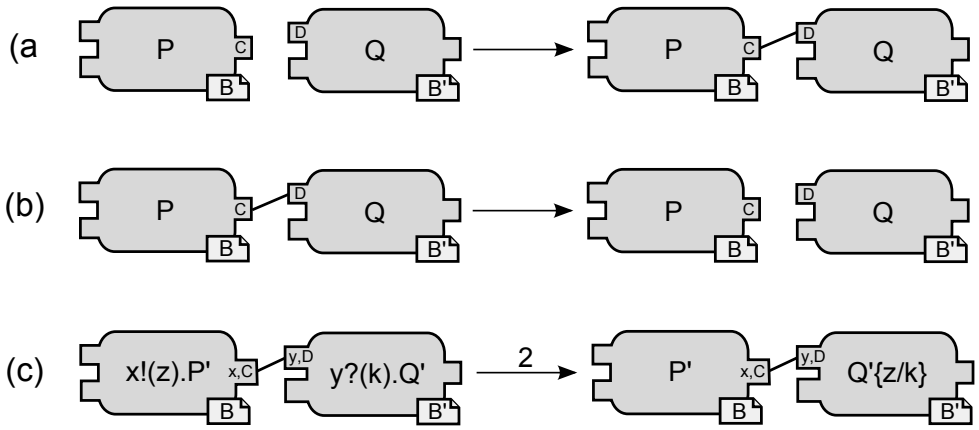


Figure 3.6: Complexes in the language: (a) formation of a complex; (b) decomplexation; (c) interaction while complexed

Notice that we want to preserve also the previous way of specifying affinity: using a single communication rate is still valuable for representing those interactions when complexation is not involved or is abstracted.

Using complexes, the model for enzymatic interactions (with or without inhibition, with or without multiple substrates) becomes very compact and easy to write and understand:

```
E1 := (activate: EA)
   [ rep activate!() ];

let P1p: proc = //P1 behaviour...
S1 := (activate: S1A)
   [ activate?().P1p ];
```

In a separate file, we declare the various affinities:

```
(EA, S1A, rate(...), rate(...), rate(...));
```

Adding an inhibitor, or another substrate, can be done easily; in the model file, we add a definition for the I process:

```
I := (i: IA)
   [ nil ];
```

And in the affinity file, we add a new affinity:

```
(EA, S1A, rate(...), rate(...), rate(...));
```

The I process does nothing, as its only role is to bind with an enzyme and "occupy" its interaction site; the language deals automatically with this behaviour. Adding another substrate, or any other combination (another inhibitor, a different catalyst, and so on) requires a comparable, limited effort. Using a specific construct for complexes has great advantages for modelling low-level, reversible reactions.

However, it has also disadvantages with respect to the usage of join and split functions: it cannot be used at a higher level of abstraction, where the detail of the reaction are not known and we simply need a way of "replacing" one or more "reagent" entities with one or more "product" entities, and cannot be used to produce or delete entities.

It is possible to argue that this way of modelling follows the philosophy of Antoine Lavoisier, the father of the modern chemistry:

> Rien ne se perd, rien ne se cre, tout se transforme.

However, from a system perspective sometimes we need to model the overall behaviour; several biological *operational patterns* – endocytosis, meiosis, exocytosis, replication – are easy to express with constructs like join and split functions [181] while they not find an immediate mapping to formation and dissolution of complexes.

As a simple example, we wrote a model for the classic Lotka-Volterra predator-prey dynamics. This model is easy to express using chemical equations:

$$Predator + Prey \rightarrow 2Predator$$

$$Predator \rightarrow Nil$$

$$Prey \rightarrow 2Prey$$

Preys breed with a specific rate, with no specific interaction with the environment; instead predators need to interact with the system in order to proliferate (in this case, by "eating" a prey).

Without join and split functions, we can still write the model in our language by making the preys interactions with the system explicit:

```
let PreyP: proc = rep r?().
    (x?().ch(x, PZ).p!() //being eaten, become a predator
   + x!().r!()); //eat food

let FoodP: proc = rep f?().
    (x?().ch(x, PX).r!()); // when eaten, become a prey

let PredatorP: proc = rep p?().
    (x!().p!() // eat a prey
   + delay(die_rate).ch(x, PZ).f!()); //or die and become food

Prey := (x: PX)
   [ PreyP |
     FoodP |
     PredatorP |
     r!() //activate the first instance as a prey
   ];
```

```
Food := (x: PZ)
   [ PreyP |
     FoodP |
     PredatorP |
     f!() //activate the first instance as a "food"
   ];

Predator := (x: PY)
   [ PreyP |
     FoodP |
     PredatorP |
     p!() ];
```

In a separate file, we declare the affinities:

```
(PX, PZ, ...)
(PY, PX, ...)
```

In this model, we make explicit the interaction with the environment, introducing "Food" for preys. In this way, the number of boxes is conserved, as they are simply transformed from one species to the other. Notice that this is possible thanks to the choice of structural congruence as an equivalence relation to group boxes in the same class. In this case, there is no need for constructs for creating or merging boxes; however, this is not always the case and the resulting model is harder to create and understand.

Events

Implementing join and split as functions is too expensive to be practical; however, we can think of joining and splitting as *events* of a *rewriting system*. Events can be based on conditions over the global state of the system, mainly presence or absence of same species, and have an associated stochastic rate as every other action in the language:

```
Prey := (x: PX) [ nil ];

Food := (x: PZ) [ nil ];

Predator := (x: PY) [ nil ];

Predator2 := (x: PY2) [ nil ];

when (Predator, Prey : eat_rate) join (Predatory2);
when (Predator2 : inf) split (Predator, Predator);
when (Prey : breed_rate) split (Prey, Prey);
when (Predator : death_rate) delete;
```

Besides join and split, other events (delete, new) can be introduced to make modelling easier. Conditions on species (presence, counting) are based on structural congruence, making event implementation very efficient.

3.1.4 Real world models

Up to now the language was used to build simple models. However, the real test for a language starts when other people use it to build real world models: things are not perfect and there are many uncertainties and unknowns.

Patterns

A very common model used as a *test drive* for modelling languages is the classical MapK cascade signalling pathway [108]. Paradigms based on communication should make it easy to express this kind of pathway; moreover, the introduction of *complexes* in our language makes very easy to write a detailed model, where all the enzymatic reactions are expressed as a sequence of binding-transformation-unbinding:

```
let E1: bproc = #(e1: signalE1)
[ rep e1!().nil ];

let E2: bproc = #(e2:1, signalE2)
[ rep e2!().nil ];

let pKKK : pproc =
   act?().ch(act, ubI).unhide(deact).unhide(p).
   deact?().hide(p).ch(deact, ubD).ch(act, irecKKK).
            ch(deact, drecKKK).hide(deact).kkk!().nil;
let pKKKr : pproc = (rep kkk?().pKKK | pKKK);

let KKK: bproc = #h(p, KKKkase), #(act, irecKKK), #h(deact, drecKKK)
   [ pKKKr | rep p!().nil ];

// Second stage of phosphorilation: we wait for a deactivation
// from our phosphatase
// Then we loop back to first phase of phosphorilation (X_P)
let pKK_PP : pproc = ( deact?().hide(p).xp!().nil );

// First phase of phosporilation
let pKK_P : pproc = rep xp?().(
   act?().ch(act, ubI).ch(act, incKK2).ch(deact, decKK2).unhide(p).pKK_PP +
   deact?().ch(deact, ubD).ch(act, incKK).ch(deact, decKK).hide(deact).x!().nil);

let pKK : pproc = act?().ch(act, ubI).ch(act, incKK2).unhide(deact).xp!().nil;
let pKKr : pproc = (rep x?().pKK | pKK);
```

```
// KK bio-process
let KK : bproc = #h(p:1, KKkase), #(act:1, incKK), #h(deact:1, decKK)
   [ pKKr | pKK_P | rep p!().nil ];

// Second stage of phosphorilation: we wait for a deactivation
// from our phosphatase
// Then we loop back to first phase of phosphorilation (X_P)
let pK_PP : pproc = ( deact?().hide(p).xp!().nil );

// First phase of phosporilation
let pK_P : pproc = pre xp!().(
   act?().ch(act, ubI).ch(act, incK2).ch(deact, decK2).unhide(p).pK_PP +
   deact?().ch(deact, ubD).ch(act, incK).ch(deact, decK).hide(deact).x!().nil);

let pK : pproc = act?().ch(act, ubI).ch(act, incK2).unhide(deact).xp!().nil;
let pKr : pproc = (rep x!().pK | pK);

// K bio-process
let K : bproc = #h(p, Kkase), #(act, incK), #h(deact, decK)
   [ pKr | pK_P | rep p!().nil ];

// Phospatase for KK
let KKPase: bproc = #(de:1, KKpase)
   [ rep de!().nil ];

// Phospatase for K
let KPase: bproc = #(de:1, Kpase)
   [ rep de!().nil ];
```

It is possible to notice that we have three basic structures here:

- *signals*, like $E1$ and $E2$, which continuously send their messages of activation or deactivation;
- *phospatases*, like $KPase$ and $KKPase$; they have the same structure and functionality of signals, sending messages of deactivation with an appropriate rate;
- *kinases*, which have "sensing" domains (interaction sites, binders in our case), on which they "listen" for messages of activation and deactivation and an "effecting" domain that, like in the case of signals and phospatases, continuously send a message of activation.

This kind of structures, where entities have roughly the same behaviour and differ only for some little particulars (like rates) are pretty common; a technique from programming languages helps in this situation.

Templates, also called *generics* or *parametric processes*, are constructs used to replicate basic structures (e.g. classes in object oriented programming). Templates enable a metaprogramming technique that allows a primitive to work with many different parameters without being rewritten for each one. In programming languages templates are usually parametric with respect to data types, even if other kind of parameters, like dimensions, can be used.

Templates play an important role in the easy definition of *patterns*, elements of reusable design through which it is possible to express and reuse common behaviours (see Box 1).

Box 1 What are patterns?

The concept of *patterns* was developed originally by the architect Christopher Alexander. He intended patterns as basic architectural design ideas to be collected in a list of reusable descriptions, as elements of a pattern language. Elements of this language may be combined, governed by certain rules, and serve as an aid to design cities and buildings. A pattern is not a finished design; it is a description or template for how to solve a problem that can be used in many different situations.
The concept had a great success in computer science. Alexander writing had a great influence in the research on programming language design, modular programming, object-oriented programming, software engineering. Its work *A Pattern Language*[5] had a great influence in the software engineering *design patterns* movement, where patterns are intended as general reusable solutions to a commonly occurring problem in software design.

In our case, we can imagine boxes being parametric with respect to their internal *process* (or part of it), channel *names*, *binder* identifiers; similarly, processes can be parametric with respect to *names* for channels or binders, and processes or part of them. Writing generic templates for kinases, phosphatases and signals allow us to rewrite the model in a more compact way:

```
// Signal template
template Signal: bproc<<binder S>> = #(e,S)
   [ rep e!().nil ];

// Phospatase template
template Phospatase: bproc<<binder P>> = #(x, P)
  [ rep x!(minus).nil; ]

// Kinase templates: with single or double activation
template pSTATE2: pproc<<binder Intermediate, binder Active>> =
   rep state2?().(
      unhide(p).ch(recv, Active).
      recv?().hide(p).ch(recv, Base).state1!().nil);

template pSTATE1s: pproc<<binder Base, binder Active>>  =
   rep state1?().(
```

```
      unhide(p).ch(recv,Active).
      recv?().hide(p).ch(recv,Base).state0!().nil);

let pSTATE1: pproc =
   rep state1?().(
      recv?(what).what!().nil |
     (plus?().state2!().nil + minus?().state0!().nil));

template pSTATE0: pproc<<binder Base, binder Active>>  =
   rep state0?().(
      ch(recv,Base).recv?().ch(recv,Active).state1!().nil);

template SingleK: bproc<<binder K, binder Base, binder Active>> =
  #h(p, K), #(recv, Base)
[ recv?().ch(recv,Active).state1!().nil |
  pSTATE0<<baseK3,activeK3>> |
  pSTATE1s<<baseK3,activeK3>> |
  rep p!(plus).nil ];

template DoubleK: bproc<<binder K, binder Base,
                         binder Intermediate, binder Active>> =
  #h(p, K), #(recv, Base)
[ recv?().ch(recv, intK2).state1!().nil |
  pSTATE0<<Base,Intermediate>> |
  pSTATE1 |
  pSTATE2<<Intermediate,Active>> |
  rep p!(plus).nil ];

// Definition: activation and deactivation signals
let E1: bproc = Signal<<signalE1>>;
let E2: bproc = Signal<<signalE1>>;

// The three kinases (one with single phosporilation, two
// with double phosphorilation)
let K3: bproc = SingleK<<kaseK3, baseK3, activeK3>>;
let K2: bproc = DoubleK<<kaseK2, baseK2, intK2, activeK2>>;
let K1: bproc = DoubleK<<kaseK1, baseK1, intK1, activeK1>>;

// The 2 phosphatases
let P1: bproc = Phospatase<<paseP1>>;
let P2: bproc = Phospatase<<paseP2>>;
```

Notice that, besides being more compact, this model is also more general: it is possible to use the template definitions for kinases, phosphatases and signals to build any other signalling pathway.

Conditional events

The MapK model works as an ultra-sensitive switch: it has to be robust with respect to noise, but when a signal molecule is introduced in the system, it has to respond quickly with a non-linear response curve; similarly, when the signal cease to exists, it should quickly return to the initial, non active state. This behaviour is a consequence of the cascade of signals ($E1$ activates $K3$, which in turn activates $K2$, which will phosposrilate $K1$) and of the double-phosporilation of some of the kinases.

In order to test the model and run experiments on it, it is necessary to introduce and take away the signal molecules $E1$ and $E2$ at will. A natural way of doing it during a simulation is to insert and delete boxes using events; this time, however, events have a condition on time:

```
when (E1: time=3000.0: inf) delete(2);
```

This kind of events allow the modeller add controlled perturbations to the system.

Variables and expressions

Another example of a real-world model that we tried to express in our language is the cell-cycle model from Tyson and Novak [162]. Its original version is based on ODEs; when trying to translate it in our modelling language, a problem immediately arises from the chosen level of abstraction: almost every reaction in the model can be expressed in our language, except for those involving the mass.

The mass of a cell is the sum of all the contained proteins and molecules; during mitosis, the cell-cycle machinery checks both concentrations of particular molecules and the mass of the cell to decide weather is time to split the cell in two parts. The dependence between these molecules and their relation with the mass of the cell is unknown, therefore an abstraction must be used.

In the original model, the mass is a continuous quantity depending on time. The following equation is commonly used to express the growth of mass:

$$\frac{\delta m}{\delta t} = \mu \cdot m$$

where μ is a constant. If we discretize it we obtain:

$$\frac{\Delta m}{\Delta t} = \mu \cdot m \quad \rightarrow \quad \Delta m = \mu \cdot m \cdot \Delta t$$

To update the m variable every Δt, we can write the following expression:

$$m_{t(i)} = m_{t(i-1)} + \Delta m \quad \rightarrow \quad m_{t(i)} = m_{t(i-1)} + (\mu \cdot m \cdot \Delta t)$$

The discretized version of this equation can be used inside a modified version of the stochastic simulation engine to update the mass over time.

To integrate this kind of abstraction in the language, we need to introduce *variables*, *constants* and *expressions*. For example, the syntax to write the previous equation could be:

```
let m(0.1): var = mu * m init 0.2;
```

where 0.1 is an "hint" to the simulator, that could be used to set the integration step Δt[6].

With the introduction of variables, we also need an event that can modify their value. We shall call this event *update*. The event acts on variables, not boxes: when it is fired, an expression is evaluated, and the resulting value is assigned to the variable.

The condition of an *update* event involves no entities and no rate: the event is triggered as soon as a conditional expression is valid. This particular kind of condition is based on the traversal of successive *states*. Suppose we want to recognize the oscillatory behaviour in Fig. 3.7, and adjust the variable n, also depicted in the figure, by increasing it at every oscillation:

```
let n : var = 1;
let f : function = n + 1;
...
when (: A -> 20, A <- 20 :) update (n, f);
```

The concatenation of an arbitrary succession of states allows to overcome possible limitations that are often encountered when dealing with a stochastic approach, mainly noise. As an example, look at Fig. 3.8: the simple state list just introduced is not enough to capture the correct period of oscillations, as highlighted in the upper-right corner of the figure.

This issues can be solved by adding more states:

```
let n : var = 1;
let f : function = n + 1;
...
when (:A -> 10, A -> 20, A <- 20, A <- 10:) update (n, f);
```

This event can capture correctly the behaviour of the noisy oscillating system, as depicted in Fig. 3.9.

[6]The current version of BetaSim, the simulator for the BlenX language introduced in Sec. 3.2, uses this hint to set the time step for an Euler solver, which is used to integrate the value of continuous variable over time. Clearly, the ideal solution would be to use an hybrid simulation algorithm; this solution will be studied in a future release of BetaSim.

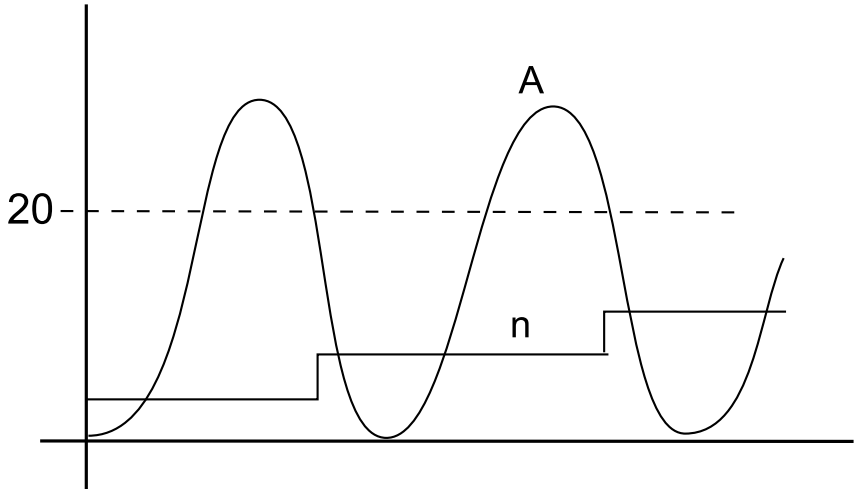


Figure 3.7: The specie A exhibits an oscillating behaviour, captured by a state-list condition. n is a variable that "counts" the number of oscillations.

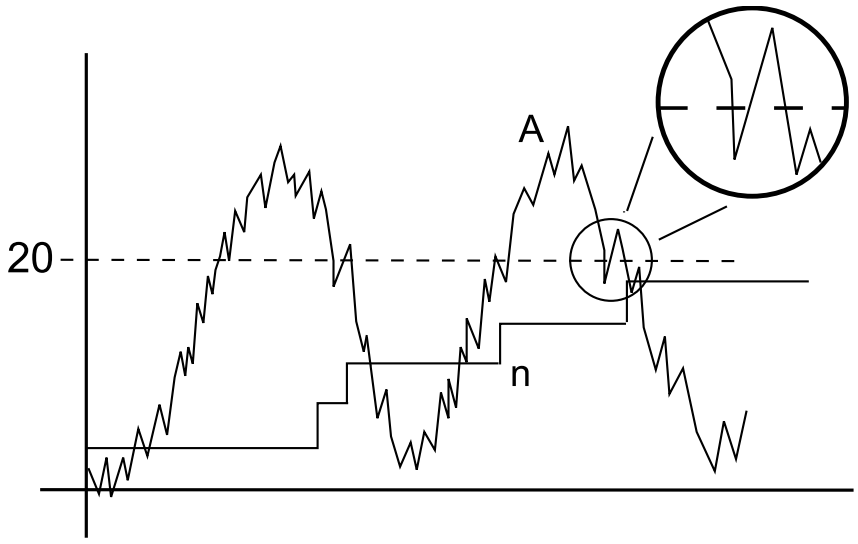


Figure 3.8: The specie A exhibits an oscillating behaviour, but data has some noise: the state-list condition cannot capture it and n is updated in a wrong way.

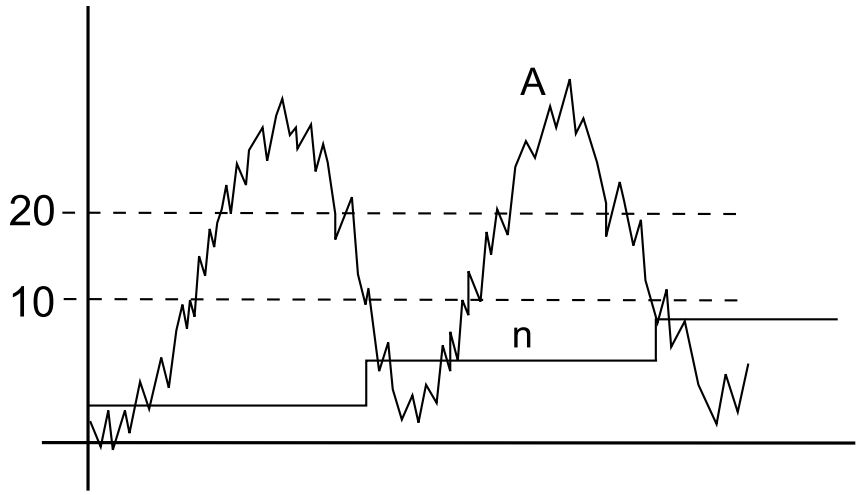


Figure 3.9: The new state-list condition is able to capture the oscillations correctly.

The addition of variables, expressions (functions) and *update* events with state conditions enrich the language to the point that it is now usable for many different scenarios.

The language we have built in this section is BlenX.

3.1.5 History and credits

The design of a language is a team effort; many people at CoSBi were involved in the definition of BlenX, at various levels: specification of the semantics, implementation, definition of the syntax, refinement, suggestion and on-field testing.

The process of creation of a new language accounted in the previous section mirrors not only the rationale behind BlenX, but also its evolution through time.

2004: Beta-binders

The language we "re-created" in Section 3.1.1 closely resembles the original Beta-binders process algebra, designed and defined by Priami and Quaglia; in particular, the concepts of compatibility based on sets and of join and split functions was introduced in the original Beta-binders papers [180, 181].

The proposal to relax the key-lock assumption also came from Beta-binders [180]. In fact, Beta-binders provide the means to model the enclosing surfaces of entities and possible interactions taking place at the level of the enclosing surfaces. Processes are encapsulated into boxes with interfaces; interfaces have an associated *type* that represents the interaction capabilities of the box, closely following the concept of cells of computations [189] introduced in the previous chapter:

> Processes, the basic interacting computational entities of these languages, have an internal state and interaction capabilities. Process behaviour is governed by reaction rules specifying the response to an input message based on its content and the state of the process. The response can include state change, a change in interaction capabilities, and/or sending messages.

Types as sets of names is also how they were shaped in the original Beta-binders papers; as in Section 3.1.1, the interaction was enabled if and only if the types of the interfaces of the two partners are not disjoint.

Prandi et al. introduced the notion of *affinity* [176]; this allowed a finer control than the one expressed by the intersection of the types of interfaces.

2006: First version of BlenX

The language introduced in Section 3.1.2 is essentially the first version of BlenX (which was still called Beta) and it is mainly the work of Alessandro Romanel: he developed both the theory and a working implementation of the structural congruence in BlenX [196], opening the way to the idea of species as classes of structural congruent processes that leads to an efficient implementation of the Gillespie Stochastic Selection Algorithm.

2007: Second version of BlenX, BetaWB

The second revision of BlenX [58], which we re-created in Section 3.1.3, was implemented in the first publicly available version of BetaWB [55].

This language revision contained the concepts of complexes and events; with Romanel and Priami we designed and implemented them in a Workbench, a set of tools for simulation (BetaSim), model building (BetaDesigner) and inspection of results (BetaPlotter).

In particular, we considered that complexes are really common in biological systems; even if in some cases it is possible to abstract over them, in other cases they have a central role and must be modelled explicitly. As we have seen, modelling a complex formation using exclusively message exchange it is possible, but not very convenient. Hence, in BlenX complexes as first-class citizens.

Complexes as native language constructs are a peculiar feature of BlenX, where boxes play the role of monomeric units; they can be used to represent any ensemble of two or more boxes, from dimers to more complex polymers. New complexes can be generated at runtime as a consequence of binding and unbinding actions -similarly to what happens when new boxes are created after monomolecular and bimolecular reactions-, and so it is possible to easily generate complex biopolymers made of dozens or hundreds of boxes from relatively simple and compact programs.

The addition of events based on structural congruence added an easy way to deal with reactions that involved the creation, destruction and fusion of boxes. Without them, a box can only be modified in its internal process or in its interface (the set of binders); therefore, events filled the gap left by the lack of join and split functions.

2008: BetaWB version 2

The second version of BetaWB [56] brought more features in the language. The workbench allowed other people to experiment with the language, building more complex model. Listening to their feedback, the language was modified to include new features [57], the ones introduced in Section 3.1.4.

The first important new feature is the macro rewriting system that allows the definition of parametric processes and parametric boxes, called *templates*. Being entirely a compiler transformation, it does not add to or modify in any way the semantics of BlenX, but it allows the modeller to save considerable time and to write more compact code by eliminating many repetitions in the code.

The second feature worth mentioning is the introduction of variables and functions; they led to the realization of an hybrid simulation algorithm that allows the modeller to insert some global behaviour inside the system. Both this constructs are a step away from a pure process algebra approach (the first introduces a sort of rewriting system to specify rewriting rules, the second the ability of introducing continuous variables in a way that was possible only with ODEs before); this is an example of the kind of compromises we had to face when dealing with a real-world language that has to be both formal and usable.

Models

Besides people involved directly in the design and implementation of language features, people who worked on the creation of models helped in shaping BlenX too. In particular, the NfKB model [135] used binders to codify space, following the original Beta-binders ideas; as such, it proved to be a good check of the validity of our approach, which retained Beta-binders capabilities. The Cell Cycle model [164] was a test for events, variables and functions; finally, the Actin model and Self-assembly models [136] helped in refining complexation and templates.

3.2 The Beta Workbench

The Beta Workbench (BWB for short), is a set of tools to design, simulate and analyse models written in BlenX[7].

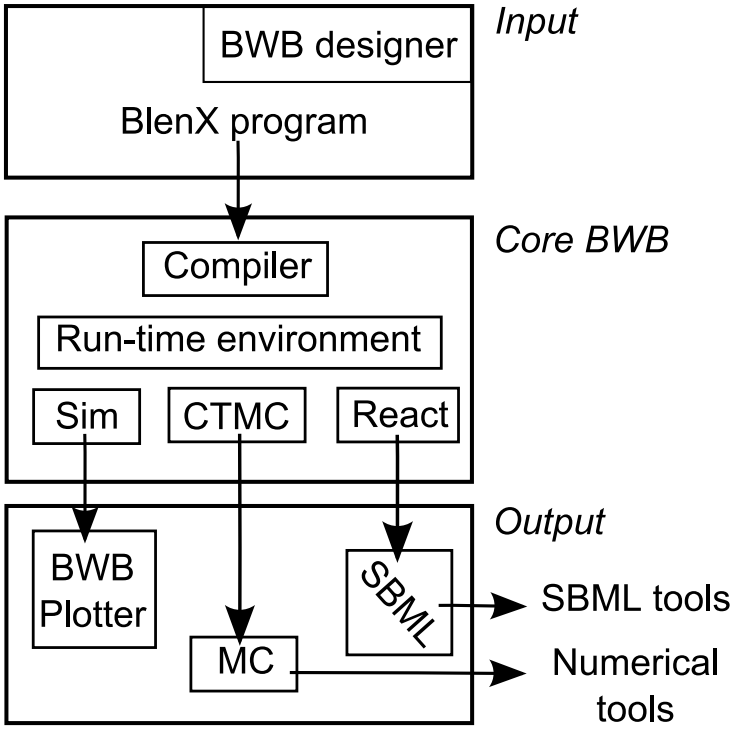


Figure 3.10: The logical strucure of BWB

The core of BWB is a command-line application (core BWB) that hosts three tools: the BWB *simulator*, the BWB *CTMC generator* and the BWB *reactions generator*. These three tools share the BlenX *compiler* and the BlenX *runtime environment*. The core BWB takes as input the text files that represent a BlenX program (see Sec. B), passes them to the *compiler* that translates these files into a runtime representation that is then stored into the *runtime environment*. The logical arrangement of the computational blocks above is depicted in Fig. 3.10.

3.2.1 BWB simulator

The BWB *simulator* is a *stochastic simulation engine*. The *runtime environment* provides the *stochastic simulation engine* with primitives for checking the current state of the system and for modifying it. The *stochastic simulation engine* drives the simulation handling

[7]BWB is available at `http://www.cosbi.eu/Rpty_Soft_BetaWB.php`

the time evolution of the environment in a stochastic way and preserving the semantics of the language. The stochastic simulation engine implements an efficient variant of the Gillespie's algorithms described in [84].

Stochastic rates

The BWB simulator computes the probability of an action to occur using the *reaction probability density function* introduced in Section 2.2.4. The way of computing the combinations, and consequently the *actual rate*, varies with the different kind of reactions we consider; here we list the possible reactions that can be used in BlenX, which are supported by the BWB simulator.

Rate of a *monomolecular* reaction:

the simplest kind of reactions we can encounter are first-order reactions, usually referred to as *monomolecular reactions*, that take the form:

$$S_1 \rightarrow S_2 ... S_n$$

In this case, the number of combinations h_μ is equal to n, where n is the number of entities (the *cardinality*) of S_1.

Rate of a *bimolecular* reaction:

second-order reactions, usually referred to as *bimolecular reactions*, take the form:

$$S_1 + S_2 \rightarrow S_3 ... S_n \quad \text{or} \quad S_1 + S_1 \rightarrow S_2 ... S_n$$

The second case explicitly consider the fact that the two elements reacting are indeed of the same species, as in homodimerization reactions.

To obtain h_μ, we have to compute the number of all possible interactions that can take place between elements of the first species and elements of the second species. Let n be the *cardinality* of the species S_1, and m the cardinality of the species S_2.

In the former case, the number of combinations h_μ is equal to $n \cdot m$, while in the latter the number of combinations h_μ is equal to $\frac{n \cdot (n-1)}{2}$.

Constant rates:

constant rates are used when the number of combinations h_μ is not meaningful; for example, consider zero-th order reactions like $Nil- > S_1$. In this case $h_\mu = 1$, so the base rate constant c_μ is directly used as the exponent of the exponential distribution form which a time of execution will be sampled.

Rate functions:

the numerical integration of the *reaction probability density function* has been proved by Gillespie to be *exact*: a Monte-Carlo simulation of the method represents a random walk that is an unbiased realization of the chemical master equation.

However, when a specie represents a higher aggregation entity (e.g. a cell) then the input-output relation can exhibit a non-linear behaviour (e.g. sigmoidal dose-responses for signalling molecules). In this case, we let the user specify a *rate functions*, that is used in place of the Gillespie method to compute the propensity function.

Note that in this case the proof that the method, and so the algorithm, is *exact* does not hold anymore. It is up to the user that chooses a rate function to demonstrate that the assumptions he/she made are realistic and that the produced results are correct. We are only providing the BlenX programmer with the highest flexibility in specifying the quantitative parameters that drive the simulation engine.

3.2.2 BWB CTMC and Reactions generators

When rates are drawn from an exponential distribution and models are finite-state, a BlenX program gives rise to a continuous-time Markov process (CTMC). The BWB *CTMC generator* adds to the core blocks a set of *iterators* to exhaustively traverse the whole state space of a BlenX program. The *CTMC generator* also labels all the transitions between states with their exponential rate.

The BWB *reactions generator* identifies state changes that can be performed by *entities* and *complexes* generated by the execution of a BlenX program and produces a description of the system as a list of *species* and a list of chemical reactions in which species are involved. These lists are abstracted as a digraph in which nodes represent species and edges represent reactions (see Fig. 3.11). This graph can be reduced to avoid presence of reactions with infinite rate. The final result is an SBML description of the original BlenX program (Fig. 3.12).

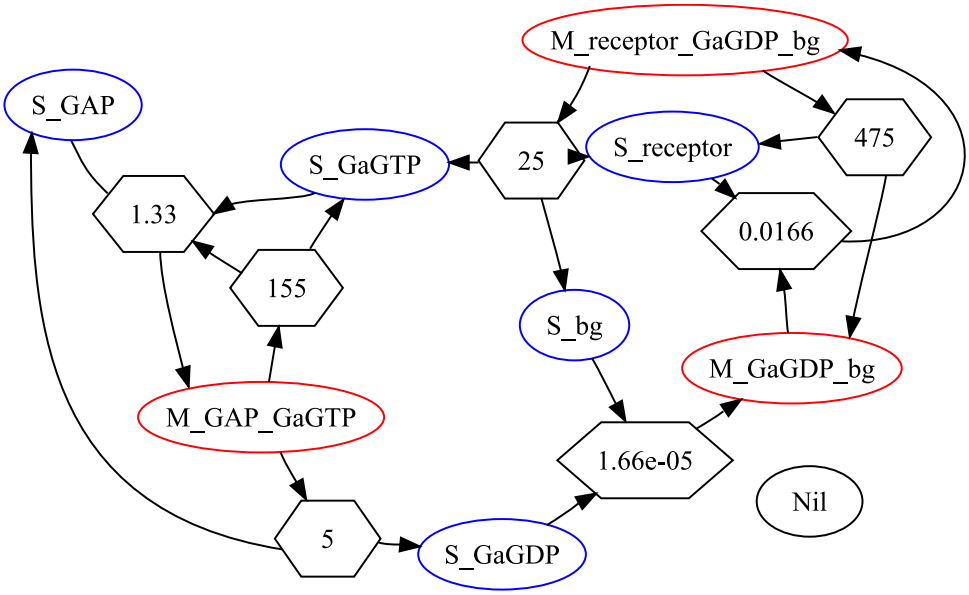
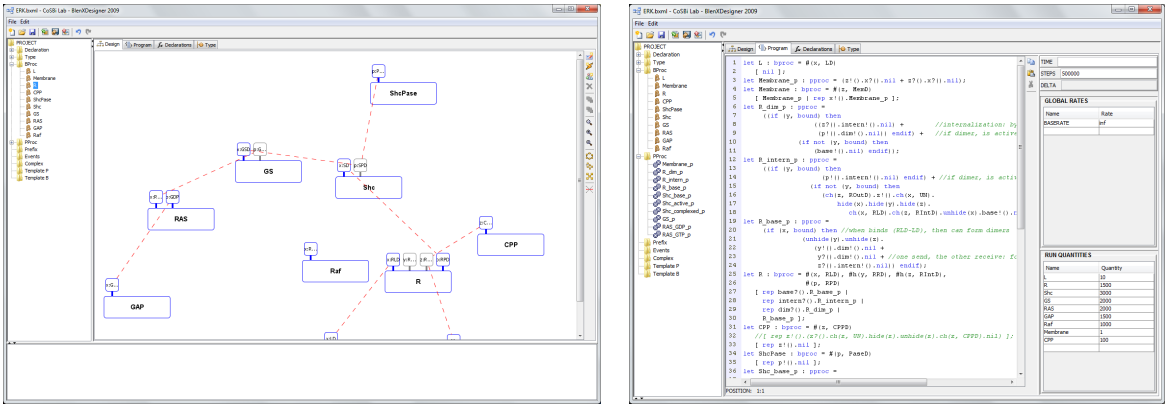


Figure 3.11: The graph of all the reactions generated by the BWB Reactions generator

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level1" level="1" version="1">
 <model name="SBMLmodel">
  <listOfSpecies>
   <specie name="S_GAP" compartment="compartment" initialAmount="206"/>
   <specie name="S_GaGTP" compartment="compartment" initialAmount="600"/>
   <specie name="S_GaGDP" compartment="compartment" initialAmount="600"/>
   <specie name="S_bg" compartment="compartment" initialAmount="1500"/>
   <specie name="S_receptor" compartment="compartment" initialAmount="313"/>
   <specie name="M_receptor_GaGDP_bg" compartment="compartment" initialAmount="0"/>
   <specie name="M_GaGDP_bg" compartment="compartment" initialAmount="1100"/>
   <specie name="M_GAP_GaGTP" compartment="compartment" initialAmount="0"/>
  </listOfSpecies>
  <listOfReactions>
   <reaction name="R0" reversible="false">
    <listOfReactants>
     <specieReference specie="M_GAP_GaGTP"/>
    </listOfReactants>
    <listOfProducts>
     <specieReference specie="S_GAP"/>
     <specieReference specie="S_GaGTP"/>
    </listOfProducts>
    <kineticLaw formula="M_GAP_GaGTP * c0">
     <listOfParameters>
      <parameter name="c0" value="155"/>
     </listOfParameters>
    </kineticLaw>
   </reaction>
   <reaction name="R1" reversible="false">
```

Figure 3.12: The SBML file generated by the BWB Reactions generator



(a) (b)

Figure 3.13: The model of the ERK pathway in the designer.

3.2.3 BWB designer and BWB plotter

The core BWB is enriched by two tools for input/output operations: The BWB designer and the BWB plotter.

The BWB designer[8] is a tool that allows to write BlenX programs both in a textual and in a graphical way. The two representations are interchangeable: the tool can parse and generate the graphical representation from any valid BlenX program, and generate the textual representation from the graphical form (see Fig. 3.13). In particular, it is possible to draw boxes, pi-processes, interactions, events and to form complexes using graphs (see Figures 3.14 and 3.15). The textual representation can then be used as input to the core BWB.

The BWB plotter is a graphical tool that parses and displays simulation outputs as changes in concentrations (Fig. 3.16), graphs of the reactions executed by the simulator (Fig. 3.17) and other views of the relations between entities and reactions. The BWB plotter provides to the user a picture of the dynamic behaviour of a simulated model and the topology of the network that originated that behaviour.

3.3 Related work and Future directions

Work on computational tools for systems biology is vast and diverse. We choose to focus on *executable models*, i.e. on the design of executable computer algorithms that mimic biological phenomena, for which a number of reviews and essays can be found [75, 130, 137].

[8]The designer for the second version of BlenX, shown in these pictures, was developed with Daniele Furlan from the University of Trento.

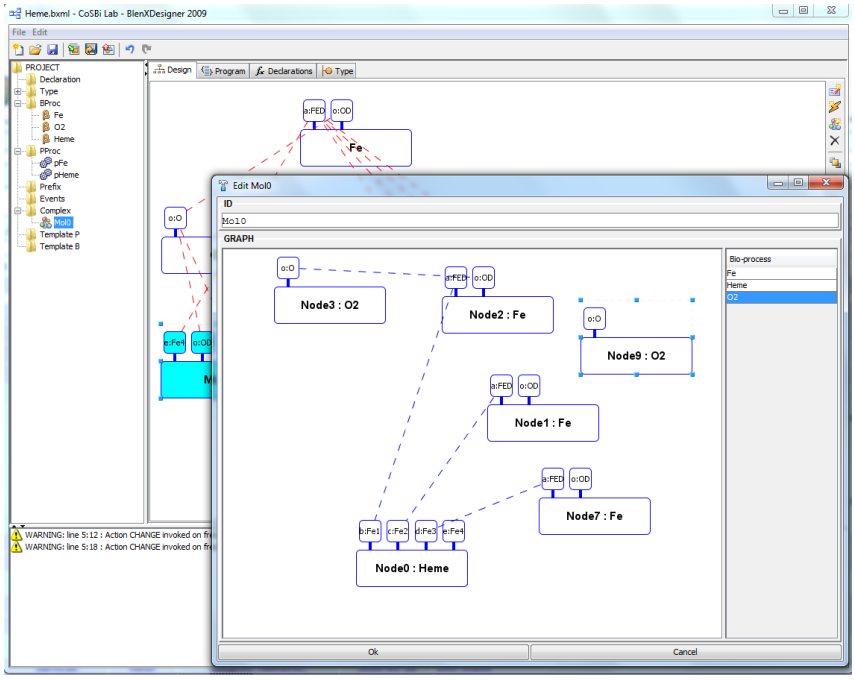


Figure 3.14: Definition of a complex through the Designer interface.

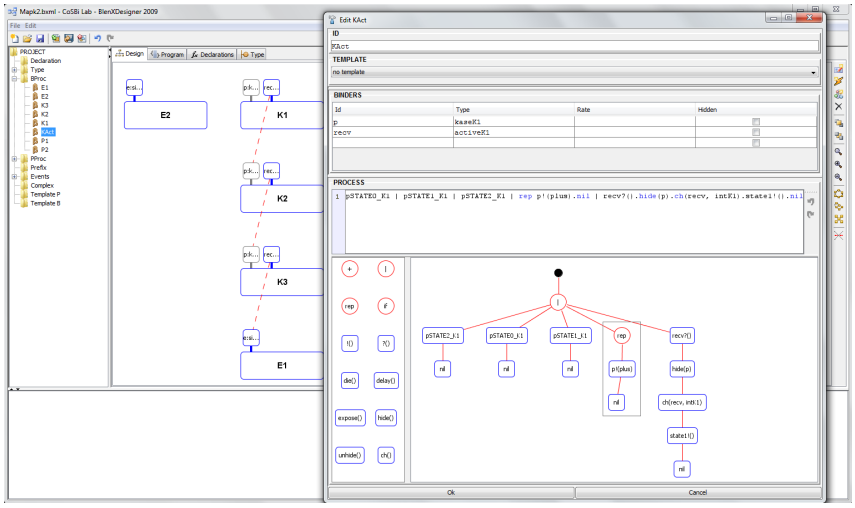


Figure 3.15: Definition of the internal process for a double-phosphorilated kinase.

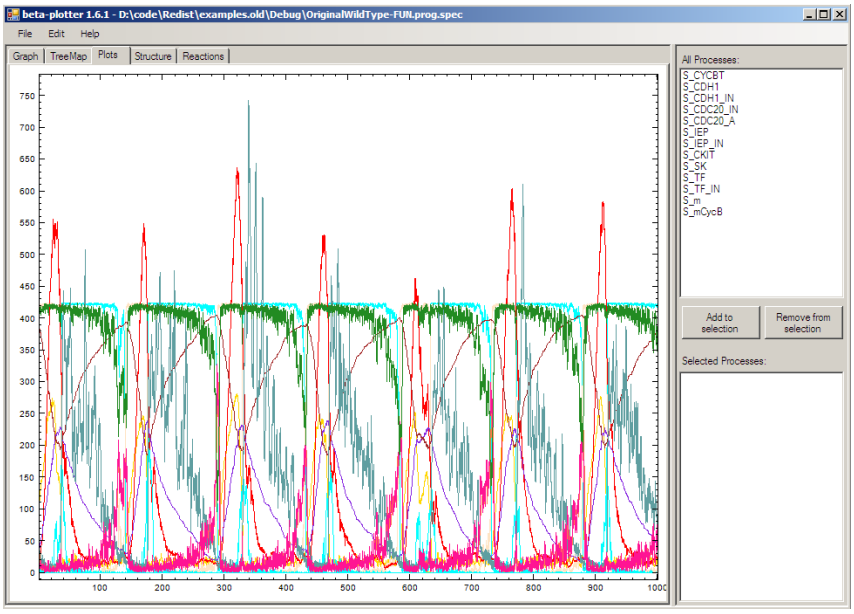


Figure 3.16: The Plotter displaying the result of a simulation of a BlenX Circadian Clock model.

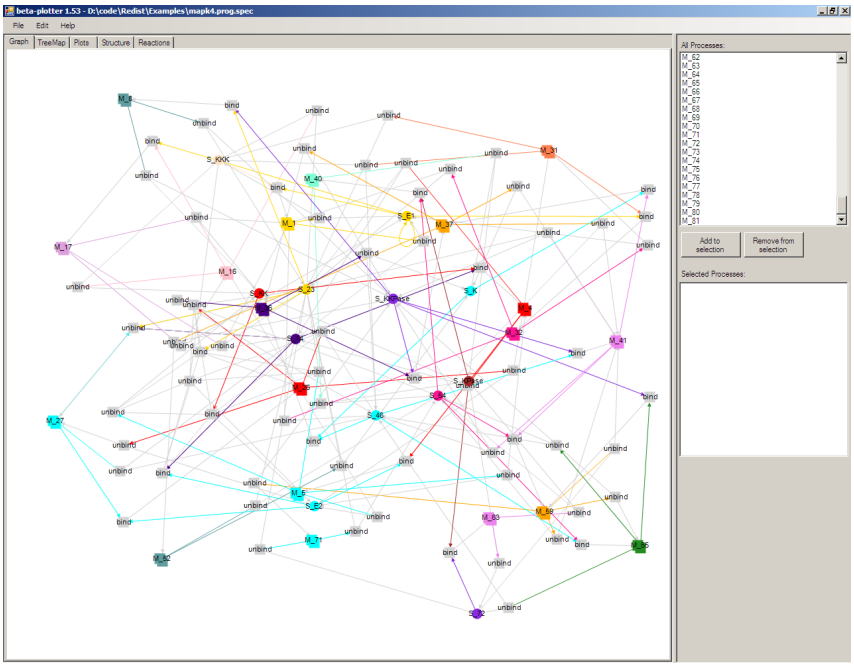


Figure 3.17: The Plotter displaying the graphs of reactions executed during a simulation.

Moreover, our focus is on *a language approach* for systems biology, specifically on features linked to scalability of those languages. Computational languages proved to be a general, reusable, powerful and scalable way of describing complex computations and interactions, thanks to features like modularity and abstraction we highlighted during our introduction of BlenX.

Relevant languages for systems biology were already introduced in Chapter 2; focused surveys can be already found in the literature: we advise the reader to refer to [94], [48] and [40].

In this section, we want to recall the formalisms and frameworks more closely related to BlenX and BetaWB, highlighting similarities and differences in the light of scalability issues.

SPiM, the Stochastic Pi Machine [171], is a programming language based on the pi-calculus. SPiM features a stochastic simulation machine based on the Gillespie SSA method (Sec. 2.2.4), and a simple graphical notation for modelling a range of biological systems [173]. The basic textual notation can be used to model large systems incrementally, by directly composing simpler models of subsystems. However, the same modularity issues we highlighted in our introduction of process algebras are true for the pi-calculus dialect used in SPiM: the language was designed to model computer systems, and therefore building some kind of biological interactions on top of it (e.g. complex formation) can be cumbersome. Similarly to BetaWB, SPiM provides also a set of tools for visualization and interpretation of the output data, as well as tools to import and export models in different format.

Bio-PEPA [41] is another language designed for modelling and analysis of biochemical networks. Like BlenX, Bio-PEPA is based on a process algebra (PEPA), and extends it in order to introduce features and characteristics needed to model in a closer way biochemical networks. Examples of these features are the support for different kinds of kinetic laws, which are very similar to BlenX functions, stoichiometry (a unique characteristic of Bio-PEPA) and SBML-events, an extension that was introduced following the same considerations we made for BlenX events. Therefore, Bio-PEPA allows to build models with different levels of abstraction.

Bio-PEPA can support different kinds of analysis, including stochastic simulation, analysis based on ordinary differential equations (ODEs) and model checking in PRISM, through the Bio-PEPA workbench. A mapping from the graphical notation SBGN-PD to Bio-PEPA is also available.

The k-calculus [49] is a language for the formal description of molecular biology. Like in BlenX, interactions are modelled at the domain level; however, bonds are represented by means of shared names, and reactions follow a rule-based approach instead of message

passing. In both cases it is possible to build models and complexes starting from components in a modular way. The differences between the two languages can be appreciated by examining how different solutions to the self-assembly of complexes, a process in which disordered components form an organized structure only through local interactions, are modelled in the k-calculus [46] and in BlenX [136].

Finally, the University of Rostock modelling and simulation group actively develops James II, a flexible, extensible and reusable simulation framework which supports many modelling formalism and many simulation methodologies. The same group designed and developed on top of JAMES simulators for the *Attributed π-calculus* [121], a variant of the pi-calculus where reaction rates can be made dependent on attributes that are assigned to processes, and for *SpacePi* [119], a variant of the pi-calculus that provides a way for individual based modelling of systems dependent on spatial and temporal interactions of individuals. Roehl et al. developed on top of James the concept of model components, which helps in building models from components, therefore sharing the goals of BlenX templates introduced in this chapter [195]. Realization is, however, quite different being based on a separate language (XML based) for the description of components and of their extensibility points. It would be interesting to compare and possibly merge ideas from the two approaches. Both the languages and the James framework shows concepts and practices that we are considering for our future research directions, which will be illustrated in the next section.

3.3.1 Next steps

Work on BlenX is far from being complete. As we mentioned in Section 3.1, the definition of BlenX has been an interactive process: as such, the language itself it's not static, but it is alive and continuously improved to meet the community requirements. BlenX limits arose during *on-field* tests, i.e. the creation of complex, real world models.

An example is the in-silico experiment we will describe in Chapter 7; that particular research topic was addressed using the in-silico experiment framework that will be introduced in Chapter 6. However, our initial question was whether it was possible to model the entire evolutionary framework in BlenX. It turned out that it was not possible due to some language limitations. Later, we realized that by performing some minor adjustment to the language a whole new class of evolutionary computations, which included our initial scenario, could be expressed in BlenX.

On a related note, work done on the research on Horizontal Gene Transfer (HGT) highlighted similar problems. In this case, it was possible to build a limited HGT model using BlenX, but the model was not flexible enough to allow for the kind of in-silico experiments we had in mind. An ad-hoc solution proved to be more convenient.

Finally, the current version of BlenX has no explicit support for spatial characteristics. As we have seen in Chapters 1 and 2 spatial aspects are very important in many biological systems.

One of the promises of beta-binders was to allow for a hierarchical spatial model even without the nesting of beta-processes [181]. This hierarchical model could be obtained thanks to the use of binders and affinity functions: with each beta-process representing a compartment, and pi-processes the species in that compartment, affinity functions governed the ability of communicating with other compartments, defining a logical "neighbourhood" and "containment" relationships. Affinity functions had to be created so that logical relationships reflected the real, physical ones. However, defining spatial relations in this way was cumbersome and started to collide with the notion -which was becoming dominant, and later was adopted by BlenX- of one beta-process as one biological entity.

As a solution, an extension to beta-binders was proposed by Romanel et. al. [95]; in this extension, every beta-process was labelled with an index reflecting the position of the process inside a tree-structure, representing the nesting of compartments. Actions were added to allow for movement of processes, and the affinity relation update to taken in consideration the spatial relationships.

However, this solution was never incorporated into BlenX. As Larcher demonstrated with the NfKB model [135], it is possible to encode space in the model using types, like in beta-binders. However this encoding is not always simple, and can be cumbersome if more than a couple of species or locations are involved.

The following sections contains my opinion, maturated during several meetings and fruitful discussion with other CoSBi researchers, on how the current version of BlenX could be evolved to address these shortcomings.

Richer types

The notion of typed interaction sites proved to be one of the winning points of BlenX. It helps composability through isolation, it can be used to closely reflect many biological concepts, and it allowed the introduction of complexes as first-class citizens. Furthermore, the same notion was later implemented by other bio-inspired languages, like Kappa [50], proving the wisdom of this approach.

However the current implementation of binders is, in my opinion, too weak. Right now the *binder identifier*, i.e. the type associated to a binder, is only a name, an identifier, that is used to look-up an affinity measure in the affinity table. The affinity function itself, in fact, is an explicit list of affinity values described in a set-theoretic way; this is handy in many situations, but expressing the function in a computable way, using formulas or algorithms, is an invaluable help in many cases. Moreover, they could make

models *self-contained*, eliminating the need for external programs to compute the affinity function/table. The limitations showed by in-silico evolution, HGT and modelling of space can all be overcome by using more powerful types and affinity functions.

Let introduce an example, derived from evolutionary computation. As we will see in Chapter 7, mutations at DNA level are reflected in many cases on the domain sites of a protein. This kind of mutation can be implemented by changing the type of a binder. But then, which affinities the new type should have? Should they be copied from the original ones, with random modifications? Which relations have to be kept, and which have to be dropped?

If we could compute the affinities directly from the types, the problem would not arise. Let's represent domains using a simplified model, using 0-1 strings of length 5:

```
let Domain : type = boolean[5];
```

and use a compatibility function based on complementarity:

```
let f(a,b) : function = //function for complementarity
let affinity(Domain d1, Domain d2) = f(d1,d2);
```

The function is written so that complementary domains are fully compatible, while domains with some differences can still match in a weaker way: for example, two Domains `[0,1,1,1,0]` and `[1,0,0,0,1]` will have an affinity of 1, while `[0,1,0,1,0]` and `[0,1,0,0,1]` will have an affinity of 0.4, for example.

Processes and boxes will use binders as before, with the addition of functionalities to read and write binder values:

```
let Protein : bproc = #(x, A:Domain), #(y, B:Domain)
   [ ...x?().ch(A@1, A@1 xor RandBool())... ];
```

Channels can be also augmented with types, acquiring the ability to carry messages of a specific type:

```
let threshold : const = 0.5;
let P : pproc =
   mutate?(x).if (RandDouble() > threshold) mutate!(x xor RandBool()) else mutate!(x);
```

where the `mutate` channel will carry boolean values. The mutation process can then be reused inside boxes, where it is used to mutate the box own binder types in response to some event:

```
let Protein : bproc = #(x, A:Domain), #(y, B:Domain)
   [ P |
     ...x?().mutate!(A@1).mutate?(value).ch(A@1, value).
             mutate!(A@2).mutate?(value).ch(A@2, value)....];
```

As another example, consider spatial relationships. As we have seen in Chapter 2, some methods, in particular species based methods, use a discrete mesh structure to define

space. Consider for example a model for micro-tubules: the movement of molecules along the microtubule can be modelled as discrete movements along a 1D axis.

With richer types, we can devote the *value* of a binder to keep track of the space, while still retaining its functionality as a communication channel:

```
let SpaceBinder : type = int;
```

or, if we want to give a limit to the number of discrete grid positions:

```
let SpaceBinder : type = (0..10);
```

Translocation can be represented as a stochastic process:

```
template moveLeft<<binder B, rate l>> = ch(l, B, B - 1);
template moveRight<<binder B, rate r>> = ch(r, B, B + 1);
let Protein : bproc = #(x, A:SpaceBinder)
   [ ...
     | moveLeft<<A, 0.05>>
     | moveRight<<A, 0.1>>
   ];
```

Communication (interaction) can obviously only take place only between proteins in the same space; in this case, an affinity function can be defined as:

```
let rate1 : const = 1.0;
let rate2 : const = 0.0;
let f2(a,b) : function = if (a==b) then rate1 else rate2;
let affinity(SpaceBinder d1, SpaceBinder d2) = f2(d1,d2);
```

Affinity functions

One aspect that it is important to consider is the *expressive power* and *computability* of affinity functions.

Other calculi, use lambda calculus to express conditional events (like the original beta-binders with its join and split functions) or communication constraints (like the Attributed Pi-calculus [121]). In this calculi, a lambda function is applied to state-dependent arguments (binders and processes in the first case, attributes in the latter); the result of the evaluation is used to choose whether the communication should happen or not.

In Beta-binders, this led to inefficiencies in the simulation. In the case of the Attributed pi-calculus, it has been shown that

language is a call-by-value λ-calculus)

When we limit in-silico experiments to simulation, ... he language to express affinity functions should not be too powerful. In order to guarantee some aspects of the simulator (like termination), to maintain the language tractable, and to make model execution efficient, the language for affinity functions should not be Turing-complete.

After some initial considerations, I believe that the direction is to have simple mathematical and boolean expressions, with no recursion (and no *while* commands or equivalent construct). To manipulate data types that consist of more than one entry, like in the `Domain` binder type of the previous example, a *bounded for* or *foreach* construct can help. Notice that as long as we allow only for finite or immutable *lists* or *sets* of elements, many language properties should still be decidable.

Another way may be to introduce some pre-defined functions, for which we already proved termination and other properties of interest, that use expressions (or *pure functions*) as an input parameter. Notice that my proposal is not to introduce pure functions as a first class construct, nor to add higher-order functions across the language. For simplicity, the only higher-order functions are the intrinsic *special functions*. This means that functions (or expressions, or operators) are only allowed as an input parameter of these special functions:

```
let f (x, y) : function ...
let r (t1, t2) :
   fold f t
   map f t
   filter f t
   apply f t
   append t1 t2
   zipwith f t1 t2
```

In this case, the *function for complementarity* between Domains mentioned above can be expressed as:

```
let f(a,b) : function = fold(+, zipwith(xor, a, b)) / length(a);
let affinity(Domain d1, Domain d2) = f(d1,d2);
```

The affinity function uses the *zipwith* intrinsic function to apply the *xor* operator to each element of the Domains, then it will sum them (using the *fold* intrinsic function and the + operator) and divide the sum by the Domain length. For example, the application of *zipwith* with *xor* on `[0,1,0,1,0]` and `[0,1,0,0,1]` produces `[0,0,0,1,1]`, map with + leads to 2, divided by $5 = 0.4$.

A final consideration about species: the structural congruence have to be updated. We can still retain the notion of species based on structural congruence, as it will work for the examined examples. For example, the introduction of a numeric SpaceBinder to encode a position in space effectively separates entities of the same species but located in a different position on the grid, as the spatial Gillespie method requires (see Section 2.2).

Moreover, if we substitute the integer, discrete location with a real floating-point value (or better, with a pair or triple of real values representing Cartesian coordinates), each entity will belong to a different species[9], making the language suitable for the description

[9]Assuming that there will not be two entities having the same exact location

of individual based systems. In this way, it should be possible to emulate most of the concepts of languages with an explicit notion of space (like SpacePi [120]). Of course, this approach requires different simulation methods, like the BD methods introduced in Section 2.2.

In conclusion, the extension of types and affinity between types in BlenX is an area where much work can be done: some work has already started (see for example Romanel et al. [185]), but many of these ideas require further research. In particular, the language for the definition of affinity functions has to be defined and studied in a comprehensive way, in order to verify and prove which kind of functions it is possible to express and which properties are decidable and at which cost. My opinion is that this language should represent a compromise, keeping many desirable properties (i.e. termination) decidable, while allowing for some interesting functions to be computed.

The price to pay is that static analysis, compiler optimizations, and, in general, proving properties on models will become more difficult tasks in comparison with the current version of BlenX, but we should avoid to make them impossible.

3.4 Summary

In this chapter we have seen that a programming language can be used to model biological entities by writing compact textual code, which let us build very complex models in a more manageable way.

In particular, we introduced BlenX, a language for Systems Biology, and we focused on the aspects of BlenX that allows to build composable and scalable models: modular composition through affinity-based interaction, reuse through patterns and templates, and abstraction. Work on BlenX is still active: its usage with realistic models as well as comparison with other languages gave us ideas for future extensions of the language, aimed at increasing its modularity. Our goal is to create a language that simplifies modelling of large scale systems by enabling seamless composition of simpler, smaller and/or abstract parts.

Chapter 4

Simulation

The Gillespie algorithm (Sec. 2.2.4, [84]) is the most widely used and known algorithm in systems biology; thanks to this stochastic method, it is possible to perform accurate and physically correct simulation of realistic biochemical systems.

However, more complex and large models need to cope with spatial aspects and movement of particles, which are not considered by the Gillespie SSA. *Space* is becoming a very important aspect in model simulation.

Furthermore, the Gillespie algorithm is inherently sequential. If we want to deliver the promise done by systems biology to be able to understand a system as whole, we also need to move from sequential to *parallel simulation* algorithms. Models could possibly be very large in order to capture the systems behaviour in a reliable way. This is particularly true in the case of *spatial models*, which add a new degree of complexity to simulation methods, as we shall see in this chapter. Algorithms that can leverage parallel architectures will give modellers a great advantage.

This chapter will treat the problem of execution and simulation of large models. We will see how spatial aspects need to be taken into account when the scale moves up from the level of detail of a single pathway, and how parallel computing can speed up the simulation of spatial methods.

4.1 Space

When studying a single localized pathway, the macroscopic description of its kinetics usually suffices. However, many biological processes are not local and they often take place in an *inhomogeneous* medium, the *cytosol*, where spatially localized fluctuations of inorganic catalysts and intracellular diffusion can have an important role.

Space and variations in concentration of biochemical entities play e central role in many interesting biological processes, including mRNA movement within the cytoplasm [79],

ASH1 mRNA localization in budding yeast [4], morphogen gradients – for example, across egg-polarity genes in Drosophyla oocyte [4] –, bacterial chemotaxis [209], synapse-specificity of long-term facilitation in neural cells [124], and microtubule assisted protein movement during the cell cycle [162]. When dealing with such processes, it is necessary to explicitly consider the cell geometry and, in general, spatial conformation and diffusion processes.

As we have seen in Chapter 2, stochastic simulation of biochemical models is a fundamental analysis tool for systems biology: models allow researchers to quickly replicate experimental conditions in-silico, that can be tested by simulation. However, many stochastic methods make some assumptions about the reaction volume and about the representation of molecules.

For instance, SSA assumes that a system is a well-stirred mixture of molecules (see Section 2.2.4); all the entities of the same species are treated in the same way: if the well-stirred mixture assumption holds, it means that we can consider *diffusion* inside the reaction volume as *instantaneous*. Therefore, the probability of any of two molecules of one species to meet (and react) with any two molecule of another species is the same. Obviously, the consequence is that we need to consider *species* or classes of molecules instead of single molecules; this is good with respect to performance, but it has drawbacks when considering inhomogeneous systems.

We presented some stochastic methods able to deal with spatial information and diffusion of entities in Section 2.2. Here, we recall briefly two ways of overcoming the limitations of Gillespie-like methods: choose a different level of abstraction, using single molecular detail, and therefore a different method not based on the well-stirred assumption, or adapt the existing methods, making the assumption co-exists with diffusion and space.

Before proceeding further, we briefly review the concepts of diffusive flux and Fick's law [74]. The analytical description of diffusion is summarized by the definition of *diffusive flux*, the number of molecules which pass through a small surface S per unit of area per unit of time. J, the net flux of a solute, depends on the number of molecules passing through either side of the surface: if there are more molecules on the left, then we expect a left-to-right flux which grows in size as the difference of concentration between the two sides of the surface increases. As this local difference varies from one point in space to another, the flux is a vectorial quantity depending on the position in space. The simplest description of the flux as quantity dependent on the concentration of species is the *Fick's first law*, which states that the flux is proportional to the local derivative of

the concentration c of solute with respect to the spatial variables:

$$\vec{J_B} = -D_B \nabla c_B$$

where B is some generic chemical species, $J_B = J_B(x, y, z)$ is the local flux of B molecules, c_B is the local concentration of B, and D_B is the *diffusion coefficient* for B. The *diffusion coefficient* includes the parameters driving the diffusion dynamics.

4.1.1 Particle-based methods

Particle based methods consider each molecule in the system as an individual entity that has to be simulated and analysed. Therefore, each entity in the system has attributes, like position, kind of molecule and velocity. These attributes are used to compute the movements and the reactions inside the system for each individual; the actual computation and the attributes used and updated by the system vary from one method to another (see Sec. 2.2).

We will focus on Brownian Dynamics (BD) methods, where each molecule is represented as a point-like particle with continuous x, y and z coordinates, that diffuses freely in space following Fick's first law. No assumption is made about the distribution in space of the various molecules; these methods can simulate localized fluctuations, gradients, local events without any adjustment.

A drawback of particle-based methods is represented by execution speed, because the speed of execution is influence by both the number of species and of reactions in the systems and by the number of individuals for each species. Therefore the usage of particle-based methods, even of faster, more high-level methods like BD ones, is historically seen as impractical for the simulation of large systems.

4.1.2 Species-based methods: spatial Gillespie and NSM

As we mentioned, the most diffused mesoscopic level simulation algorithms for intracellular stochastic kinetics are based on the premise that diffusion is so fast that the concentrations of all the involved species are homogeneous in space. However, recent experimental measurements of intracellular diffusion constants indicate that the assumption of a homogeneous well-stirred cytosol is not necessarily valid even for small prokaryotic cells.

Reaction-diffusion methods, as introduced in Section 2.2, are based on a discretization of the simulated volume: the system spatial domain is divided into a number of reaction chambers, called *cells*, *voxels* or *sub-volumes*, small enough for the assumption of a homogeneous medium to hold. As their names indicates, reaction-diffusion models consist of

two components. The first is a set of biochemical reactions which produce, transforms or removes chemical species. The second component is used to model diffusion as the movement of species between reaction chambers; in particular, the movement of a molecule M from sub-volume i to sub-volume j is represented as a first order reaction $M_i \rightarrow M_j$, where d is the *diffusion coefficient*.

Assuming that both reaction rates and diffusion coefficients are given, reaction-diffusion systems can be simulated using the Gillespie SSA, as demonstrated by Bernstein et al. [14].

A more efficient alternative is represented by the *Next Sub-volume Method* (NSM) proposed by Elf et al. [65]. NSM is a two level method in which every cell computes individually the next event, a chemical reaction or a diffusion, using the Gillespie direct method [84]. The cell where the next event will occur is determined using a global priority queue holding the reaction times of the quickest reaction event for each cell. More in detail:

- Initialization

  1. Distribute the initial numbers of molecules between the cells and store in a *configuration matrix*. This can be done randomly or according to any initial distribution.
  2. Calculate the sum of reaction rates r_i for each cell i and store in the rate matrix. The reaction rates are calculated for the size Δ of the cell, according to the *reaction-diffusion master equation* [15].
  3. Calculate the sum of diffusion rates for each cell, store it in the rate matrix.
  4. For each cell i, calculate the first event (diffusion or reaction) time $T(i) = \frac{ln(rand)}{\sum_{events} r_i + s_i}$
  5. Make an initial ordering of the cells according to their next event times. The cells are kept sorted in a priority queue.

- Iterations

  1. Assume that c is the cell in which the next event occurs at time $T(c) = \tau$ according to the top element of the priority queue. Generate a random number rand uniformly distributed between 0 and 1, choose a Reaction Event if $rand < r_c/(r_c + s_c)$ and otherwise a Diffusion Event.
  2. Reaction Event:
     (a) Rescale rand from the previous point to [0, 1] to determine which reaction occurred as in the direct method.
     (b) Update the state of the cell c in the configuration matrix according to chosen reaction

(c) Recalculate the $\sum r_c + s_c$ for the cell c, calculate the time of the next event.

(d) Insert the new event time of cell c in the event queue and order the queue

3. Diffusion event:

(a) Update the states of both cell c and its neighbour, d, which got an additional molecule.

(b) Recalculate the sums, sample the time to the next event in the cells

(c) Insert the new event times in the event queue

4.2 Processing speed: Parallel Execution

The interest in concurrent and high-performance computing for computational and systems biology has grown steadily in the last years. When my PhD started three years ago, only a handful of groups were working on this subject. Soon after, we assisted at the dawn of the so-called *concurrency revolution*: researchers in life-sciences and developers started to understand what computer scientists already knew: concurrent software was becoming the main way to speed-up computations. When using the same abstraction, and algorithms with the same complexity, concurrent execution the only way of scaling up the size of research problems, and ultimately one of the few viable ways to fully understand large systems.

In the past years, in order to run an application faster on a new computer, the programmers had to do little or nothing: some tweaks could be done, maybe in assembly language or by recompiling the source code with a new compiler, in order to take advantage of the new features available with each generation of new processors. Running the existing program on a new processor of the same family required no change, and gave big speed-ups for free. This progress was possible thanks to the refinement to both the electronics technology, which allowed for smaller circuits, and the inner processor architecture. Smaller circuits meant less current drawn, and this allowed for astonishing increases in the CPU clock, the frequency at which a processor is able to process and execute instructions. It also meant that more transistors could be packaged inside the same chip, resulting in more complex architectures: features like superscalar processing, branch prediction, multiple cache levels, out-of-order and speculative execution were gradually introduced by Intel in the 4th, 5th and 6th generation of their PC processors [44]. All these improvements resulted in faster and faster CPUs, but it was mainly the increase of frequency to drive their performances. Since the introduction of micro-chips, manufacturers were able to design and introduce a new generation of CPUs that was two times faster than the previous one roughly every 18 months.

This rate of growth is often referred to as *Moore's law*, from a famous quote of the Intel co-founder Gordon Moore. Moore made the empirical observation that the number of transistor packed into a chip is expected to double every 18 months. This is only an empirical observation, but to date it has never been disobeyed. Since the raw processing power is strictly linked to the number of transistors, the law suggests also that the processing power and computing speed also are doubling every 18 months. In the past this growth of power was easily linked in an apparent way to the increase in CPU frequency and also, in a less apparent way, to the number of instructions processed per clock cycle due to architecture improvements. However, staring from 2004 onwards, new processors started not to follow this trend anymore (see Figure 4.1). Physical limitations started to disallow increase in power and frequency, and the processor architecture could not be revolutionized again, but only improved.

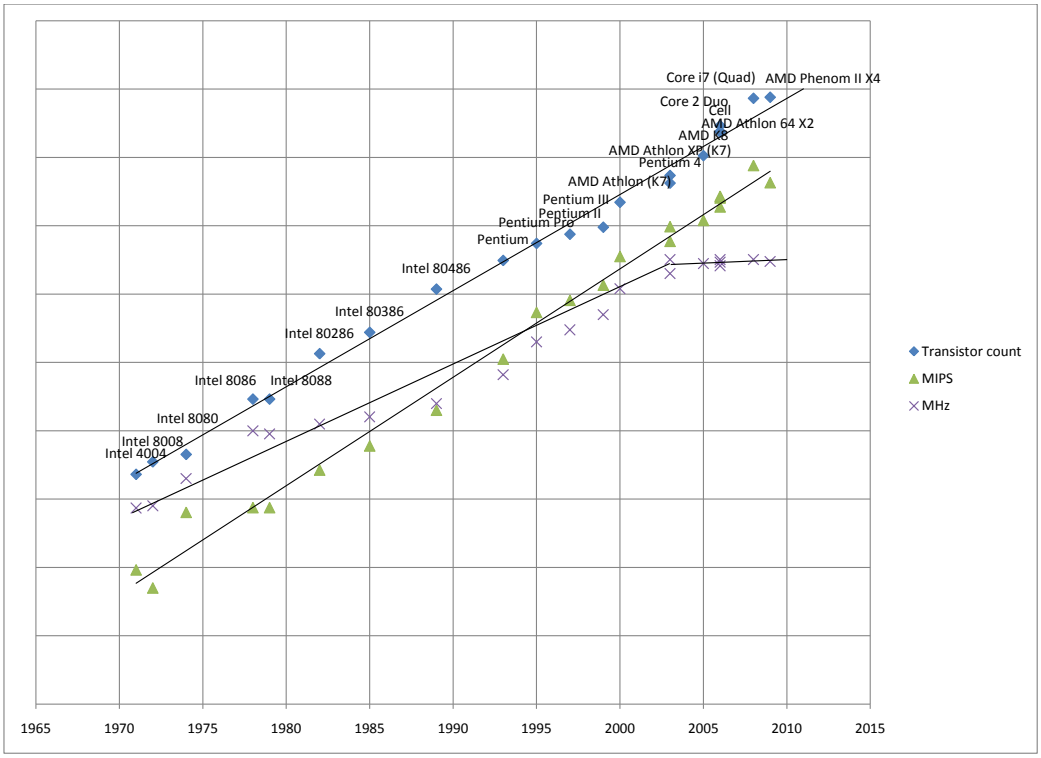


Figure 4.1: Moore's law: IPS (Instructions Per Second), numbers of transistors and frequency: from 2004 onwards the last one is not growing anymore.

The solution was to keep packing more transistors, by inserting more *cores*, i.e. more

execution units, inside the same chip. In this way, raw performances and processing power is still doubling every couple of years by doubling the number of cores. This means that it is no longer possible to run a program on the newest CPUs and obtain an increment in performance: having many cores inside a chip requires not only to recompile a program, but to rewrite it and rethink it completely. Quoting Herb Sutter *"The free lunch is over"*: programming multi-core CPUs requires a paradigm shift, from sequential to parallel programming [219].

Box 2 The difference between Parallel and Concurrent

Parallel and Concurrent are commonly used in an interchangeable way. Concurrent derives from the Latin verb *concurrere*, which means run together, make the same thing, operate at the same time. Parallel derives from the Greek *parallelos*, "beside one another". The difference between the two is tiny, and often negligible.

However the difference is sometimes significant; in our case it is advisable to highlight the differences that arise when these words are coupled with words like *computing*, *program* or *language*.

A concurrent language, for example, means that the language has some constructs to make program statements or functions "operate at the same time". A parallel program, instead, is a computer program designed to run in parallel, i.e. as a collection of computational processes running "beside one another". Concurrent computing therefore is more about the ability of reasoning about a program as a set of intercommunicating processes, and compose them using ad-hoc constructs (like BlenX, in our specific case); on the other hand, parallel computing is more about the ability of running a program on multiple processors at the same time: you can have simple, sequential programs running in parallel (like in MRIPs, as we shall see later in this chapter), and concurrent programs running in a sequential way (think about BlenX or Concurrent ML).

While it is important to know the difference, it is so small that using them as synonyms is not a mistake, especially when there is no possibility of misunderstanding.

Nonetheless, even if concurrent computing needs to become mainstream, it is still too difficult: developing concurrent applications requires a deep knowledge of both the application domain and of the tools and methods for parallel programming.

A correct parallel implementation of any existing method or problem requires to consider four aspects: (i) the best computational splitting policy; (ii) how to handle synchronization among the computational workers, (iii) the more suitable hardware architecture and software packages to use and, above all, (iv) the nature of the inherent parallelism.

Some problems are naturally *parallelizable* while others are purely *serial*. Parallelizable methods can fall under two great umbrellas: *task parallel* and *data parallel* applications.

Following Dematté et al. [54], we will examine which parallel programming techniques can be applied to stochastic simulation methods.

4.2.1 Concurrent Monte Carlo simulations

To enhance the efficiency of Monte Carlo simulations, two computational paradigms were widely studies in the past: Single Replication in Parallel (SRIP) and Multiple Replications

in Parallel (MRIP).

Single Replication in Parallel. The SRIP approach is based on the decomposition of a stochastic trajectory into logical processes, running on different processors and communicating by means of message passing protocols [78]. For naturally divisible problems, it shows elevated performances in speed-up and scale-up benchmarks. Significant drawbacks originate from the necessity for warranty of synchronism.

Multiple Replications in Parallel. The method speeds up simulation by launching independent replications on multiple computers and using different random seeds in such a way that the processes are approximatively uncorrelated. Therefore, more observations can be collected during a given time interval than running a single replication on one computer within the same period of time. Traditionally, one runs a simulation for a fixed time and then performs the data analysis [101]. When the accuracy defined by the user is reached, the simulation stops and a confidence interval is generated. If the number of processes, the length of each replication and the deletion period are carefully chosen, the statistics will be valid [69]. In contrast to SRIP, MRIP can be easily applicable to any system, independent of the inherent system parallelism. However, the fact that a single replication cannot be executed on a unique processor and that outputs (or pieces of them) almost deterministic are identical when replicated, make the use of MRIP approaches sometimes inappropriate [87]. The MRIP and SRIP approaches are not exclusive, i.e., it is possible to use MRIP and SRIP in the same simulation program.

In biology, whereas the MRIP policy, well understood and investigated for a long time [16, 69, 86, 87, 88, 113, 146, 158, 217, 231], finds straightforward application to real case-studies [22, 224], the SRIP policy has a rather vague characterization. SRIP methods can be further divided into two opposite sub-categories which include: (a) methods that exploit *data-parallelism* (or *loop-level parallelism*), namely that speed-up simulation of interacting particles on a finite grid in which individual processors are in charge of simulating the state of each site [206]; (b) methods that exploit *task-parallelism* (or *functional parallelism*), namely that divide the computation of a realization into a set of sub-computations among cooperative processors by computational dependency criteria [78, 151] (See Fig. 4.2 for a compact view of the parallel paradigms just described).

To date, the research in distributed-parallel processing has successfully solved many related problems; parallel computing has been applied successfully to the field of computational and systems biology too (for a survey on the various methods and techniques applied to this field of study, see Mazza et al. [11]).

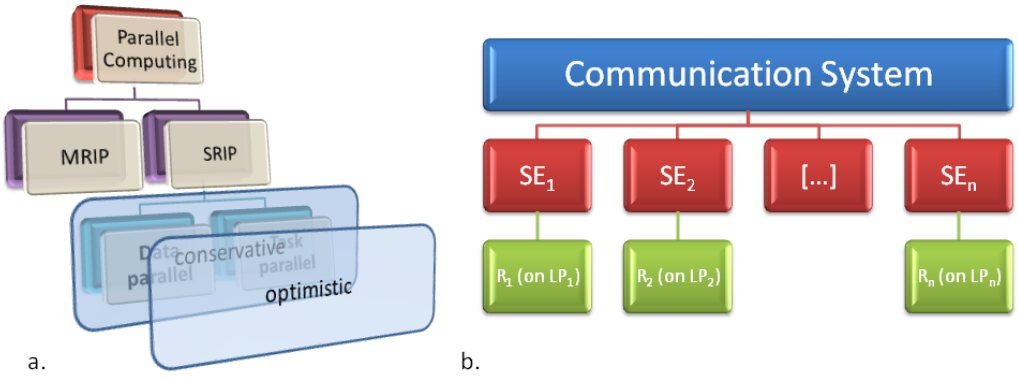


Figure 4.2: a. Parallel paradigms hierarchy. b. Model partitioning structure into Logical Processes and Simulation Engines

Unfortunately, up to now little attention has been paid to the parallelization of *stochastic* simulations with respect to the SRIP policy, especially for species-based models (essentially, the Gillespie SSA and derived methods). A notable effort is the pioneering work of Schwehm et al.[206]; unfortunately, the algorithm reached a good maturation only recently, with the work of Jeschke et al. [118].

The situation is brighter when we look at individual-based models; Tomita with the E-Cell project is an excellent example of how individual-based techniques can scale very well [218].

In this thesis, we present our investigation of parallel techniques applied to stochastic simulations of both species-based models and of individual-based models. In the first case, we aim at explaining why species-based models based on the Gillespie SSA are difficult to parallelise, driving the reader through the theoretical basis and strategic decisions that influenced our reasoning and design.

In the second case, we wanted to explore the usage of unconventional architectures, and exploit the computational power of Graphic Processors (GPUs)[1] to run simulations based on an existing individual-based methods orders of magnitude faster.

We will briefly introduce the category of computer-simulation systems known as Discrete Event Simulation (DES) and the work done on these systems in the light of parallel and distributed computing. We will show how the Gillespie SSA introduces in Section 2.2.4 can be reformulated in term of a DES system, and we will show the characteristics assumed by the algorithm when it runs in a parallel environment.

[1]The class of hardware processors known as GPUs, is presented in Appendix C, were we present their characteristics, architecture and programming philosophy.

4.2.2 Discrete Event Simulation (DES)

In DES, the life of a *system* is modelled as a sequence of timed events. With this approach, a system is set up by a collection of *processes* $P = \{p_1, p_2, \ldots\}$ and of *activities* or *events*[2] $E = \{e_1, e_2, \ldots\}$. A DES *process* is fully characterized by a finite set of *states* $S = \{s, s', \ldots\}$. At any given time, each process has exactly one *active state*. Each state s has a set of *actions* $A_s = \{\alpha_s, \alpha'_s, \ldots\}$ that can be performed when the process is in that state; the aim of an action is to change the current active state. *Activities* or *events* are sets of actions that are executed together to transform the state of the system (see Figure 4.3). Here, we refer to the state of a system z as the collection of all the active states of the processes in the system. A *run* is thus meant as a sequence of interleaved system states and events: $r : z_0|e_0 \rightarrow z_1|e_1 \rightarrow z_2|e_2 \ldots z_{(u-1)}|e_{(u-1)} \rightarrow z_u$.

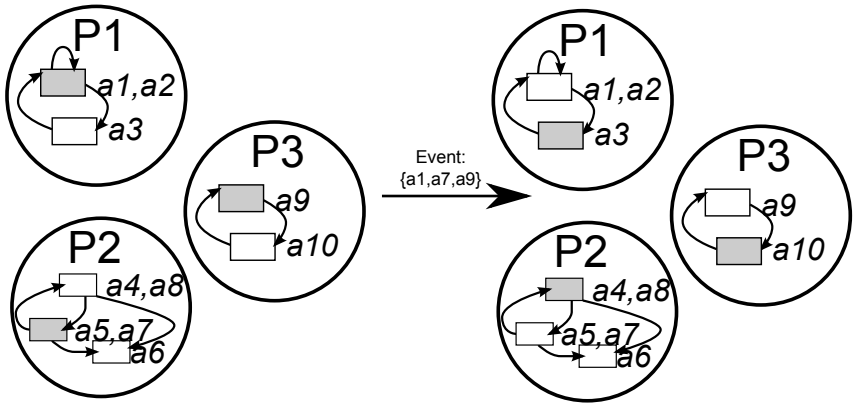


Figure 4.3: DES terminology: each process (circles) has various states (squares - the active one in grey), each with a set of actions (listed next to them). An event is a collection of actions which are executed to bring a system from one state (left) to the next one (right).

As opposed to continuous simulation, in discrete event simulation state changes of the simulated system are assumed to happen at discrete points of the virtual time and are thus controlled by non-continuous functions, resulting in a succession of *events*.

DES can be used to simulate stochastic processes. In a stochastic process, each state is partially but not fully determined by the previous one. Typically, a stochastic process can have one or more deterministic *arguments*[3] and their values range over an index collection of non-deterministic random variables X_i with certain probability distributions. Such functions are equally known as *realisations* or *simple paths*. The view of a stochastic process as an indexed collection of random variables is the most common one. The *events*

[2] *DES Events* should not be confused with BlenX Events; same for processes, actions and states. The concepts are very similar, but the terminology is used here in a more general way. This section apply to simulations in general, not only to simulation of process algebra models

[3] we consider the *time* as always present among the arguments.

to be executed are bound to the set of random variables X_i, that determine which event will be executed and when. A simulation executes events in non-decreasing time-stamp order so that the virtual time (the time-stamp on the last executed event) never decreases. Indeed, the occurrence of an event typically causes four actions: (a) progression of the virtual time to the time-stamp of the simulated event; (b) changes of the state of the simulated system; (c) scheduling of new events and (d) unscheduling of other events. Thus the basic data structure of a DES program consists of: (i) a virtual simulation clock; (ii) a time-stamp ordered list of pending events and (iii) the state variables.

4.2.3 Parallel and Distributed Discrete Event Simulation (PDES and DDES)

In this summary, we deal with *parallelism at model function level*. In particular, we focus on methods which make intensive use of multiprocessors architectures for DES and which can be classified in between the following two classes: *parallel discrete event simulation* (PDES) and *distributed discrete event simulation* (DDES).

In PDES and DDES, a simulation model is partitioned into regions or domains[4]. Each region is simulated by a so-called *logical process* (LP). Each LP consists of [73]: (i) a spatial region R_i of the simulated system; (ii) a simulation engine SE_i executing the events belonging to the region R_i and (iii) a communication interface, enabling the LPs to send messages to and receive messages from other LPs (see Fig. 4.2b).

LPs are mapped onto distinct processors with (as an assumption) no common memory. Thus, every LP can only access a subset of the state variables $S_i \subset S$, disjoint to state variables assigned to the other LPs. The simulation engine SE_i of each LP processes two kinds of events: *internal* events which have no direct causal impact on the state variables held in the other LPs and *external* events that can change the state variables in one or more other LPs. If an external event is processed, the LP holding the state variables that are to be changed is informed through a message sent by the LP. The message routing between the LPs is done by a communication system, connecting the LPs. Incoming messages are stored in input queues, one for each sending process.

Box 3 The principle of causality

Causality, a notion central to natural science and logic, describes the relationship between cause and effect. In general, the principle of causality says that the cause must precede its effect; in parallel execution, and for PDES in particular, we mean that the succession of time-stamped events that we execute can be always reproduced by a sequential simulation. Therefore, no event with an higher time-stamp (effect) that is influenced by an event with lower time-stamp (cause) can be executed out of order. If this happens, we have a *causality violation*.

[4]for the purposes of this paper, only spatial decomposition is considered; however, the concepts illustrated here are also suitable for decompositions into general domains.

Due to different virtual time progression within the various LPs, it is difficult to guarantee the causality principle (see Box 3). Special considerations have to be made to obtain the same simulation results from DDES as from sequential DES. The two most commonly used synchronization protocols in DDES are: (i) The *conservative* (or Chandy-Misra) synchronization protocol developed by Chandy and Misra and [36], [35] (ii) the *optimistic* (or time warping) simulation protocol based on the virtual time paradigm proposed by Jefferson [117]

4.2.4 Conservative vs Optimistic

The basic idea of the conservative protocol is to absolutely avoid the occurrence of causality violations. This assert is granted by strictly freezing the computation of an event e with virtual time (VT) t_e until when no messages with VT lower than t_e will be received. Under the assumption of FIFO message transport, this is achieved by only simulating an event if its VT is lower than the minimum of the time-stamps of all events in all input queues. An obvious problem arising in conservative simulation is the possibility of deadlocks [106]. Some deadlock resolution schemes have been developed during the last years. Among them, the more interesting are: avoid deadlock by the use of NULL-messages [36] and detect and recover deadlock in advance [35]. Some optimization protocols are discussed in [24, 230] (NULL-messages approach), in [10] (NULL-messages on request), in [45, 91] (lookahead computation), and in [175, 199] (local deadlock detection).

In contrast with the conservative protocol, there is no blocking mechanism in the optimistic one. An event is simulated even if it is not safe to process. Thus, causality errors are allowed to occur, but are later detected and solved. To guarantee causality, a mechanism called *time warp* or *rollback* is implemented. Time warp is optimistic in the sense that each processor P executes events in time-stamp order optimistically assuming that causality is not being violated. At any point, however, P may receive a straggler event E, that should have been executed before the last several events already executed by P. In this case, it rolls back to a check pointed system state that corresponds to a time-stamp which is a global minimum among all VT (global virtual time) and then less than the straggler's time-stamp. Processor P resumes its execution from this point, and P processes E, in the right time-stamp order. A successful optimistic DDES minimizes the runtime costs of (i) state-saving system state, (ii) rollback, (iii) global virtual time (GVT) computation, and (iv) interprocessor communication.

4.2.5 Characterization of the Gillespie SSA as a DES system

From a computational point of view, a biochemical system designed to be simulated with the Gillespie algorithm can be seen as a collection of interacting processes, where each process can be in a different state among a set of discrete states. In this view, *biochemical species* are treated as *processes* that are able to perform a set of actions, changing their state in response to an external or internal action; *reactions* can be codified as *events* that are composed of a number of complementary actions, so that the execution of a reaction results in a simulation event that executes two (in the case of mono-molecular reactions) or more (in the case of bi-molecular reactions) actions in two or more processes (see Fig. 4.4).

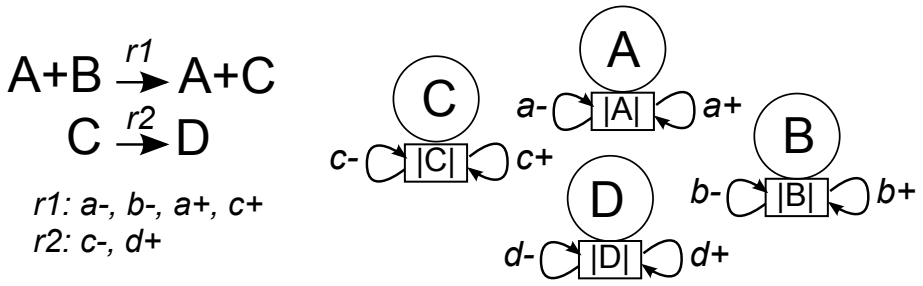


Figure 4.4: A set of species and a set of reaction (left) represented as a set of processes and events (right). Each event is composed by two or more actions that modify the state of each process, typically decreasing or increasing the counter for the cardinality of the corresponding species.

Formally, a *biochemical system* $S = (P, E)$ is a set of processes $P = \{p_1, ..., p_n\}$, each holding a set of states and a set of actions, and a set of events $E = \{e_1, ..., e_m\}$, each composed by a set of actions; typically, for every process p there will be two actions $(a+, a-)$ in charge of decreasing and increasing the counter for the cardinality of the corresponding species. However, it is possible and sometimes useful to add additional state variables and corresponding actions (for example, to encode BlenX global events introduced in Chapter 3).

Following this computational view, the simulation of a biochemical system with the Gillespie algorithm becomes a DES, where event times are generated by sampling an exponential distribution. The fact that times are generated by an exponential distribution leads to some insights in how this particular DES can be parallelised. In particular, we will show that it is almost never convenient to parallelize biochemical systems by using a conservative approach. In support of our analysis, we shall consider a *dependency graph* between events, defined as the graph of reactions introduced by Gibson and Bruck [80].

Definition 4.2.1. *Let Reactants*(e) *and Products*(e) *be the sets of reactants and prod-*

ucts, respectively, involved in the event e.

Here, for *reactants* we indicate the processes whose actions decrease the cardinality of their state variable, identified with the name of the process and a '-' suffix. For example, the event *e1* in Fig. 4.5 is composed by the actions $\{a-, b-, c+\}$; the actions $a-$ and $b-$ modify the state of A and B, so $Reactants(e1) = \{A, B\}$. *Products* are defined in a similar way as the processes whose actions increase the cardinality of their state variable.

Definition 4.2.2. *Let $DependsOn(e)$ be the set of processes whose state change affects the execution time of the event* e*, and $Affects(e)$ the set of processes whose state changes when an event is executed.*

Following the description of the Gillespie SSA given in Sec. 2.2.4, we have that $DependsOn(e) = Reactants(e)$. Typically, $Affects(e) = Reactants(e) \cup Products(e)$, or better, the set of processes on which the actions in e act. Sometimes, when two actions are complementary (i.e. one cancels the effects of the other), the set can be a little smaller. This is the case of the event $e3$ in Fig. 4.5 for example, where $e-$ cancels $e+$ and $Affects(e3)$ can be reduced to $\{D, F\}$.

Definition 4.2.3 (Dependency graph)**.** *The* dependency graph *of a biochemical system S is a directed graph $G(V, E)$ in which the set of nodes V corresponds to the set of events and there is a directed edge between $V(e1), V(e2)$ if and only if $Affects(e1) \cap DependsOn(e2) \neq \emptyset$*

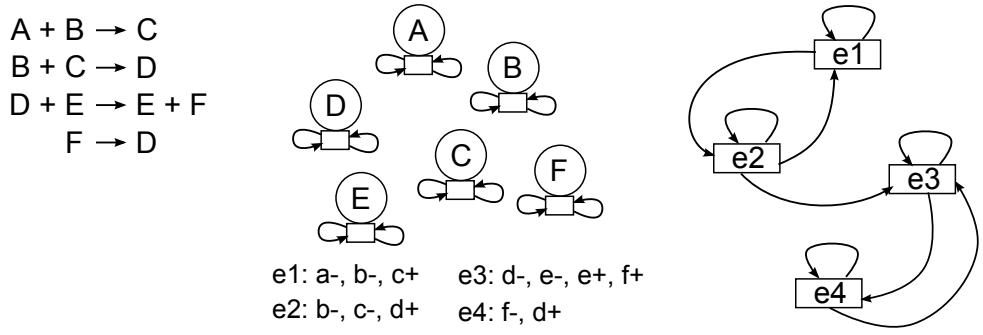


Figure 4.5: The dependency graph (right) for a simple biochemical system (left).

The dependency graph can be used to show that the dependencies among reactions, united with the times sampled from an exponential distribution, in many cases lead to the need for sequential execution.

Definition 4.2.4. *Considering a system S, its dependency graph can be partitioned into a set of* strongly connected components*. We call the set of processes and events belonging to*

a strongly connected component of cardinality greater than one a subsystem[5] *(see Fig. 4.6).*

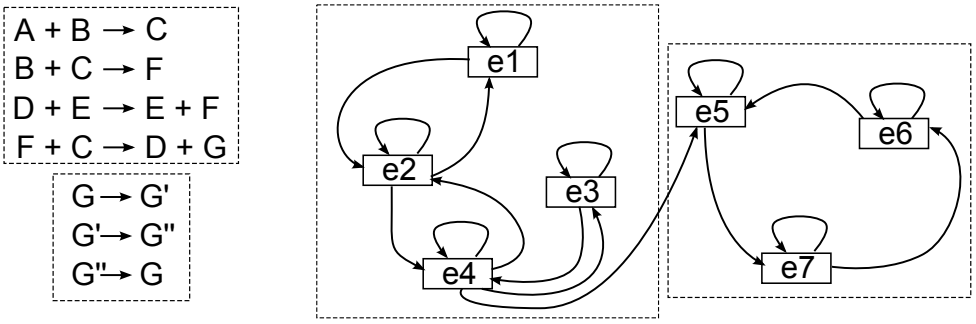


Figure 4.6: A biochemical system partitioned into subsystems.

Theorem 4.2.1. *In a DES system implementing the Gillespie SSA, the execution of an event may lead to the need to recompute the next execution time for all the events in a subsystem.*

Proof: whenever an event is executed, the next execution time of the events depending on it, i.e. its neighbours in the dependency graph, must be updated. Since in the Gillespie SSA times are exponentially distributed, there is no lower bound that guarantees us that the times we are going to recompute will be higher than a certain threshold.

The events with the new, lower, time-stamps will in turn lead to the need for recomputing the time of other events, with the possibility of generating lower time-stamps for the events they affect. By definition, in a strongly connected component there exists a path between any two vertexes, so it is possible that the generation of new times ripples and affects all the events in the subsystem.

Consider Fig. 4.7: execution of event $e1$ leads to the re-computation of $e2$, which leads to the re-computation of $e4$... In the end, all the events in the same subsystem are influenced, leading possibly to a completely different schedule (order of execution).

From this theorem, we can immediately derive the following two corollaries:

Corollary 4.2.2. *Even when times are drawn from an exponential distribution, the absence of causality errors in a subsystem is guaranteed whenever actions are executed in increasing time-stamp order (serialization, or* zero lookahead*).*

Sketch of proof: if events are executed in increasing time-stamp order, without speculatively executing events in the 'future', they will be not influenced by ripple effects on a schedule.

[5]Since each reaction event has a dependency on itself, strongly connected components always exists; the case of trivial SCC with cardinality of one is explicitly non considered in the definition of subsystems

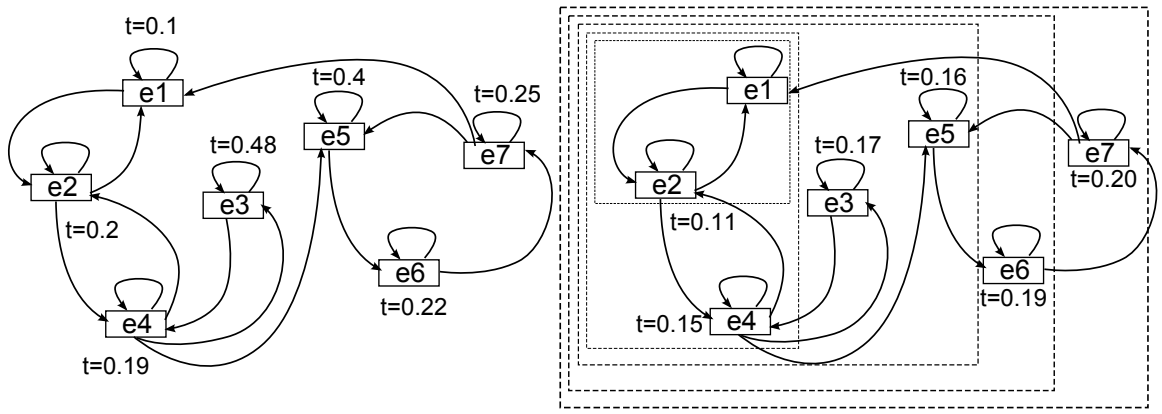


Figure 4.7: Due to the exponential distribution used to generate execution times, the execution of an event can lead to the re-computation of the times of all the events in the same subsystem during the successive simulation steps.

Corollary 4.2.3. *When times are drawn from an exponential distribution, the absence of causality errors is guaranteed only if actions are executed one after the other; therefore a pure conservative approach to PDES -which allows actions to be executed only when they cannot incur in causality errors- has a lookahead of zero, leading to a* serialized *execution where no speedup is possible.*

We considered the usage of some methods to obtain the *lookahead* necessary for concurrent execution. However, using these techniques would lead to unacceptable compromises concerning the accuracy of the simulation. In fact Jeschke et al. [118] showed that most of the techniques used to obtain lookahead for the conservative approach, such as artificially inserting lookahead into the computation and relaxing ordering constraint, have drawbacks that makes them not suitable for our goals.

An alternative approach for having some lookahead even in presence of exponential distributed random numbers is pre-sampling. Pre-sampling is a technique proposed by Nicol [160] for computing lookaheads in queueing network simulations with exponential distributed service time, and then used also for federated military simulations by Loper and Fujimoto [149]. However, it presents a number of problems that makes it an infeasible approach in our domain. As noted by Nicol and Fujimoto, the service time variation has a strong effect on speedup. Under high variation very small lookahead values are possible, meaning that lookahead is computed more often, thereby incurring in increased overhead. Furthermore, they also noticed that rich interconnections between simulated entities, such as those used for simulating a spatial environment, cause increased uncertainty in future behaviour, resulting again in small look-ahead, with poor performances especially when using exponential distributed times.

Fujimoto concludes that, to perform well, this technique requires: (i) fixed sized time

intervals, with the same time distribution for all messages; (ii) precise time-stamps with few random number samples and (iii) some knowledge concerning the number of messages produced in the near future [149]. Since reaction-diffusion biochemical simulations do not meet any of these requirements, we decided to discard this approach.

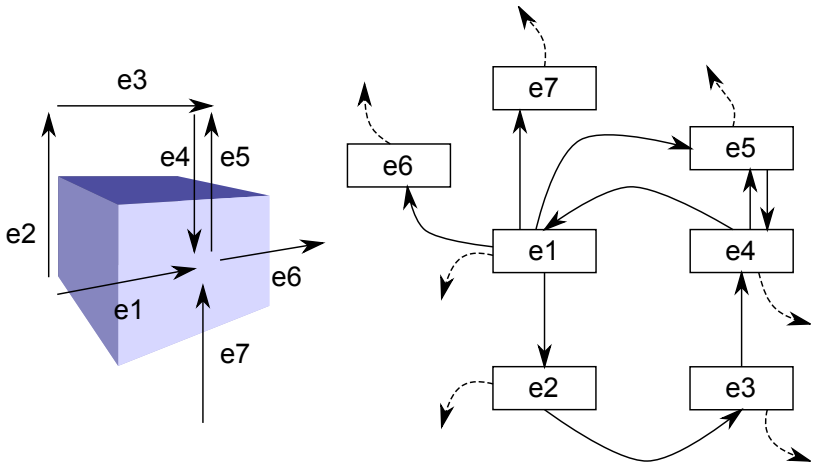


Figure 4.8: Due to the diffusion events, a reaction-diffusion system has only one single subsystem.

Indeed, it is possible to make two crucial observations about reactive-diffusive simulations: (a) in a reaction-diffusion systems where species are free to diffuse in every direction, the dependency graph for diffusive events is fully connected; thus, the whole system is a single big subsystem (see Fig. 4.8) and (b) many biological systems show a little number of big subsystems; compounds, molecules and enzymes in a cell are reused over and over, forming big interconnected networks with loops. Indeed, regulation and transcription processes are often based on feedback loops, whose introduce loops and connected components (subsystems) in the dependency graph.

In conclusion, the Gillespie SSA can be characterized as a DES. Between two main approaches to parallelize DES, the optimistic approach is the more promising one: as the Cor. 4.2.2 and Cor. 4.2.3 show, a pure conservative approach, united with exponentially distributed times and the particular dependency structure we have in biochemical systems, is very likely to perform poorly.

4.3 Parallel Simulation of Species-based Systems: Redi

As a proof of concept, we designed and developed *Redi*, a parallel stochastic reaction-diffusion simulator that uses a new model with state-dependent diffusion coefficients for a reaction-diffusion Gillespie method.

4.3.1 Redi method: state-dependent diffusion rates

Similarly to other methods, Redi is based on the Fick's law we introduced earlier in this chapter. The Fick's law depends on a parameter D, the *diffusion coefficient*, which represents the diffusion dynamics. When the medium is isotropic, the *diffusion coefficient* is a constant scalar independent of the concentration of the solute. However in biological system even for purely diffusive transport phenomena the classical Fickian diffusion is at best a first approximation [2, 1]. A better approximation is to make the diffusion coefficient of a species dependent on the concentration of that species and on the other species of solutes eventually present in the medium.

With Paola Lecca [140], we developed a new model for concentration dependent diffusion coefficients for a reaction-diffusion system. Then, we used the computed concentration-dependent diffusion coefficients to calculate the rates of diffusion of the biochemical species. For simplicity we treated purely diffusive transport phenomena of non-charged particles, focusing on diffusion driven by a chemical potential gradient.

Our method consists of the following five main steps:

- calculation of the *local* virtual force F per molecules as the spatial derivative of the chemical potential
- calculation of the particles mean drift velocity in terms of F and local frictional f;
- estimation of the flux J as the product of the mean drift velocity and the local concentration;
- definition of diffusion coefficients as function of local activity, concentration and *frictional coefficients*, modelled as linearly dependent on the local concentration too.
- calculation of diffusion rates as the negative first spatial derivative of the flux J.

The determination of the activity coefficients requires the estimation of the *second virial coefficient*: in our model this estimation is performed using a Lennard-Jones potential to describe the molecular interactions. Details about the method derivation can be found in Appendix.

We developed an algorithm that computes the diffusion rates as described. The algorithm first subdivides the volume into cells of fixed dimension. The dimension of each cell in the mesh is chosen to be not too fine-grained, in order to reduce simulation time, but within the constraints described in [14] to preserve accuracy.

The algorithm is designed starting from NSM. NSM is efficient but centralised and sequential in nature, and can have problems in scaling to very large systems. Moreover,

it cannot be easily adapted to take advantage of parallel or distributed systems, which is one of our goals for *Redi*. Our algorithm overcomes these limitations by eliminating the use of a global priority queue.

Let assume that for every cell of the mesh the concentration, the diffusion and reaction rates of the chemicals and their interactions are known. Let assume also that for every cell of the mesh the type next event (reactive or diffusive) and the time at which it will occur are known. In order to apply the original Gillespie algorithm to the chemical reactions occurring in each reaction chamber, we require the concentrations of chemicals located in that cell to be homogeneous. For each cell we draw a set of *dependency relations* with neighbour cells; the cell can perform its next event only if it is quicker than the diffusion events of the neighbour cells, because diffusion events can change reactant concentrations, and therefore the time and order of the events. The algorithm makes use of this property: as each cell can *evolve* independently from other cells if it does not violate the restrictions imposed by its dependencies, at every step all the cells that can evolve are allowed to consume one event and advance one simulation step. Note that, as reactions executed in the current cell are *older* than the ones in its neighbours, we do not have to worry about them altering our concentrations meanwhile.

The algorithm has still the same average computational complexity of Elf and Bernstein methods. Nevertheless, removing the global priority queue should allow us to design an implementation that scales better with the number of reactions and processors.

4.3.2 Redi implementation: Optimistic Spatial Gillespie

The implementation of the *Redi* simulator is driven by two goals: *correctness* (the simulator must respect the assumptions underlying the method of choice) and *scalability* (the addition of further processing power must result in an increased execution speed).

Both goals are achieved by using an approach based on PDES with an optimistic scheduling policy, as discussed in Sec. 4.2.2.

Notice that the two objectives must be considered together, as the one heavily affects the other. Some methods, like the one presented in [193], violate both the assumptions made by the Gillespie SSA (homogeneous and well-stirred environment) and the properties stated by Bernstein (that diffusion events must be at least as frequent as reaction events) to obtain fast parallel execution through volume subdivision. The algorithm, as the authors admit, can be useful in some cases, but it is not correct in a general sense. Indeed, when the spatial localization of molecules becomes important for the purposes of the experiment, the algorithm produces incorrect results.

For an effective implementation of the simulation algorithm as a PDES, the global state information should not be maintained as much as possible, so that different pro-

cessors can process and update their partial local state concurrently. Algorithms like the Next Subvolume Method (NSM) maintain information of execution times in global data structures and therefore they are not immediately adaptable to a parallel environment, even if a distributed version of the algorithm was recently proposed by Jeschke et. al. [118].

We take a slightly different approach with respect to the NSM: we analyse the problem from the point of view of a single cell on the two or three-dimensional grid. We assume that every cell knows and stores its local information: concentrations of species, diffusion and reaction rates, and next reaction time, as well as references to its neighbours. In each cell there are some dependency relations, both between species inside the same cell and between those in neighbour cells that can diffuse inside (see Fig. 4.9(a)). We have noticed that each cell on the grid can *evolve* (execute simulation events) independently of the other cells if the executed events do not violate the restrictions imposed by the dependencies. Following the optimistic approach, we let each cell evolve independently, up to when a diffusion event occurs. When a neighbour notifies to the current cell a diffusion event with a clock T_{diff} smaller than the current clock T_{act}, reactions in the current cell with times between T_{diff} and T_{act} are marked as *straggler*. So, we rollback every action executed within T_{diff} and T_{act}, recompute propensities and reaction times and restart the simulation of the events in that cell from time T_{diff}.

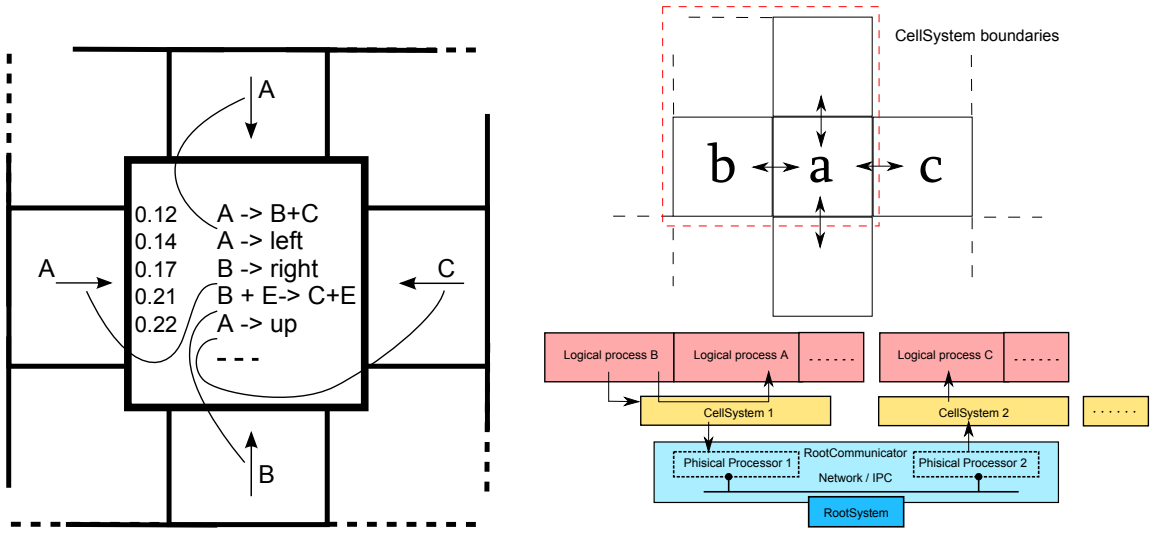


Figure 4.9: Each cell is modelled as a process in a PDES (a). Cells are grouped into systems in order to reduce communication overhead (b).

Each cell is mapped to a logical process (LP) and multiple logical processes can be mapped onto a physical processing unit. The assignment of logical processes to physical

processes may be done either dynamically, possibly by using a load balancing algorithm, or statically, by exploiting spatial locality to reduce communication overhead.

4.3.3 Performance Considerations

The *scalability* goal is not easy to achieve because a lot of practical, real world considerations have to be taken into account. The SSA was designed to run efficiently on hardware of the late '70s, and it is indeed quite efficient. An implementation of the Gillespie algorithm can process and simulate roughly 10^5 reaction events per second; that is, a simulation loop takes approximately 10000-30000 CPU cycles to execute. Since a simulation loop is so fast, it is really difficult to speed it up by means of a parallel architecture. Execution of diffusion or reaction events on different processors requires synchronization in order to exchange messages. In the best scenario, processes can run on a single multi-core machine, where communication is done using shared memory and mutexes. According to the literature and to our tests, even in this case the cost of switching context and proceeding the execution on a different thread (roughly 5000 CPU cycles) can easily result comparable to the loop time (see Fig. 4.10).

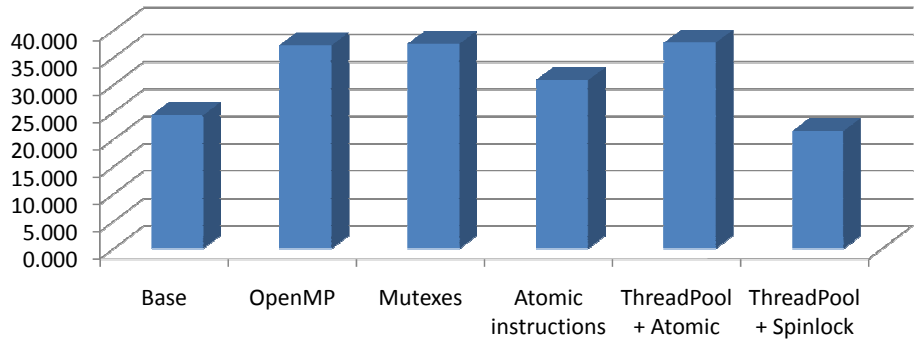


Figure 4.10: The execution time of a parallel simulation (running on 2 processors) using various techniques of synchronization and inter-thread communication, compared to a serial simulation (Base). Notice that the overhead for running on multiple threads actually *increases* the execution times in all but the last case, where we used a pool of threads and hand-written assembly code for synchronization.

On a shared memory architecture, the problem is even worse. Even if current HPC architectures can rely on very low-latency connections and very efficient message passing implementations (like the MPI interface we used), communication overheads can vanish any performance gain. For this reason, we chose a coarse granularity when we assigned cells (i.e., logical processors) to physical processors, in order to reduce the overhead to the minimum. For this reason, we designed our parallel simulator in a hierarchical way (Fig. 4.2b): logical processors representing cells are grouped into *Cell Systems*, that hold

the partial state for a set of spatially contiguous cells. *Cell Systems* are then grouped and driven by a *Root System*, that holds topological information on the Cell Systems and that caches some essential information on the global state of the system. Every *System* has a specialized communicator. The *Root System* has a communicator based on MPI to let *Cell Systems* communicate with each other across processor and machine boundaries. The *Cell Systems* have a single threaded, shared memory communicator in charge of maximizing the performance and reduce the overhead on a single processor or core. A further layer can be added with the aim to manage groups of *Cell Systems* which execute on different CPUs or cores on the same computation node, i.e. on a machine that shares the same memory and that does not need for network communication or message passing in case of inter-groups interactions.

Cells and *Cell Systems* communicate through a consistent interface, that is *transparent*, and that allows cells to communicate any diffusion information without taking care of the hierarchy. To communicate a diffusion from the cell C_1 to the cell C_2, C_1 sends a message to its *Cell System*; if C_2 is on the same physical processor (i.e. it belongs to the same Cell System), the information is directly propagated. If instead the *Cell System* realizes that C_2 does not belong to the set of cells it manages, it forwards the information up to the next System, until it reaches a System that knows C_2 or until it reaches the *Root System*. In the second case, the information is propagated using inter-thread communication or MPI messages (see the pseudo-code in Fig. 4.11).

Example

Redi accepts an input file that specifies reactions, reaction rates and diffusion coefficients, as well as the initial location of the chemicals.

We tested our simulator with some models (enzymatic reactions, oscillatory networks, chemotaxis pathway) under realistic conditions: most or all the molecules not attached to membranes have been let to move and, mostly important, the diffusion coefficients have been set always higher or at least comparable to the reaction rates. Such conditions obviously increase the number of messages sent, making harder for our simulator to appropriately scale. However, it is fundamental to provide a realistic model that respects the assumptions we made [14].

Here we briefly show a spatial version of the Lotka-Volterra predator-prey model. This model is simple yet effective; in particular, the model exhibits a different and interesting behaviour when ran in an environment that includes spatial information [204]. The results we obtained (see Fig. 4.13) are consistent with what we expected and with what is found in the literature [204].

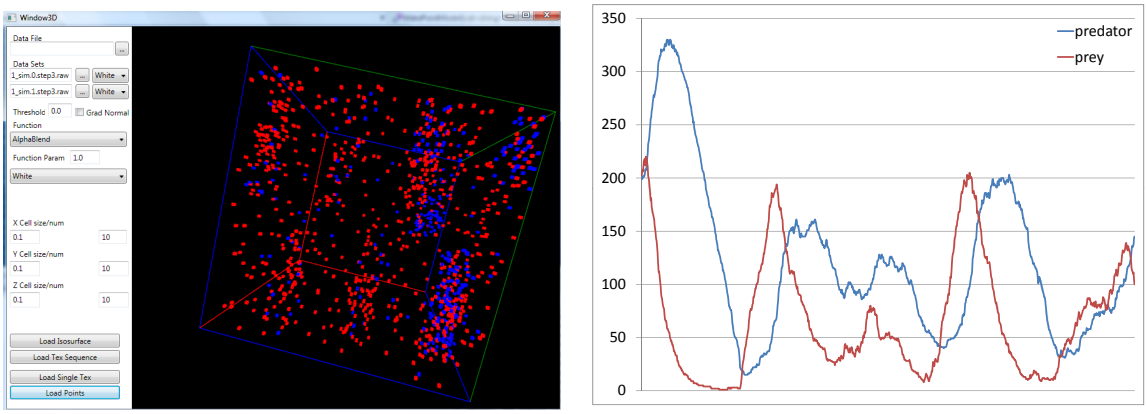
This model allowed us to perform some initial performances estimations, listed in Ta-

```
CellSystem ():
  while true do
    NextAction := FastestCell().FastestAction;
    StateChange := Action.Execute();
    History.Add(StateChange);
    UpdateClock(StateChange);
    if Action.IsDiffusion()
      if Action.TargetCell ∉ CellSystem.Cells
        RootComm.Notify(StateChange);
      else
        Action.TargetCell.Notify(StateChange);
    if RootComm.HasNotification
      Event := RootComm.HasNotification;
      switch Event.Type
        case Rollback :
          DoRollback(Event.Time);
        case Diffusion :
          Event.TargetCell.Notify(Event.DiffusionAction);

RootSystem ():
  while true do
    Timer := StartTimer();
    Event := WaitForEvents(Timer, RootComm);
    switch Event.WakeReason
      case TimerTick :
        SendCheckpointCommand(GlobalTime);
        SystemState := RecvCheckpointData();
        DoCheckpoint(SystemState);
      case Communication :
        switch Comm.Type
          case Error :
            BroadcastRollback(Comm.Time);
          case Diffusion :
            TrgtSystem := LookupSystem(Comm.SourceCell);
            TrgtSystem.ForwardDiffusion(Comm);
    CurrentGlobalTime := Event.UpdateTime();
```

Figure 4.11: Pseudo-code for *CellSystem* and *RootSystem*

```
var predator : rate 100;
var prey : rate 100;

predator + prey -> predator + predator [55];
prey -> prey + prey [15];
predator -> nil [10];

run  prey [1, 1, 0, 100]; prey [14, 14, 0, 100];
     predator [2, 2, 0, 100]; predator [12, 12, 0, 100]
```

Figure 4.12: The input file for the 3D Lotka-Volterra model.



(a) (b)

Figure 4.13: A time-step of the Lotka-Volterra simulations (a) and the variation in cardinality of each species over time (b).

ble 4.1. We measured the execution time of the serial version of the algorithm, where all the inter-process communications were removed and substituted with direct manipulation of data structures in shared memory, and of the optimistic parallel algorithm. The hardware used for the simulation consists of PCs with AMD Opteron 64-bit CPUs at 2.4GHz, 4GB of Ram, interconnected with a 10Gbps Infiniband connection. We can observe that the overhead is significant when dealing with a small 16x16 2D grid, for a total of 256 node; the overhead starts to be less heavy starting with a 100x100 2D grid. As the grid becomes larger and larger, given a fixed number of subsystems, the diffusion events between different subsystems becomes less frequent. Note that a number of cells in the tens or hundreds of thousands is not unrealistic; for example, data for the last row of Table 4.1 were obtained for a 32x32x32 3D grid.

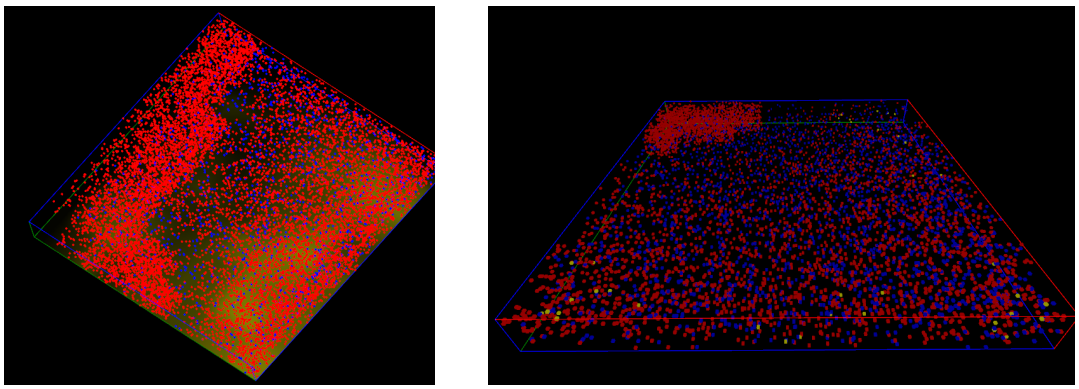
4.3.4 Case studies

The model of diffusion we have proposed has been successfully applied to simulate the spatial dynamics of molecules in non-homogeneous media. We examined several case

N cells	Serial	2 Cores		5 Cores		12 Cores	
256	1.5	14.8	0.1x	-	-	-	-
10000	13.1	10.7	1.22x	-	-	-	-
16384	17.4	(12.3/15.4/13.5)	1.29x	(9.1/10.1/9.4)	1.86x	-	-
26896	64.1	(34.7/42.8/38.7)	1.66x	(14.2/17.2/15.1)	4.25x	(7.0/9.4/8.0)	8.06x
32768*	75.8	(42.7/47.3/45.3)	1.67x	(18.4/20.7/19.2)	3.95x	(16.7/17.1/16.9)	4.49x

Table 4.1: Times in second for the execution of $5 \cdot 10^4$ simulation steps, 400 entities, (min/max/avg of five runs), and speedup for the parallel algorithm (*: on a 3D grid)

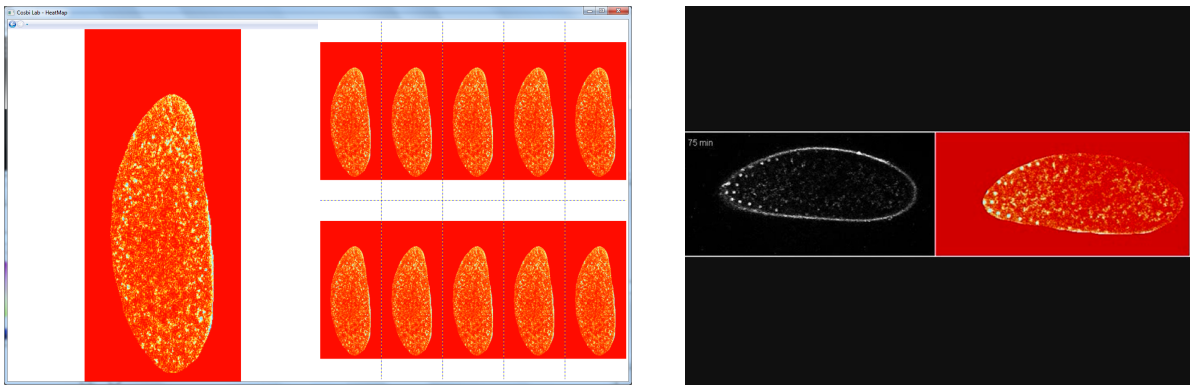
studies where spatial effects are relevant: spatial effects due to the irregular distribution of chaperones on the kinetics of the chaperone-assisted protein folding [140, 142] (see Figure 4.14), tubulin diffusion in cytoplasm [141] and Bicoid diffusion in Drosophila embryo [143].



(a) (b)

Figure 4.14: A sample view of the distribution of chaperones (bluepoints)and nascent proteins (red points), right-folded proteins (yellow points), misfolded proteins of type 1 (green points) and misfolde proiteins of type 2 (magenta points).

In the latter case study, Lecca et al. [143] modelled a morphogen gradient with Redi. During embryonic development, cell differentiation is position-dependent and is regulated by signalling molecules, called morphogens. Morphogens are produced in a specific region of a tissue and move away from their source to form long-range concentration gradients: cells subsequently differentiate in response to the morphogen concentration. The study concerned with the simulation of the dynamics of the Bicoid protein spatial distribution in Drosophila embryo, using Redi to point out a plausible range of the diffusion coefficient for this protein (see Figure 4.15).



(a) (b)

Figure 4.15: Stochastic simulation of the Bicoid diffusion in Drosophila embryo with Redi: (a) Simulation running (b) comparison with in-vitro fluorescent microscopy.

4.4 Parallel Simulation of Individual-based Systems: GPUSmol

As part of the effort to apply parallel computation to simulation methods for systems biology we developed another, quite different algorithm. *GPUSmol* is based on Smoldyn [8], an individual-based, BD (Brownian Dynamics) level stochastic simulator. Therefore, GPUSmol performs individual based simulation with single molecule detail, in contrast with Redi, which is a species-based simulator. The most important aspect is that the algorithm is targeted to Graphics Processing Units (GPUs), whose structure and programming model allows for great speed-ups. A brief itroduction to GPUs and to GPU computing is given in Appendix C.

4.4.1 A GPU based implementation of Smoldyn

Smoldyn

Smoldyn adopts an extension of the Smoluchowski model for diffusion-influenced systems. In the Smoluchowski model time increases continuously, as it does in nature; Smoldyn instead adopts finite time steps of fixed length for the simulation algorithms.

Each molecule is treated as a point-like particle with continuous x, y and z coordinates, that diffuses freely in space following Fick's first law. The Smoluchowski description also accounts for external and long-range forces, but they are ignored in Smoldyn as they typically have minimal influence in biochemical systems.

Smoldyn ignores the dynamics of the solvents and of other unreactive species, leading to a detailed Brownian motion of the reactive molecules that allows for accurate results at larger scale. It also ignores steric interactions, spatial orientations and internal energy levels (which are events that occur faster than the diffusive and reactive processes of

interest). Therefore, the complete state of the model at each time step is fully specified by a list of the molecular positions.

In Smoldyn, a bimolecular reaction occurs when two reactive molecules collide with each other. However, as we have seen in the description of the Gillespie algorithm, most reactions occur at a slower rate because only collisions with the right orientation and with a minimal energy lead to a reaction. This is addressed in Smoldyn by replacing the sum of the molecular radii with a smaller effective *binding radius* to reproduce the correct steady-state reaction rate for bimolecular reactions (see Fig. 4.16).

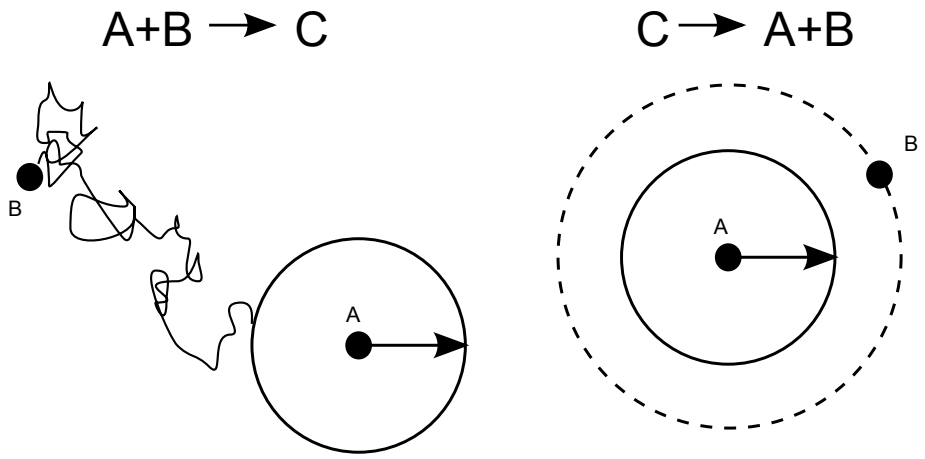


Figure 4.16: A forward reaction occurs when one A and one B molecule diffuse within a distance that is less or equal to the binding radius. If a reverse reaction is present, the A and B products are initially separated by the unbinding radius which is made larger than the binding radius to prevent the instant recombination of the products.

These binding radii are computed at the beginning of the simulation and stored in memory to improve performances. Besides the binding radius, an additional parameter, called *unbinding radius*, is computed for reversible reactions. In reversible reactions, like $A + B \leftrightarrow C + D$, the products C and D need to be positioned at a distance that is greater than the binding radius for the reverse reaction ($C + D \rightarrow A + B$), otherwise it will occur immediately. This is neither desirable nor correct. Therefore, products are initially separated by the *unbinding radius*, which is automatically computed from diffusion coefficients and reaction rates [8].

After unbinding, the products can diffuse away from each other or they may diffuse closer to each other and rebind again.

Smoldyn implements this model in a five steps iterative algorithm, with an additional step for pre-processing:

- Initialize

  - compute coefficients for unimolecular reactions;
  - compute binding and unbinding radii.

- At each time step:
  - Perform diffusion
  - Compute Surface interactions
  - Perform Zeroth-order reactions
  - Perform Unimolecular reactions
  - Perform Bimolecular reactions

During initialization step Smoldyn computes the value for several parameters (e.g. binding and unbinding radii). Then, the algorithm proceeds iteratively, computing events occurring at discrete time steps (remember that Smoldyn uses discrete time steps in place of a continuous time model).

In the *diffuse* step, a normally distributed number is generated for each spatial dimension, for each molecule. The position of each molecule is updated, and then surface interactions are computed.

Zeroth-order reactions are computed for every kind of molecule. The probability that exactly j molecules of type A are produced in a single time step is given by a Poisson distribution:

$$Prob(j) = \frac{(k_0 \Delta t)^j exp(-k_0 \Delta t)}{j!}$$

Some computational efficiency can be gained by calculating the required probabilities during the program initialization and storing them in look-up tables (one for each zeroth-order reaction). However, the overall improvement in speed is typically negligible because only one Poisson deviate is required for each zeroth-order reaction at each time step, and it is therefore not employed by Smoldyn.

Unimolecular reactions are computed for each molecule in the system. The probability that a specific A molecule reacts during a Δt long time step is:

$$Prob(reaction) = 1 - exp(-k1\Delta t)$$

However, if a molecule can react via multiple first-order reaction, a sequential application of equation leads to a bias towards the first one. Instead, the following formula is used:

$$Prob(reaction_i) = \frac{k_i}{\sum_j k_j} \Big[1 - exp\Big(- \Delta t \sum_j k_j \Big) \Big]$$

where k_i is the rate constant for the i-th reaction. Rather than re-calculating the reaction probabilities at each time step, Smoldyn employs a faster method: during the program initialization, it calculates the coefficients for each possible unimolecular reaction for a given type, and then sums these probabilities to form a list of cumulative reaction probabilities, stored in a look-up table.

At each time step during the simulation, a specific molecule reacts with reaction i if a uniformly distributed random number is between the i-th and $(i+1)$-th stored cumulative probabilities.

The computation of *bimolecular reactions* employs a spatial partitioning scheme of the simulation volume. When checking for bimolecular reactions, the program only needs to investigate pairs of molecules that are in the same or neighbouring regions, checks their distance and, if the distance is less than the binding radius, performs the reaction by substituting the reactants with the products, placing them in appropriate positions (taking into account the unbinding radius when appropriate).

GPU implementation

In order to obtain great performances from GPUs a *data parallel* approach must be taken; GPU kernels[6] must be programmed so that they access a little subset of data points (one or more) in the domain, in a way that it is as much as possible independent from the others. The interested reader may refer to Appendix C, which introduces GPUs architecture and explains the GPU programming model.

In the case of Smoldyn, the natural data set on which to operate a subdivision is the molecules set. In particular, we choose to run on the GPU the three central and more time consuming steps: first order (unimolecular) reactions, diffusion and bimolecular reactions. Code for initialization (computation of rate, binding and unbinding radii, and so on) remains on the CPU, as well as code for zeroth-order reactions.

The diffusion step needs to process each molecule in the system using a for loop, computing new positions for each one:

```
for(int index = 0; index < numMolecules; ++index) {
    float4 pos = posArray[index];
    float rate = diffusionRates[typeArray[index]];
    int randX = gaussianRand();
    int randY = gaussianRand();
    int randZ = gaussianRand();
    pos.x += rate * randX;
    pos.y += rate * randY;
    pos.z += rate * randZ;
    posArray[index] = pos;
```

[6] See Appendix C

```
}
```

It is relatively easy to adapt this code to run efficiently on the GPU:

```
  uint index = __umul24(blockIdx.x,blockDim.x) + threadIdx.x;
  if (index >= numMolecules)
    return;

volatile float4 pos = posArray[index];    // ensure coalesced read
  volatile int typeId = typeArray[index];

  int rngIndex = index % MT_RNG_COUNT;
  MersenneTwisterState* rngState = &(rngStateArray[rngIndex]);

  float rate = tex1Dfetch(diffusionRatesTex, typeId);
  int randX = MersenneTwisterGenerate(rngState, rngIndex) & gaussianTableDimMinusOne;
  int randY = MersenneTwisterGenerate(rngState, rngIndex) & gaussianTableDimMinusOne;
  int randZ = MersenneTwisterGenerate(rngState, rngIndex) & gaussianTableDimMinusOne;
  pos.x += rate * gaussianLookupTable[randX];
  pos.y += rate * gaussianLookupTable[randY];
  pos.z += rate * gaussianLookupTable[randZ];
```

Besides unrolling the for loop, a couple of adjustments have to be made in order to guarantee very good performances. The most notable problem is to get the random number generator run efficiently on the GPU. Fortunately, CUDA already provides a parallel implementation of the Mersenne Twister pseudo-random number generator [150, 174]. The algorithm for generating random numbers maps well onto the CUDA programming model, as CUDA provides bitwise arithmetic and an arbitrary amount of memory writes. The Mersenne twister is iterative, and the generation of each number requires a limited number of instructions. Therefore it is not possible to parallelize a single twister step using several execution threads. On the other hand the GPU uses thousands of threads for the computation of the new position of molecules, one for each molecule in the system.

The short and simple solution is to have many simultaneous Mersenne twisters processed in parallel. To prevent the emission of correlated sequences by each generator, each twister is provided with a different set of Mersenne Twisters parameters. The computation of twister parameters is a lengthy process, which is done off-line using *dcmt*, a special library supplied by the Mersenne Twister authors for the dynamic creation of parameters [150]. Once they are computed, however, they can be used over and over again to generate un-correlated sequences of random numbers.

The set of parameters and the actual state are stored in the `rngStateArray`, resident in GPU memory. The number of twisters used in this case is 32768, enough for common simulations. However, if more are needed, it is easy to enlarge this number by using *dcmt* to compute a larger set of initial parameters.

Another problem linked to the generation of random numbers is their distribution: like most pseudo-random number generators, Mersenne Twister produces uniformly distributed numbers. On the other hand, our algorithm requires Gaussian distributed numbers in order to simulate Brownian motion. Therefore, the generated random numbers must be transformed accordingly. The Box Muller [18] transformation is commonly used in this case; it is easy to implement and it runs pretty fast on the GPU. Moreover, the CUDA Mersenne Twister implementation includes a Box Muller transformation out of the box. However, a quicker alternative exists: the trigonometric calculations required by the Box Muller transformations make this heavily used algorithm a possible performance bottleneck. Instead, we use of a look-up table. A look-up table has several advantages: it can be computed during the initialization phase, and then stored in the very fast, cached, Multiprocessor constant memory; it is nearly as accurate but it is much faster.

The parallelisation of first-order reactions proceeds in a similar way: the computation of whether each molecule may undergo a unimolecular reaction or not, and in the former case which of the possibly multiple reactions will take place, is done by "tossing a coin" for each molecule, e.g. using again the Mersenne Twister Random Number Generator. A particular tricky point, common to other GPU implementation of similar algorithms, is introduced by the possibility of adding or removing molecules from the system. Reactions of degradation, decomplexation and so on (like $A \rightarrow 0$ or $C \rightarrow A + B$) change the total number of molecules available in the system; the natural data structure for a serial implementation, in this case, is to use a linked list to store molecular data. Smoldyn, in fact, uses this data structure internally.

However, the nature of a GPU and its programming model makes it necessary to memorize data as contiguous, and therefore fixed size, arrays (see Figure 4.17a and Box 4). An initial, simple solution is to mark degraded molecules with a special "not valid" flag and add new molecules to the end of the array, which was purposefully allocated larger than necessary. This solution, however, may lead to serious performance degradation, as writing to a shared location that could potentially be accessed by multiple threads requires atomic functions, serialization, and leads to sub-optimal memory access patterns.

The solution in this case is to record the introduction of new molecules in the system using an auxiliary array. This array will be filled with information of what happened to each molecule in the system; when the i-th molecule undergoes a reaction that leads to the addition of one molecule, the type of the new molecule is written in the i-th position of the auxiliary array, otherwise that position is filled with 0. At the same time, when a molecule is removed, we adopt the previous mentioned strategy, filling the i-th position in the *original* array with a 0 (non-valid id; see Figure 4.17b). For example, suppose that the i-th molecule, of kind C, underwent a reaction $C \rightarrow A + B$: in this case, C is changed

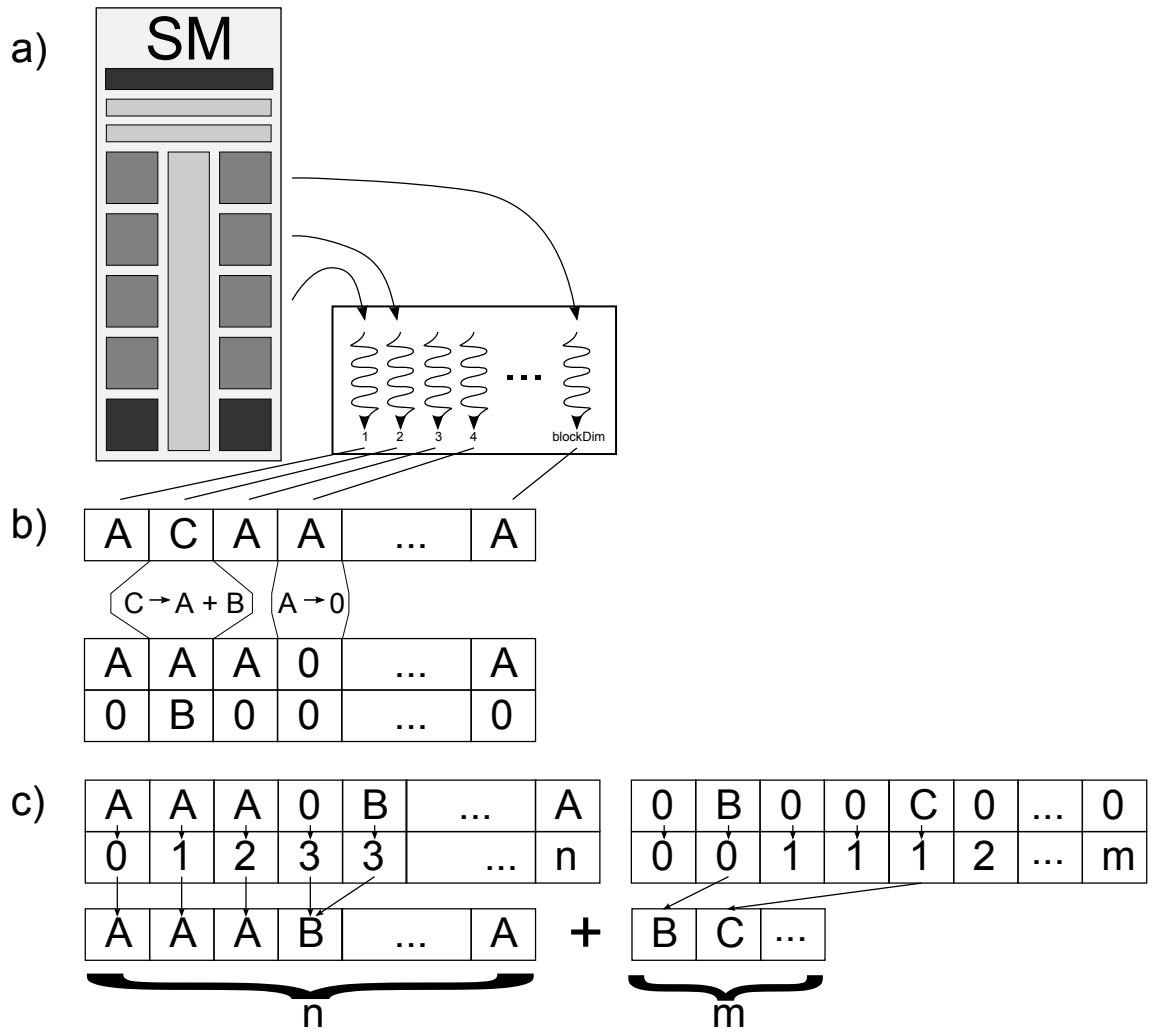


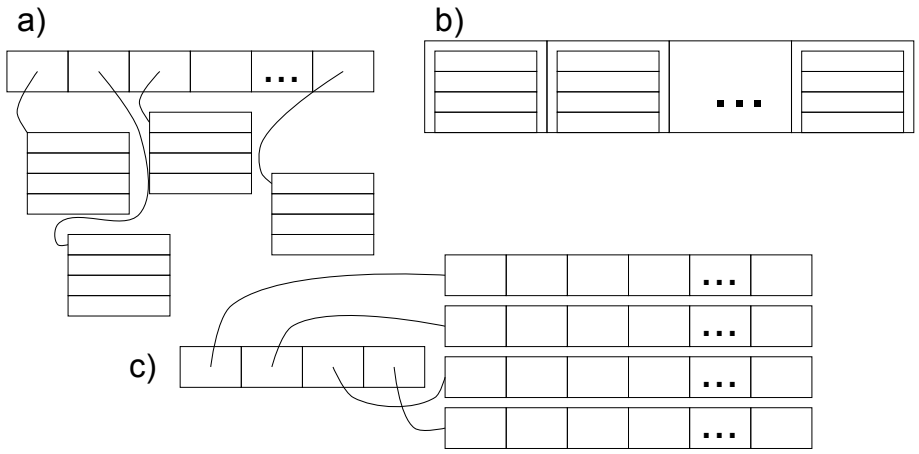
Figure 4.17: First-order reactions on a GPU. The array in the figure is the array holding molecules type; the same procedure is carried out for every other molecule specific information.

into A, but a B molecule needs to be added to the system as well. The id for C is then added to the auxiliary array at the i-th position.

At the end of this first step, a *segmented scan* is performed. Scan primitives are powerful, general-purpose data-parallel primitives that are building blocks for a broad range of applications. Harris et al [207] developed a GPU implementations of these primitives, which includes an efficient formulation and implementation of segmented scan, using CUDA. The result is *cudpp*, a library that contains several parallel processing algorithms based on segmented scans. A common algorithmic pattern that arises in many parallel applications with complex access requirements is the prefix-sum algorithm. The input to prefix-sum is an array of values. The output is an equally sized array in which each

Box 4 Data storage on the GPU

In order to store a collection of objects in memory, like molecules in our case, a plethora of choices are available. Even if we restrict ourselves to arrays, it is possible to store data as an array of pointers(a), array of structures (AoS)(b) or structure of arrays (SoA)(c):



Kernels are launched on multiple threads, and data will be split among threads for parallel processing. The best access pattern is to have each thread reading a separate, unique portion of memory; therefore, predictable, computable offset inside the data array -like in the b) and c) cases- is preferable; moreover, even if it looks like a weird alternative from a programming point of view, structure of arrays (SoA) are preferable to array of structures (AoS) for achieving global memory coalescing on the GPU. Therefore, in our implementation, we memorized molecular data in a set of arrays of basic, primitive data types.

element is the sum of all values that preceded it in the input array. Using this algorithm in conjunction with a simple pre-processing kernel, it is possible to count the number of molecules deleted and introduced in the system.

These operations are performed in multiple steps, using auxiliary arrays and several calls to the cudpp library, or combining all of them in a single, custom kernel implemented using the same ideas and algorithms. In both cases, the output of the procedure is the total number of molecules in the system, and two sets of concurrently compacted arrays, which represent the data associated to the original, still active molecules and the data associated to the newly introduced molecules respectively (Figure 4.17c).

The treatment of bimolecular reactions is more complex and interesting. A trivial solution may be that of testing the distance between any possible pair of molecules; GPUs are particularly efficient in this scenario. Consider for example n-body simulations, like in astrophysics and Molecular Dynamics. However, we have seen that in the model under examination long-distance interactions are either already taken into account or ignored because too marginal to influence the overall behaviour; we only need to examine pairs of particles within a distance equals or less than the biggest *binding radius*. As we

mentioned, Smoldyn already employs a *spatial subdivision* technique. The algorithm for spatial subdivision used by Smoldyn cannot be used directly on the GPU, but we can substitute it with a different, GPU friendly algorithm.

The usage of *spatial subdivisions* for local interactions was one of the first problems tackled by GPU programmers: indeed, like most of the first GPU programs, this problem has been tackled not for scientific purposes, but for a recreational one. Videogames gave by far the most important push in graphics technology: one need only consider that GPUs had been invented with the only purpose to accelerate video-game graphics. In many videogames, realistic rendering of natural phenomena is done using *particle systems*: movement of objects, like stones, but also of fluids (smoke, water, lava and so on) is obtained by using hundreds of thousands small, hard balls that interact locally with each other, and optionally globally with a directional force (like gravity). Techniques used to speed up simulation and rendering of particle systems are therefore an ideal candidate for accelerating simulation of bimolecular reactions in a BD framework.

In particular, we employed a technique developed especially for CUDA [90]. Spatial subdivision is obtained using a three dimensional grid; the grid cell size is double the radius of the biggest binding radius. This means that each molecule can interact only over a limited number of grid cells (27 in 3 dimensions, see Figure 4.18). The grid data structure is generated from scratch each time step: it may be possible to perform incremental updates to the grid structure on the GPU, but this method is simple and the performance is constant regardless of the movement of the particles.

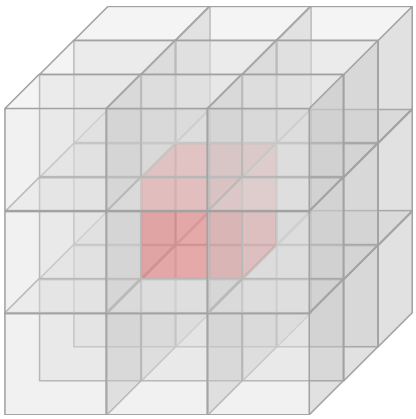


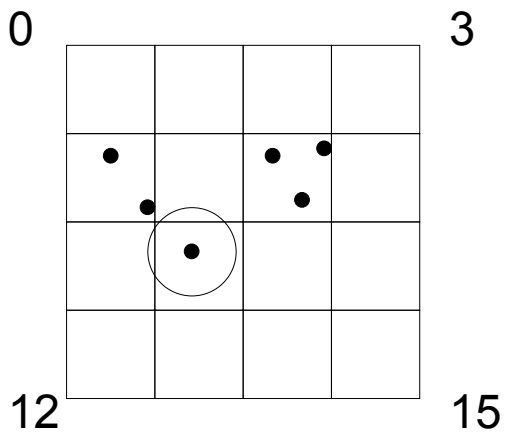
Figure 4.18: Test on neighbour cells for *collision* or, in our domain, *bimolecular reactions*.

The central point is to generate the grid in parallel on the GPU. Like explained in [90], it is easier to do it using atomic operations; however, not all GPUs support them[7]. An alternative solution is based on sorting. This solution is slightly more complex, but

[7]Actually, only the G80 family does not support atomic operations: they were added with the G90 family.

guarantees better performances.

The algorithm assigns a numerical id to both molecules and cells. During the first step it computes a hash value for each molecule, based on its cell number, and then it stores it as a pair (cell hash, particle id); the linear cell id is a good hash choice in this case. Then molecules are sorted based on their hash values, using the fast radix sort algorithm described in [89]. This creates a list of molecules ids in cell order (see Figure 4.19).



Index	Unsorted hash (cell,molecule)	Sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	
6			2
7			
8			
9			5
10			
11			
12			
13			
14			
15			

Figure 4.19: Generating a uniform grid using sorting

To be able to distribute data to the threads of the *bimolecularReact* kernel, the one that will actually check for reactions and execute them, we need to be able to find the start of any given cell in the sorted list. This is achieved by running another kernel, which uses a thread per particle and compares the cell index of the current particle with the cell index of the previous particle in the sorted list. If the index is different, this indicates the start of a new cell, and the start address is written to another array using a scattered write.

Maintaining a grid heavily simplifies the work of the *bimolecularReact* kernel: as we mentioned, we have to test for distance between pairs of molecules only inside the current cell and the 27 (3x3x3, see Fig. 4.18) neighbouring cells.

The kernel is launched with a number of threads equal to the number of molecules. Each thread calculates the grid containing its molecule. It then loops over the neighbour-

ing 27 grid cells, and then loops over the particles in these cells (using the start index computed in the previous kernel). The kernel checks if the two molecule types can react together; if so, it computes the distance between them. If the distance is less than the binding radius the reaction is performed. When the current cell index no longer equals the index of the cell we are examining, we have reached the end of the current cell.

In the case of bimolecular reactions there might be a decreasing in the number of molecules in the system: reactions like $A + B \rightarrow C$ subtract one to the global number of molecules. In order to let these reactions happen, we use the same technique described for unimolecular reactions: we write 0 at one of the reactant molecule types (in the place of B, for example); then we use a *segmented scan*, which will compact the array and will count the number of active molecules.

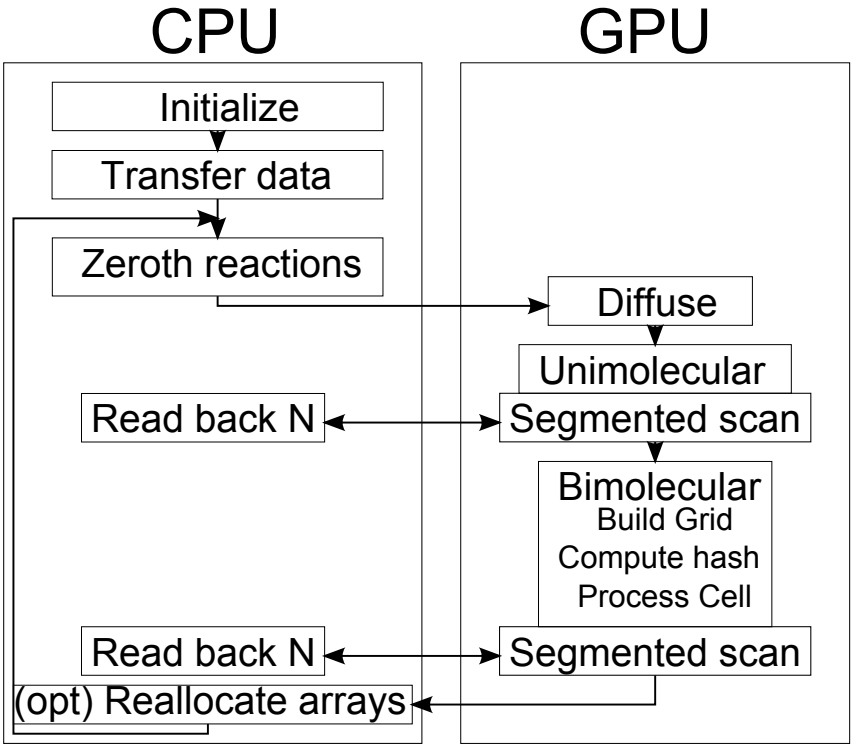


Figure 4.20: The various steps of the algorithm, showing CPU-GPU interaction.

After kernels finished to run on the GPU, the total number of molecules is computed. Device arrays are kept larger than would be needed, in order to diminish the number of re-allocation; if, however, the total number of molecules is greater than the allocated size, we re-allocate a larger space. If the GPU memory is large enough, we make a fast device-to-device copy; in this way the molecules data is always kept on the GPU, completely avoiding any transfers back and forth the CPU memory that could decrease performances.

Then, the loop starts again for the next time step. An overview of the various steps

is depicted in Figure 4.20.

4.5 Related work and Future directions

One of the obstacles on the way of systems biology is the scalability of current approaches, i.e. the ability to deal with even bigger and more complex models. Complex models are necessary to understand higher level behaviours, but need for both powerful modelling tools and efficient simulation engines to analyse them.

In this thesis we tackled the problem of designing a parallel simulator for biochemical systems; we parallelized a species-based system, developing a concurrent version of a method based on the Gillespie theory, from both a theoretical and a practical point of view. Then we exploited the huge raw computational power of modern GPUs to speed-up the execution of precise, individual based BD methods. In particular, we developed a GPU version of Smoldyn.

The design of parallel and distributed algorithms requires indeed both a strong theoretical background, in order to guarantee that the designed algorithm is equivalent to the serial one, and a good deal of experience and practical programming, in order to make it really scalable and efficient.

Systems Biology simulations as DES

The characterization of biochemical species-based simulations as DES has been explored by another group: Jeschke et. al. [118] conducted a parallel research on the same topic, focusing on the analysis of communication costs and on sizing of the window for optimistic execution in a distributed grid environment. It will be interesting to incorporate their studies and analysis of the window size to our framework, to see which are the differences between their grid-based and our HPC based approach.

Other problems we need to face are the analysis of the obtained data, whose dimension grows at an impressive rate when dealing with spatial simulations, include load-balancing techniques for workload subdivision and an analysis of the rollback mechanisms on different biochemical systems. Finally, we would like to perform an in-depth study of the performances, with different checkpoint frequencies, different number of nodes, different policy of cell allocation between nodes and different state saving strategies.

GPUs in Systems Biology

GPUs have been used to tackle a wide range of Systems Biology problems; for a gentle introduction, we recommend [178].

A particular area in which GPUs have been applied very successfully is individual based simulations; as we have seen in Chapter 2, individual based simulations directly simulates spatial interaction at a single molecule detail, and are therefore good candidates for precise simulations where spatial relationships play an important role. Following the classification in Section 2.2, we examine available simulation algorithms at different levels of abstraction, which influence both accuracy and performances.

Molecular Dynamics (MD) works at the level of the atoms; these simulations explicitly represent every detail of the chemical reaction considered, as the position and the energy of every atom in the system. MD methods map well on GPUs, and many solutions are proposed. Here, it is worth mentioning the pioneering work on Namd [216], VMD [148], and HOOMD [6].

Brownian Dynamics (BD) methods operate at a slightly coarser level of detail, where molecules have an identity and an exact position in a continuous space, but no volume, shape or inertia. Each molecule of interest is represented as an individual point. Brownian Dynamics simulation generally adopts a stochastic approach based on the solution of the Smoluchowski equation, which describes the diffusive encounter of the molecules in the solution [8, 221, 228]. The algorithm proposed here is a GPU variant of the Smoldyn method [8]. To the best of our knowledge, this is the first GPU implementation of a Smoluchowski-based method.

Recently, Januszewski et al. [116] adopted an alternative approach to perform reaction-diffusion simulation on a GPU: the dynamics of globally interacting Brownian particles is represented with the Kuramoto model. In this way, the simulation is reduced to the numerical solution of some stochastic differential equations. The integration is performed using a stochastic scheme of the 2nd order. Time steps are discrete; at each step the equations are computed and the positions of all particles are updated.

At a coarse level of detail we find **lattice-based** methods, where the simulated space is partitioned into three dimensional elements. Particularly interesting for GPU computing are cellular automata (CA) based methods. Here, space and time are discrete, and the evolution in time of the system is governed only by local information, instead of obeying to a global equation. Therefore, CA models fit nicely on the GPU model of computation. For a complete survey on CA simulation algorithm and a comparison between CPU and GPU implementation, we refer the reader to [200]. Here, we just recall two CA based methods, both of notable interest for systems biology applications and easily implemented on a GPU: Coupled Map Lattices (CML) [100] and the multiparticle model [194].

CML [99] is an extension of a CA where the discrete state values of the CA cells are replaced by continuous real values. Efficient implementation of the Gray-Scott model [165] and of the Turing pattern models [202] are obtained running CML on GPUs. They are

usually implemented as partial differential equations that describe the concentrations of chemical reactants at each lattice site over time; their GPU implementation consists of a single data stream where the concentrations of the chemical species are stored in different channels of a single texture that represents the discrete spatial grid. This stream serves as input to a kernel, which implements the partial differential equations in a discrete form.

The multiparticle diffusion model is more complex and more realistic. In this model, multiple particles per lattice site are permitted; particles move in a stochastic way by following independent random walks between positions in the lattice. Brownian diffusion is therefore modelled as a series of independent random choices for the movement of particles on a regular, uniform grid. The algorithm described in [194] implements a multiparticle model on GPU in an efficient way using a novel data structure; the authors apply the method to a 3D model of in vivo diffusion inside the E.Coli cell.

Finally, **the Agent Based Model (ABM)** generalizes the CA model. ABMs are computational representations of dynamic systems where a number of individual, autonomous constituent entities (called Agents) interact locally in order to recreate a higher level, group behaviour. This ability to simulate the emergent behaviour of complex systems from local interactions makes agents attractive for systems biology. Indeed, ABMs have been used to model and simulate inflammatory cell tracking, tumour growth, intracellular processes, wound healing, morphogenesis, microvascular patterning, pharmacodynamic and tuberculosis (see [153] for a survey).

Even if Agents are concurrent, independent objects, historically only sequential simulation algorithms have been implemented. One of the first parallel implementation running on graphics hardware was done by De Chiara et al. [39]. Notably, they study the distributed behaviour of a flock, a wide studied problem in systems biology. Recently, several research efforts concentrated on ABM simulation on GPUs; Perumalla et al. [166], for example, used an extended cellular automata approach to simulate ABMs on the GPU. However, being based on CA and therefore on lattice sites, they have limitations both in the number of agents and on replications. Two groups, in particular, pushed the state of art in large-scale ABM simulation, by extending existing ABM frameworks with rich and complete support for simulation on a GPU: Richmond et al. with FLAME [192] and D'Souza et al. with SugarScape [63]. They rely on existing agent frameworks supporting a number of key ABM features, as, e.g., birth and death allocation, agent replacement and movement, pollution formation and diffusion, collision detection. Of particular relevance for systems biology is the application of SugarScape to the 3D simulation of granuloma formation in TB infection [64]. The authors showed that ABM frameworks running on GPUs are flexible and mature enough to run complex simulations, with a speed that is three orders of magnitude faster than the sequential algorithm.

However, not all the applications are well suited for a GPU implementation and the performances vary considerably depending on the biological system considered. In Table 4.2 we relate the cited works with the obtained speedup; the table reports the improvements of combining GPUs and CPUs over CPUs only configurations, together with the GPU and the software package used. The column speedup refers to the simulation execution time; for instance, a 10x speedup means that the simulation time required by a CPU only system is 10 times the one of a CPU/GPU configuration. These data have to be considered carefully, since the way in which performance measurements are taken varies greatly; furthermore, GPU performances vary dramatically even within the same generation; Figure 4.21 reports a comparison of the GPUs listed in Table 4.2 in terms of GFLOPS[8].

Therefore, speedup values have not the same value since, for example, the GXT280 board outperforms 8600M GS by three orders of magnitude. For this reason Table 4.2 has to be considered only as a sketch of GPUs computing power without any intention of comparing algorithms or implementations.

	Method	**Software**	**GPU**	**Speedup**	**Source**
Species Based	SSA	Multiple Simulations	8800GTX	50x	[144]
		Single Simulation	8600M GS	2x	[61]
Individual Based	MD	Namd	8800GTX	10x	CUDA Zone[9]
		VMD	n.a.	125x	CUDA Zone
		HOOMD	n.a.	15x	CUDA Zone
	BD	SDE	Tesla C1060	675x	CUDA Zone
		GPUSmol	*GT220 - GTX280*	*10-25x*	*this thesis*
	CA	CML	Xenos	25x	[202]
	ABM	FLAME	9800 GX2	250x	CUDA Zone

Table 4.2: GPU performance comparison

Even if we did not considered them in this chapter, we added to the table a couple of interesting species based systems implemented on a GPU. We omitted ODEs systesm, for which a number of GPU based implementations exists, as we focus on stochastic simulations only.

As we already pointed out, these systems do not present really impressive performances, mainly because the Stochastic Simulation Algorithm (SSA) is hard to parallelize

[8]One GFLOP = one billion of FLoating point Operations Per Second
[9]Available online at `http://www.nvidia.com/object/cuda_home.html`

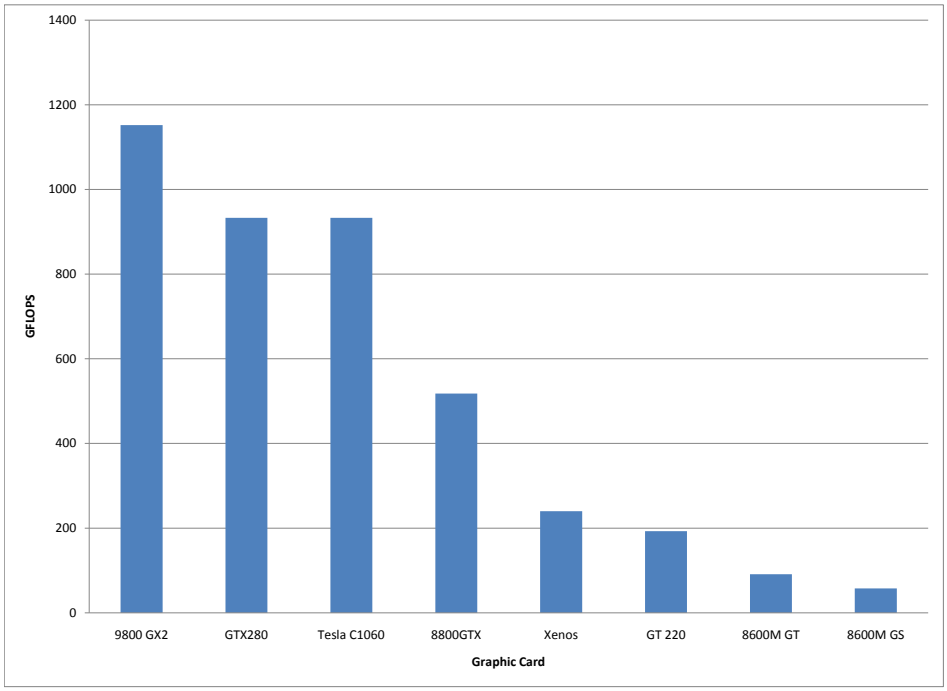


Figure 4.21: Performance comparison between different GPUs (in GFLOPs). Notice that the best performer (9800 gx2) and the worse (8600M gs) belong to the same family (g80-g90)

(see Section 4.2.5 for a thorough analysis of the problematic). The situation is better in the case of MRIPs (multiple replications in parallel) employed in [144], where 50 simulations run in parallel on a GPU require the same time of a single simulation performed on a CPU. The speed-up is even less impressive if we consider that single precision floating point numbers are used in both cases (single replication [61] and multiple replications [144]). Double precision numbers are only available from the current generation (g100-g200) of Nvidia GPUs, and with a huge performance penalty: the speedup is roughly one order of magnitude less in this case, as only one SPU out of eight is able to deal with double precision arithmetic.

Nonetheless, the result obtained in [61] is worth noting: unlike our DES based approach, the authors reorganize the structure of SSA in order to reduce the complexity in space of the algorithm. In this way it is possible to split the reactions set among blocks and to obtain a certain level of parallelism inside a single simulation. In addition, the SSA requires generating a large quantity of random numbers, a time consuming task; using GPUs as a fast random number generator reduces the time needed for a run.

Individual based systems offer specific tools to describe those models where many details, as the position and the mass of each element in the model, are needed. We first examined molecular dynamics methods that map naturally on GPUs. The methods presented offer good performances, especially the VMD software. The field of MD on GPUs is receiving great attention from the community and new applications are released every month. Instead, it is quite surprising that Brownian Dynamics methods are not supported because their nature fits well with the streaming programming paradigm, as we demonstrated in Section 4.4. Besides our algorithm, the only very recent and valuable exception is [116], that achieves an impressive speedup of 675x, using stochastic equations.

Finally, we discussed lattice based methods and agent based models. They have reached a good maturation, both in the applications and in the theory supporting GPUs. A key feature is the possibility of using the GPUs computing power without specific programming skills. For instance, the FLAME framework uses an XML specification language for Agents that is automatically compiled into CUDA code. This makes the 250x speedup more interesting, because this computing power is available to all the ABM community.

4.6 Summary

In this chapter we introduced two simulation methods, focusing on their scalability. In particular, we considered the treatment of spatial aspects, which are fundamental when we want to go beyond localized, single pathway models, and parallel execution of simulation algorithms. The increase in model size and in method complexity, in fact, requires us to create efficient algorithms that fully utilize modern multi- and many-core architectures, scaling efficiently with the dimension of the problem. The result are a species-based simulator, Redi, which is based on a spatial extension of the Gillespie algorithm with state dependent diffusion coefficient, implemented as a Parallel DES system, and a massively parallel version of the individual-based method used in Smoldyn [8].

Chapter 5

Visualization

Visual techniques for systems biology are a current topic of research, where they are useful at many levels. In particular, visual techniques can help in understanding data in the biological domain. Understanding usually is a multi-step process; extracting information is the first step in this process that leads to new knowledge. This is the role of visualization.

Visualization, in a broader sense, is referred to any process used to communicate both abstract and concrete ideas using a graphical language. In the context of computer science, visualization is mainly concerned with a systematic exploration of data and information.

In particular, *scientific visualization* is the process of manipulating *data* -composed of symbols taken from some grammar or language (e.g. series of doubles, graph structures, grid structures, DNA strings, ...)- through transformation, selection and graphical representation, in order to extract and present to the user *information*, using visual cues and graphical techniques. *Information*, processed data suitable for exploration and analysis, can provide the user answers to who, what, where and when questions.

The final goal of visualization is to gain insight and understanding into the information space, in order to help a scientist in understanding the data and give an answer to the most important question: the *how*, that gives new *knowledge* on the phenomena under study (Fig.5.1, [38]).

Scientific visualization is maybe the first known application of visualization to human knowledge: the visualization of experiments and phenomena is as old as Science itself.

The importance of Scientific Visualization is even stronger today: the visual analysis of data produced by simulations on both standard and high performance computers is now a fundamental step in the scientific process. The increasing complexity of simulation models and size of datasets resulted in longer and longer analysis times. Scientists need to comprehend in a fast way the meaning behind simulation results and datasets in order to get insight into their models. Visualization methods and technologies are able to reduce analysis time, and help scientist in dealing with large and complex models.

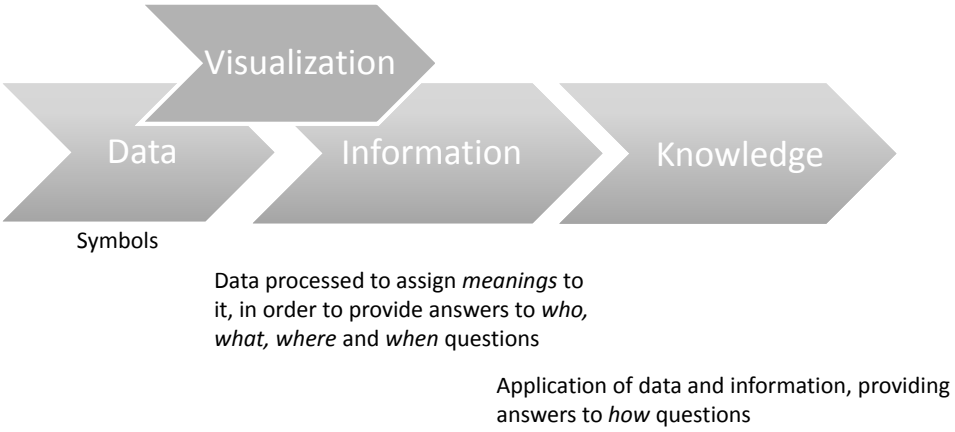


Figure 5.1: The visualization process

5.1 Space

The need for visualization is even stronger in the case of spatial simulations; some authors go to the extent of defining scientific visualization as "primarily concerned with the visualization of three dimensional phenomena (architectural, meteorological, medical, biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources, and so forth, perhaps with a dynamic (time) component" (M. Friendly). While we prefer the more general definition given in the introduction, as we believe that visualization techniques can be applied to a wider range of scientific data, it is undeniable that in order to understand three dimensional phenomena we need to visualize them: our perception of space is strongly related to our vision.

5.1.1 COVISE

For our research on visualization of spatial biochemical simulations we made use of an existing computer graphics framework: COVISE. During my PhD I had the opportunity to do an internship within the Visualization group at HRLS Stuttgart, the main group doing research and development on COVISE.

COVISE (Collaborative Visualization and Simulation Environment) is an extensible distributed framework that integrates simulations, post-processing and visualization functionalities with support for collaborative working [186]. In COVISE an application is divided into several processing steps, represented by COVISE modules, that can be developed, arbitrarily combined and executed across different computers, including parallel and vector computers.

COVISE supports flexible rendering through OpenCOVER; this support ranges from

visualization on standard desktop PCs to projection in virtual environments (powerwalls, CAVEs, ...) were users can analyse datasets and simulations results intuitively in an immersive environment, to augmented reality prototypes.

Rendering modules include both volume rendering (using the VIRVO rendering engine [205]) and fast rasterization of possibly distributed mesh objects.

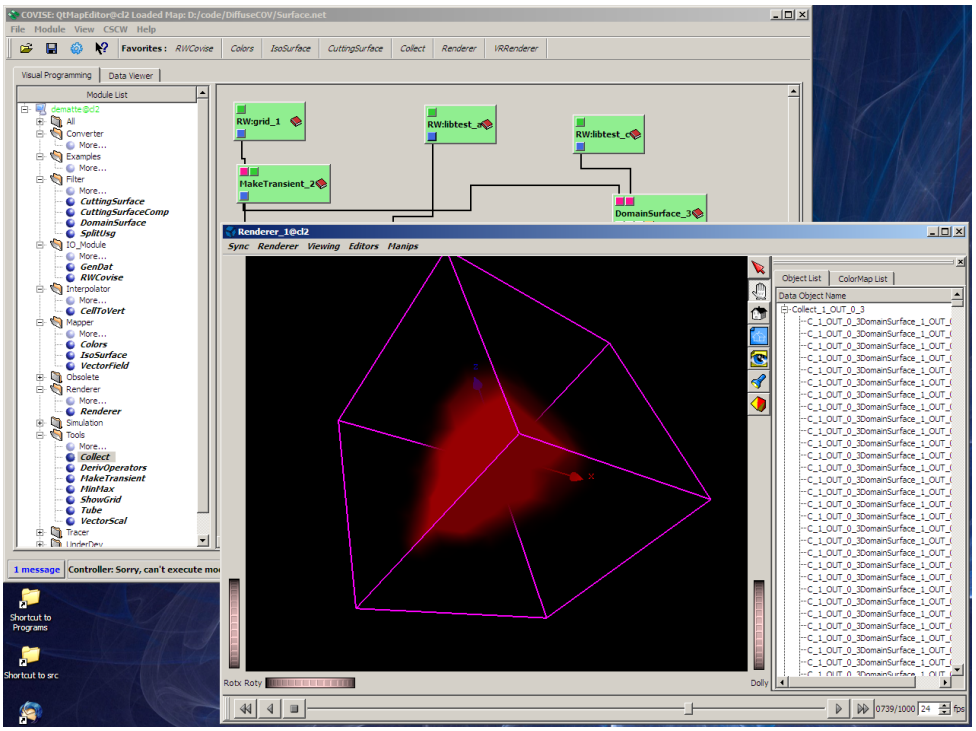


Figure 5.2: Composition of modules in the COVISE pipeline.

The usage of COVISE and OpenCOVER allowed us to concentrate on some parts of the visualization process and of the graphic pipeline (see Figure 5.2), leaving most of the details and the burden of data processing to the framework.

5.1.2 Volume rendering

Volume rendering is a technique used to display a 2D projection of a 3D discretely sampled data set [77].

A 3D data set is typically a regular volumetric grid, with each volume element -called *voxel*- represented by one or more values.

When measured, the value is usually obtained by sampling the immediate area surrounding the voxel, like in the case of 3D scanners (CT, MRI, or MicroCT scanners), where a group of 2D slice images with a regular number of pixels are acquired in a regular pattern. When obtained through a simulation, the value is given by the simulated

quantities at each position in the regular grid.

A *volume renderer* is quite often implemented in a *direct* way, where every sample value is mapped to an opacity value and a colour. The mapping is done with a *transfer function* which converts values (or tuples of values) to an RGBA (red, green, blue, alpha) value.

Different rendering techniques are used to compose RGBA values from multiple voxels and project their combination onto a correspondent pixel on the frame buffer; the two most diffused techniques take advantage of the processing power of GPUs. In particular, the oldest technique uses *texture mapping*, using the very fast texture units available in the *Rasterization* stage, while the most recent one uses GPU accelerated ray casting.

The technique of ray casting for volume rendering is derived directly from the rendering equation. A ray is cast for each desired image pixel: the ray starts at the eye point and passes through the image pixel on the imaginary image plane floating in between the camera and the volume to be rendered. The ray is sampled at intervals throughout the volume; at each sample point the data is interpolated, then the transfer function is applied to the interpolated value, and the obtained RGBA sample is accumulated. The accumulated RGBA value written to the current image pixel, and the process is repeated for every pixel to produce a complete image on the frame buffer. This technique is time consuming, but provides results of very high quality; fortunately, direct volume rendering is an extremely parallel problem, and therefore in the last year several fast algorithms for GPU ray casting were developed [115, 214]. *Virvo*, the volume rendering component of OpenCOVER, is able to use both techniques.

A crucial point of direct volume rendering is the definition of the *transfer function*: transfer functions are required in order to pull out specific features of volume datasets, as they will dictate how the volume will look like on the 2D screen. Therefore, a simple, interactive way of defining transfer functions is needed.

A classical Transfer Function Editor uses a one-dimensional plane for colour mapping. The unique, horizontal axis is used to represent the domain values (often in the 0..255 range); colour markers are inserted at user defined values on this axis. Values at these points are associated directly with the given colour, while colour at value points between two colour markers is computed through interpolation. Transfer Function Editors for alpha values typically use a two-dimensional plane, where the x-axis is used to represent the domain values, and the y-axis to represent alpha (opacity) values.

The original Transfer Function Editor embedded in OpenCOVER and Virvo used separate colour and opacity transfer function. The two functions could be edited from within a virtual environment, and changes were reflected in the image instantaneously, allowing for experimentation; however, a more powerful and immediate editor was needed

in our case.

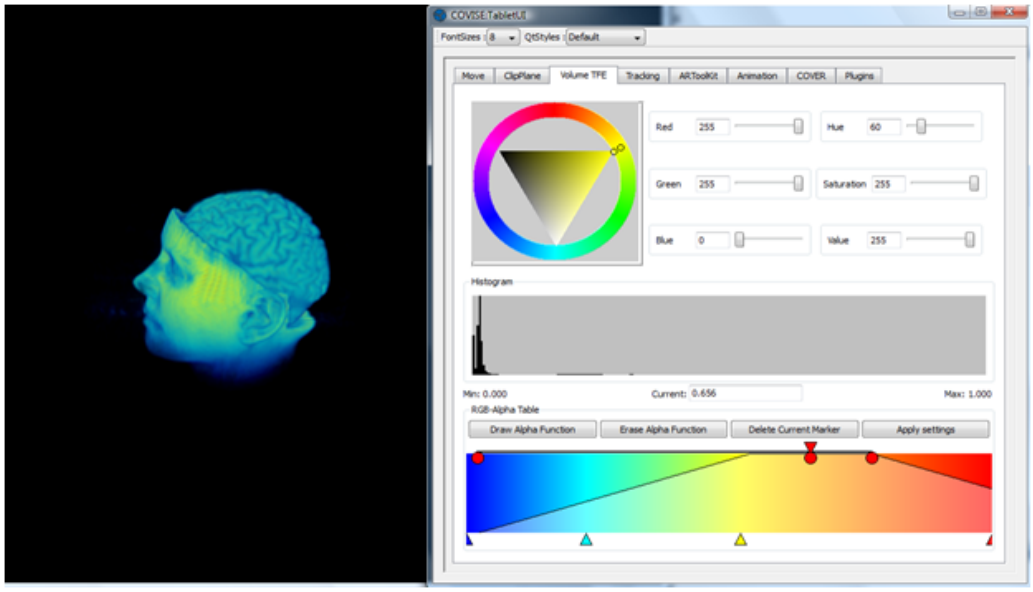


Figure 5.3: The Tablet UI Transfer Function Editor. From top to bottom: colour chooser for the selected colour marker, histogram for the distribution of volumetric data, the interactive editor interface.

With Uwe Woessner of the HLRS at the University of Stuttgart, we studied a more user friendly transfer function editor. The editor combined in a single view colour and opacity information. The user could select how input values map to colour and alpha using *markers*. Markers are freely moveable; alpha markers are equipped with anchor points to adjust the shape of the trapezium (height, bases) that will define the alpha values around the marker. The user can add (and remove) an arbitrary number of markers, to fine-tune the transfer functions (see Figure 5.3). Unlike the original Transfer Function Editor, which was implemented as a 3D widget inside OpenCOVER, the new editor was implemented as a widget for a Tablet PC user interface. The tablet UI can still be used inside immersive environments, but it is is easier to control than a 3D widget, which is controlled using tracking when displayed in an immersive environment and therefore can be less precise.

Like in the original Transfer Function Editor, we reflected changes to the function in real time, allowing for interactive exploration of different combinations. To add even more flexibility, we took advantage of the Tablet pen, using it not only to move and adjust markers, but also to draw free-form functions (Figure 5.4): the pen can be used to directly set the alpha value corresponding to each data value in the domain, allowing a fine-tuning of opacity and transparency that allows to highlight particular features of the volumetric data set.

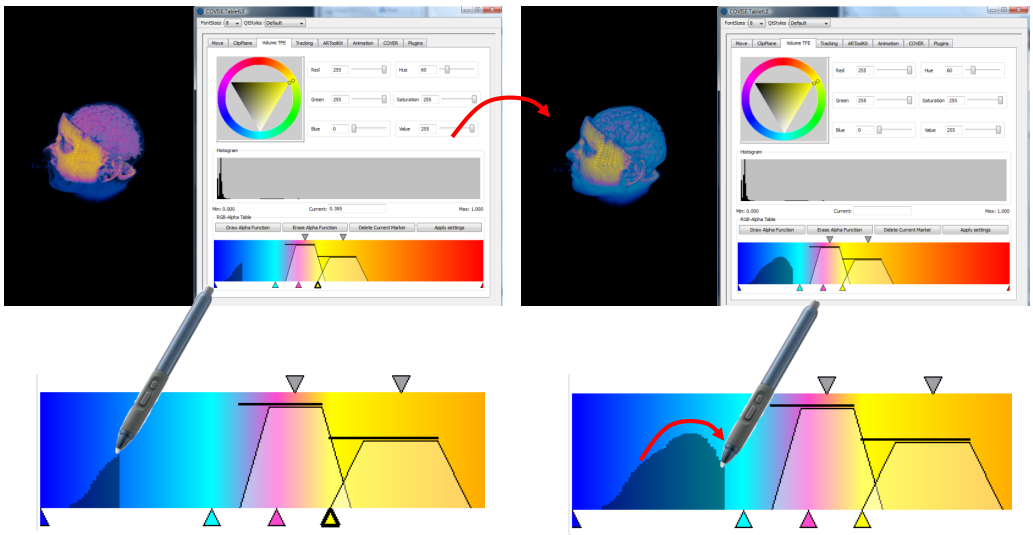


Figure 5.4: The free-form alpha function editor allows to draw or erase the alpha function using a pen

Another addition to the transfer function editor was the ability of setting functions for multi-variate volume datasets. In multi-variate volume data sets, each voxel in the volumetric grid is associated with more than one value. This case is not very frequent in the case of data acquired through scanners (even if there are some notable exceptions, like the Visible Human Project), but it is more frequent in the case of simulations, where each voxel can contain different concentrations of several species. In our case, Redi (our reaction-diffusion simulator, see Section 4.3) was configured to produce volumetric data with one or more variables; in many occasions, the scientist may be interested in showing more than one species at the same time, or even better, showing when and where a particular combination of reactants and reagents is present in the simulated space.

In this case, a multi-dimensional transfer function editor is needed. A multi-dimensional transfer function editor can associate multiple values with a single RGBA value; therefore, any combination of input values is mapped to colour and opacity information that will be used by the volume renderer.

To maintain complexity to a reasonable level, we restricted ourselves to two dimensional transfer functions. This meant re-define the various concepts and objects with one additional dimension: the colour function is now to be defined on a two-dimensional plane using 2D points; alpha markers are also centred on 2D points, and have a 3D shape. So instead of having trapezoid or Gaussian shapes, alpha markers are realized as pyramids and bells. However, we choose to visualize the widgets from above, using an orthographic projection on the bi-dimensional colour plane (see Figure 5.4). The alpha values defined

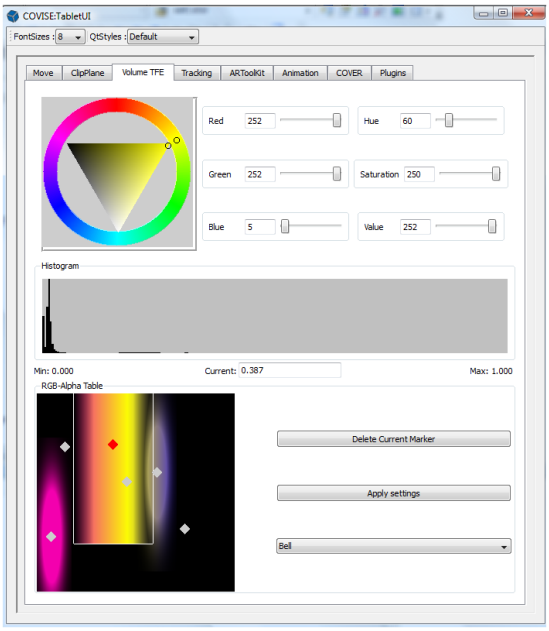


Figure 5.5: The multi-dimensional Transfer Function Editor: three colour markers (magenta, yellow and blue) are used to define the background colour, and three alpha widgets (two Gaussian bells and one pyramid) are used to define the opacity values.

by the widgets are immediately mapped onto the underlying coloured background; in this way it is possible to place and control the markers without using a complex and potentially confusing 3D representation.

We experimented the usage of both editors on a variety of preset data (CT scans, mechanical simulations) available at the HLRS Stuttgart; we also used them to define transfer functions for Redi simulation results. A picture taken from an interactive session on the testing using Redi datasets is displayed in Figure 5.6.

5.1.3 Isosurfaces

An alternative form of visualization for volume datasets is to use *isosurfaces*. An isosurface is a surface that represents points of a constant value within a volume. Isosurfaces are computed by extracting a polygonal mesh from volumetric data; the mesh will crosses the three-dimensional scalar field at the specified *isovalue*.

Isosurfaces can be drawn very quickly, but usually the extraction algorithm requires some time. We implemented a GPU algorithm for Isosurface extraction based on Marching Cubes and Marching Tetrahedron [77]; the algorithm process data entirely on the GPU, therefore any memory bandwidth bottleneck is avoided. For more information on GPUs architecture and their programming model, the interested reader may refer to Appendix C.

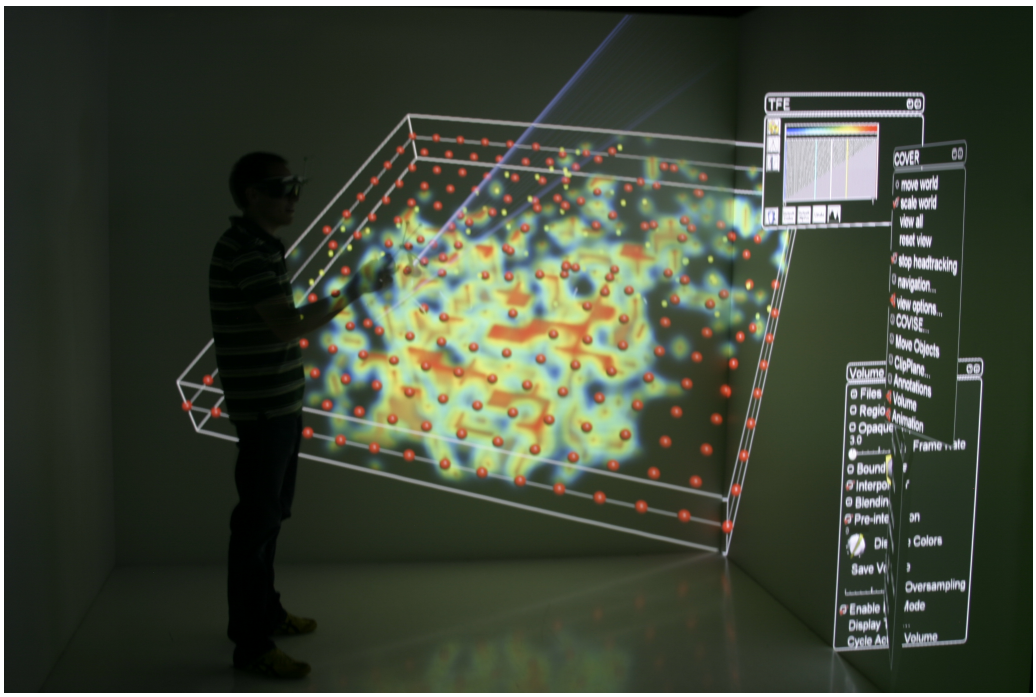


Figure 5.6: Volume rendering of a Redi simulation inside HLRS Stuttgart CAVE

The implementation is written in CUDA, and the algorithm is divided among of a set of GPU kernels[1], one for each stage of the process. Furthermore, we designed the kernels to support the mesh in COVISE unstructured format throughout the process; in this way we avoid space and time consuming conversion, and we keep memory utilization low.

COVISE allows to store a mesh in an *unstructured grid*, a grid made of different, non regular polygonal elements. Each voxel can be a different polygon (tetrahedron, cube, pyramid, diamond...) and each polygon can have different dimensions. Furthermore, the grid does not need to be regular (e.g. same number of voxels along a given axis). In contrast, a regular grid is composed of voxels of the same shape and dimension, displaced in a regular pattern. Typically, unstructured grid are more flexible and more compact, and are therefore widely used in simulations.

An unstructured grid needs three vectors to represent the grid structure, in addition to the vectors of coordinates and of scalar values associated to each coordinate. The needed data structures are listed in Tab. 5.1.

The computed mesh is saved directly in GPU memory as an OpenGL Vertex Buffer Object with an associate Index buffer. In this way, the representation is kept as compact as possible for fast rendering of the isosurface mesh. Obviously, these data structures need to be allocated on the device memory (using *cudaMalloc*) and filled with data from

[1]See Appendix C

Identifier	**Description**	**Length**
Type List	element type list (tetrahedron, hexahedron, pyramid and the other polygons supported by COVISE)	M
Elem List	list of offsets for element i inside the connection list	M
Conn List	connection list (index buffer pointing into the coords_{x, y, z} arrays)	M
vertexes	The list of vertex coordinate	N
values	The scalar values associated with each coordinate	N

Table 5.1: Unstructured grid data structures, allocated on global device memory

the host memory (with *cudaMemCopy*).

Since our grid is composed by different polygonal elements, the first step is to *classify* them. using the *classifyElements* kernel. This kernel takes as input the complete geometry of the unstructured grid and the scalar field. One thread is run for each element in the element list.

- **Input**: Type List, Elem List, Conn List, values, and an isovalue;
- **Number of threads:** one thread per element
- **Output:** element classification arrays; number of neighbours (per node); relevant edges matrix

The purpose of this kernel is to perform a classification of the type of elements, in order to compute how many elements of each type are present and where they are located, as an offset inside the Elem List array. Furthermore, for every element it tests if it cross the isosurface -and therefore the isosurface mesh will cross that element-, and if so how many vertexes are required to generate the mesh triangles for that voxel.

The number of vertexes that will be required in order to generate the isosurface is obtained using a look-up table, held in texture memory, different for each element type. This information is stored in the *element classification arrays*.

In addition, this kernel produces a 2-dimensional matrix. Each cell in the matrix, addressed by $(n1, n2)$ contains two data: whether these two nodes (which are contiguous vertexes of an unstructured grid element) are on an edge that will generate an isosurface

mesh vertex, and if so which is the offset of node $n2$ in the $n1$'s list of neighbours. This information will be needed for the computation of normals.

The last output of the kernel, the *number of neighbours*, is stored in an array of length equal to the number of nodes (i.e. vertex indexes) in the original *unstructured grid*, contains the number of neighbours for each node. It is used in conjunction with the 2-dimensional matrix: the map at $(n1, n2)$ contains a two element structure. The first element allows to find if there is a vertex-edge between $n1$ and $n2$ already; the second one, the actual number of $n1$ neighbours. So, when it is found a new $(n1, n2)$ couple, the counter at position $n1$ in the *number of neighbours* array is increased; the old value is recorded as an offset in the map at $(n1, n2)$. A successive kernel will use this information to build the list of $n1$ neighbours and $n2$'s offset inside that list.

The *element classification arrays* and *number of neighbours* array are passed to the CUDPP routine *cudppScan* which will performs a parallel sum scan (for more information about the parallel sum scan and CUDPP, see Section 4.4.1 and [207]). Briefly, *cudppScan* does a cumulative sum of elements, which is useful to obtain both an array of offsets and the total number of elements, used to obtain the exact size of buffers that are allocated before invoking successive kernels:

- **Input:** element classification, number of neighbours (per node);
- **Number of threads:** automatic
- **Output:** number of voxels crossed by the isosurface (per element type); number of vertexes the generated mesh will produce; number and offset of unique vertex coordinates (size of vertex and normal buffers).

In particular, the number of vertexes is used to compute the size of the OpenGL index buffer, while the scan on the *number of neighbours* array gives the number of *unique* vertex coordinates, that will be used as the size of OpenGL vertex and normal buffers.

The following kernels to be called are *generateTriangles*{*Thetra/Hexa/Pyramidal/...*}:

- **Input:** number and offset of unique vertex coordinates (size of vertex and normal buffers); *vertexes*, the original vertex positions in the grid; the grid geometry information (Type List, Elem List, Conn List); the *values* scalar field; an isovalue; the relevant edges matrix computed by the previous kernel; the number of voxels crossed by the isosurface;
- **Number of threads:** one per active Thetraedral/Hexahedral/... voxel
- **Output:** a vertex buffer that contains the (unique) interpolated vertex positions; the isosurface vertex positions; the number and offset of unique vertex coordinates.

Each kernel deals with a different element type, using the classification information and offset arrays provided by the previous kernels. They generate the isosurface mesh coordinates and store them in separate vertex and index buffers. Positions inside the buffer, and whether or not two generated vertexes are the same (fall on the same voxel edge) are obtained thanks to the *the relevant edges matrix* and the compacted *neighbour* array. The vertex positions inside the vertex buffer are indexed by the corresponding index buffer (one per element type).

Finally, computation of *normals* is carried out by three kernels: *createNeighbourList*, *createNormals1* and *createNormals2*

The creation of a neighbour list, carried out by the first of these three kernels, is necessary to understand between which couple of vertexes it is necessary to perform interpolation.

- **Input:** the isosurface vertex positions (index buffers);
- **Number of threads:** number of vertexes in the isosurface mesh;
- **Output:** a compacted list of neighbours for each node;

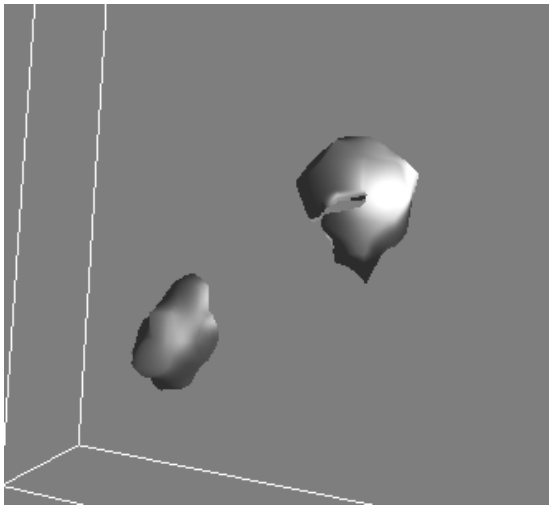


Figure 5.7: The isosurface extracted from a Redi diffusion model.

The index buffer already contains node "numbers" (positions into the vertex buffer) in a non-duplicated way. Besides, indexes are *ordered* in such a way that is possible to see which one belongs to a given triangle.

The process of generating normal staring for the value field and from the neighbour list is done in two steps, by two kernels; the first one, *createNormals1*, computes the normal gradient per-vertex:

- **Input:** the vertex buffer, which contains the (unique) interpolated vertex positions; the isosurface vertex positions(index buffers);
- **Number of threads:** number of unique vertexes;
- **Output:** the normal gradient per-vertex;

The second one, *createNormals2*, interpolates the normal gradient per-vertex between neighbour vertexes and fills the final normal buffer:

- **Input:** the vertex buffer, which contains the (unique) interpolated vertex positions; the isosurface vertex positions(index buffers); the compacted list of neighbours computed by the *createNeighbourList* kernel; the normal gradient per-vertex;
- **Number of threads:** number of unique vertexes;
- **Output:** the interpolated normals, stored in an OpenGL *normal buffer*;

The final result is a isosurface mesh complete with normals; the normal computation produces a mesh that seems much more detailed and visually pleasant (see Fig. 5.7). Our algorithm for isosurface computation, for medium sized grids (¡ 1M elements), performs at interactive frame rates.

5.2 Networks

A recurrent visualization problem in systems biology is drawing graphs in a custom, controlled, meaningful way. Systems biology commonly uses *networks* or *graphs* to represent many different aspects; the goal is to understand the interaction dynamics, and networks are the most common way of representing interactions.

Networks (or *graphs*, in computer science terms) are used to describe reaction networks, ecological food webs, dependency graphs for causal relations, or even complexes and polymers. Graphs are a convenient way of representing any kind of relation between elements in a domain; for this reason, at CoSBi we developed a software (CoSBiLab Graph [71]) that is able to read, visualize, save, compute metrics and run algorithms on any kind of graphs, even if most of the features are tailored to life science.

The task of visualising a graph is relatively simple: nodes are draw on a 2D canvas, connected by edges. The position of nodes is assigned by *layout algorithms*. Layout algorithms are usually general: they work on any type of graph, and try to displace node so that they meet some general criteria. For example, they may group together highly connected nodes, or reorder nodes so to minimize edge crossing. The result is usually a

visually pleasing graph; however, the chosen layout does not always add *information* to the graph data.

A first example is given by reaction networks. Visualization of reaction networks is really important in the case of models described using process algebras, like in the case of BlenX. As we mentioned in Chapter 2 and 3, a process algebra model describes the possible interactions between components, not the whole reaction network, as it is gives an operational description of the single components and their interactions. The advantage is a compact model, whose reaction network does not need to be wholly specified, but it is unfolded during the *execution* (usually, a stochastic simulation) of the model. Therefore, complexity is somehow transferred from the modeller to the software that executes the model. However the final reaction network, unfolded during simulation, needs to be *visualized*, as it helps the user to understand both if the model was correct and if the simulated reactions are those he expected. Indeed, the inspection of the reaction network can reveal secondary, but significant, interaction between components that were not clear or explicit before, leading to new *knowledge* on the model or on the problem at hand.

5.2.1 Reaction Networks

In a reaction network produced by a simulation, each *class* of entities already present in the model, or generated during the simulation, is a graph node, and if an interaction had taken place between two *classes* of nodes, an edge between their nodes is added to the graph. Reaction networks can be quite large, and show complicated interactions. Unfortunately, standard layout algorithms do not help in making the picture clearer.

Take as an example the NfKB and MapK reaction network showed in Figures 5.8 and 5.13 respectively: by carefully inspecting nodes and edges it is possible to extract *information* about the behaviour of the components, and about the key interactions, but the process requires time and patience.

With Danyel Fisher from Microsoft Research Redmond, we designed a new way of displaying reaction networks, which uses our knowledge of the domain in order to layout nodes in a way that gives more information to the user. The starting point was asking ourselves and other researchers which are the most common network structures in reaction networks. We realized that *cycles*, for example, are very important: many enzymes and many molecules undergo reversible transformations, and therefore they change state (going from active and inactive, for example) cyclically. Another common structure is represented by *interacting cycles*: biochemical pathways are almost always interconnected. For example, an enzyme that is cyclically activated and deactivated in one pathway can take part in the transformation of another molecule in a different part of the model or in another pathway altogether. A third frequent possibility is that a chemical will undergo

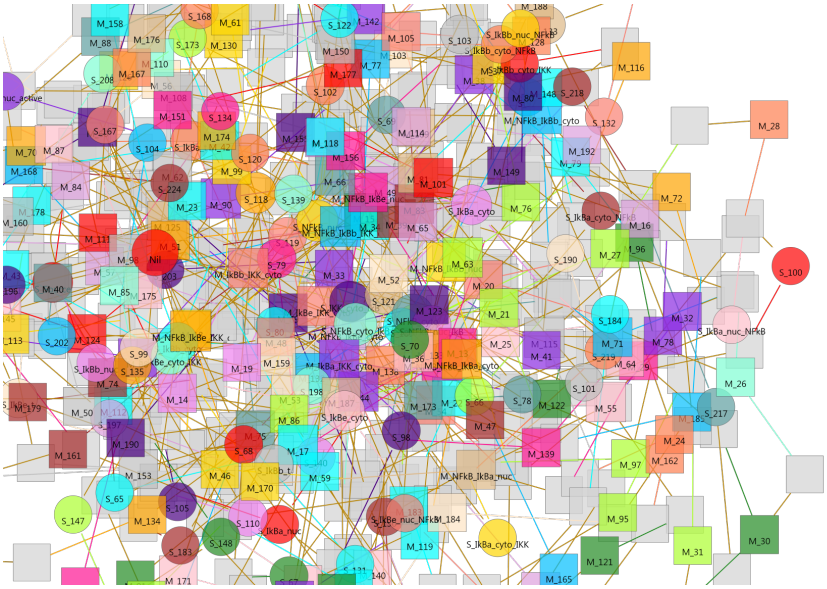


Figure 5.8: The NfKB reaction network displayed as a graph

a series of non-reversible transformations, leading to a *chain* of interactions. A fourth possibility is represented by *star* or daisy-shaped structures: a single molecule can be used in many different reactions, leading to many different products. Finally, we classified all the other possible structures as *complex networks*. For all but the last kind, for which we used a standard force-directed graph layout, we devised a possible meaningful graphical representation. The result is shown in Figure 5.9.

We enriched the structures with some additional information. For example, bimolecular reactions involve two reagents transforming into one or two reactants. In this case, instead of introducing a separate node for the reaction as we did in Graph, we inserted a single edge for each structure, and then we logically connected it to the other half of the reaction (belonging to a different structure) using a *reminder*. Furthermore, we assigned to each structure a colour, and used it in the reminder, so that the user can immediately connect them visually (see Fig 5.10(a)). Another information we inserted is about rates, or speed of that reaction: if a reaction happens more often, the edge(s) representing it are ticker. To highlight this concept better, we introduced also *speed ticks* on the edges: the highest the number and frequency of ticks, the faster is the reaction (see Figure 5.10(b)).

We developed an application, *Rings*, which extracts these structures from the BetaSim output files and displays them using our graphical representation, and tested it on several pathways. Here we show, as an example, the MapK signalling pathway (Figures 5.13 and 5.14), the ERK signalling pathway (Fig. 5.15), the NfKB pathway (Fig. 5.8, 5.11 and 5.12).

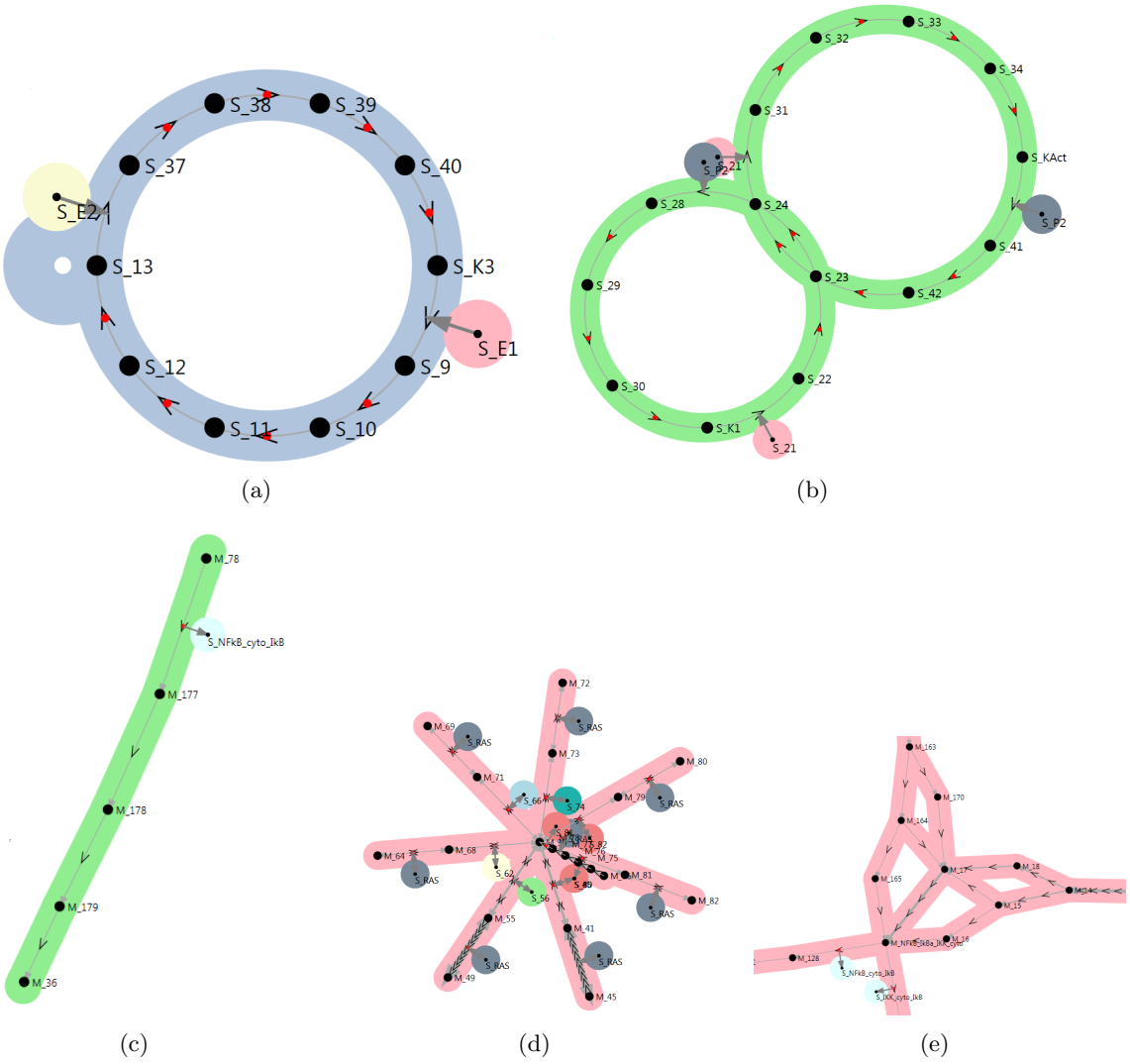
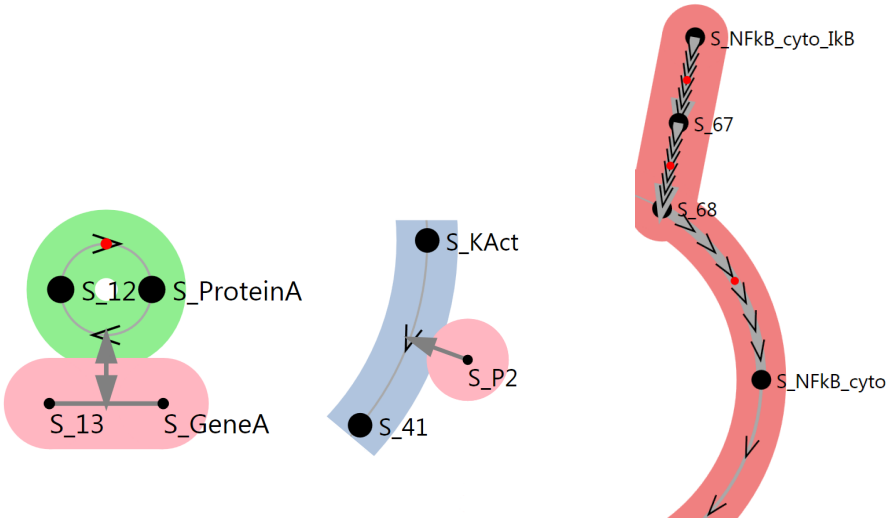


Figure 5.9: The graphical representation of common network structures used in *Rings*.

It is possible to notice how in many cases the reaction network is completely described in terms of the first four structures; only in the case of NFkB a *complex network* appears. The advantages of this visualization become clear comparing Figures 5.8 and 5.13 with Figures 5.11 and 5.11. For example, the cyclic behaviour of K, KK and KKK becomes apparent, as well as their relationship and the identical behaviour of KK and KKK, which are activated by multiple phosphorilation and are therefore categorized as interacting cycles (one for the first phosphorialtion-dephosporilation and one for the second phosphorialtion-dephosporilation stage).



(a) (b)

Figure 5.10: Reminders for bimolecular and monomolecular reactions (a) and rate 'ticks'(b).

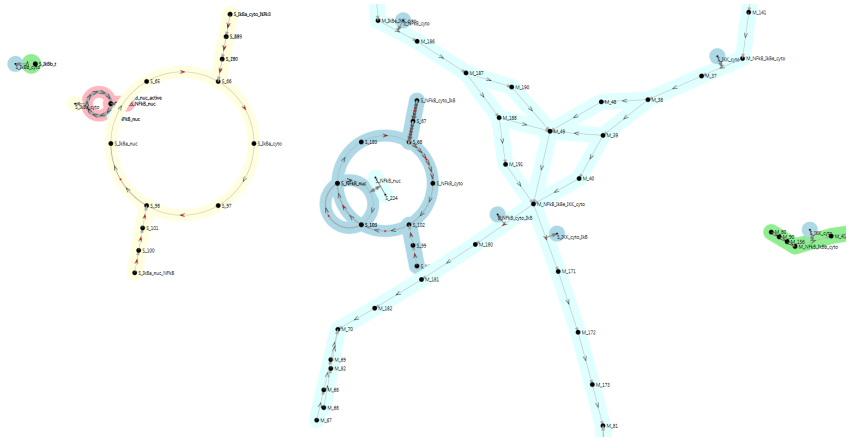


Figure 5.11: The NfKB reaction network displayed with the Rings application. Cycles, networks with multiple interacting cycles and lines of monomolecular reactions are displayed separately as functional units.

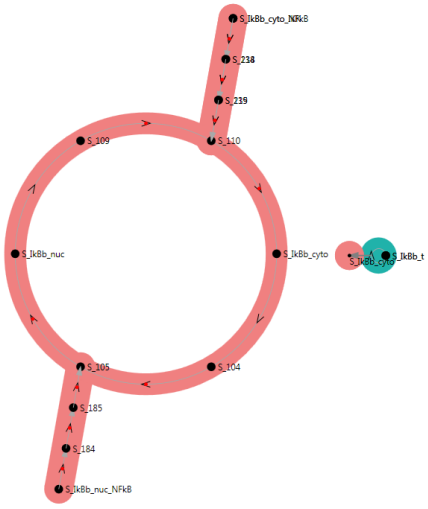


Figure 5.12: A particular of the NfKB pathway, representing the reversible process of IkB enucleatipon (the transportation of IkB from the cytoplasm to the Nucleus and vice-versa)

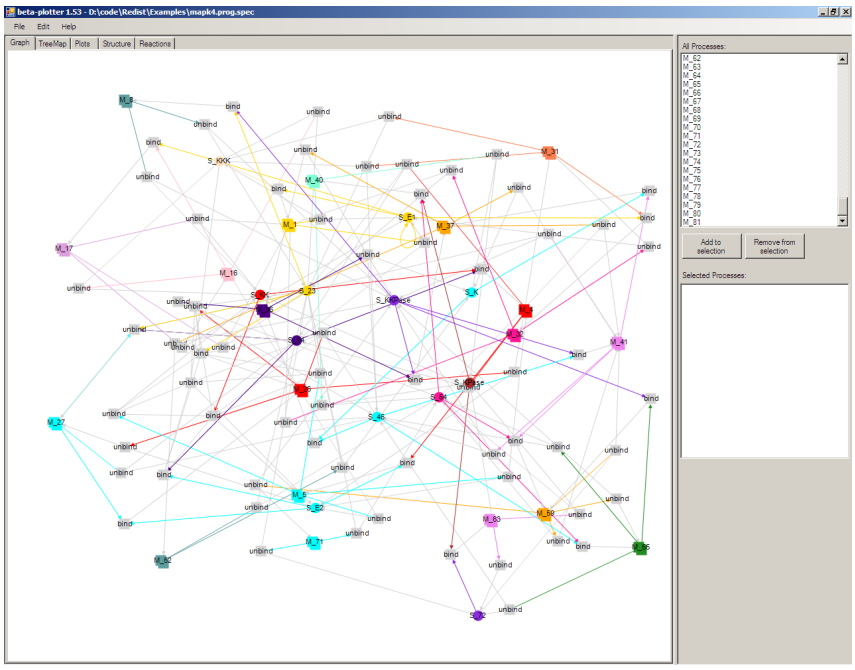


Figure 5.13: The MapK reaction network displayed as a Graph.

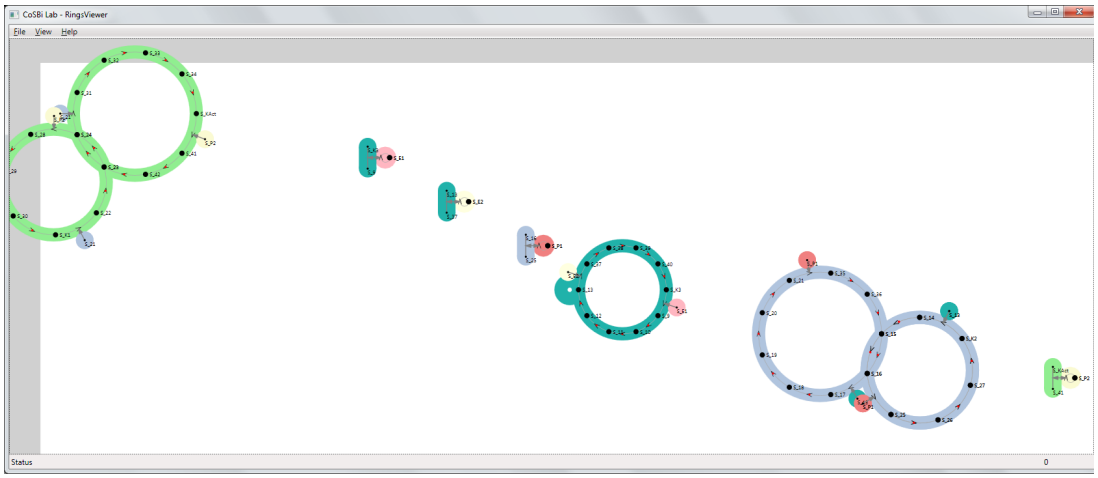


Figure 5.14: The MapK reaction network displayed in Rings.

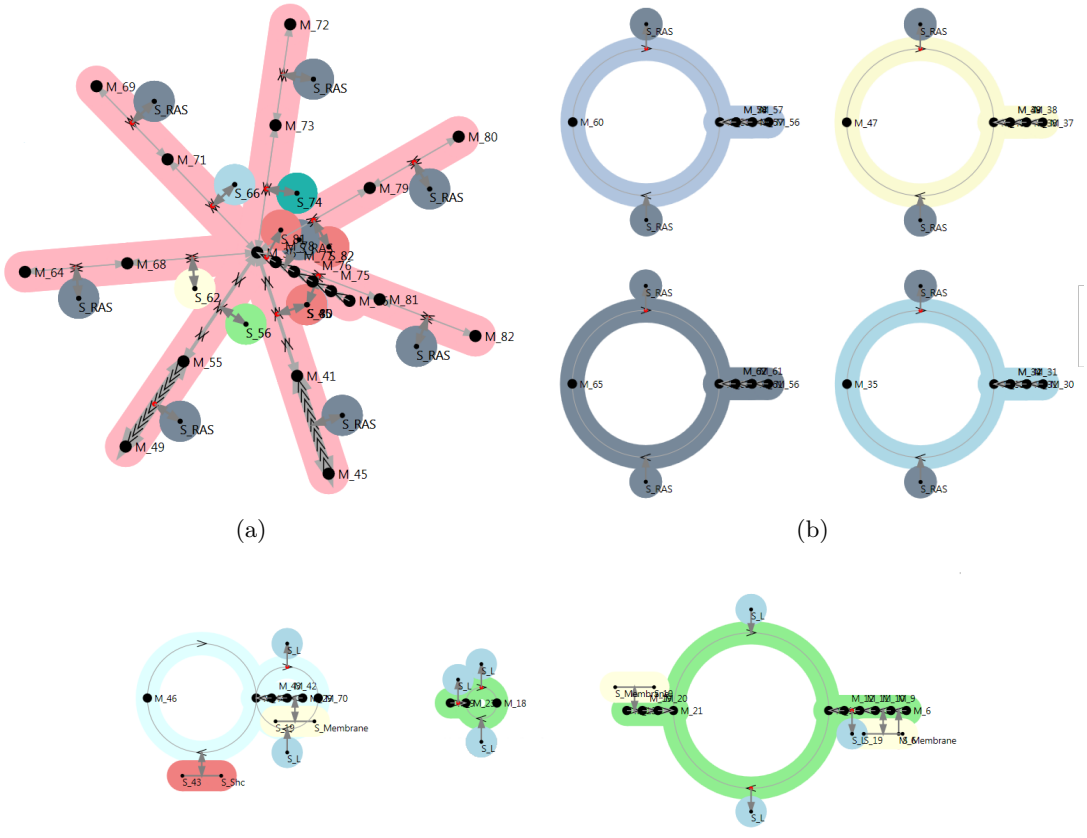


Figure 5.15: The ERK reaction network, obtained from the simulation of the BlenX version of the Fell ERK model. The applications highlights (a) a central reaction "hub", (b) how some entities follow the same behaviour, (c) the interaction of the ligands L with the Membrane

5.3 Complexes

As we have seen in Chapter 3, complexes as native language constructs are a peculiar feature of BlenX, where boxes play the role of monomeric units; complexes can be used to represent any ensemble of two or more boxes, from dimers to more complex polymers. New complexes can be generated at runtime as a consequence of binding and unbinding actions -similarly to what happens when new boxes are created after monomolecular and bimolecular reactions-, and so it is possible to easily generate complex biopolymers made of dozens or hundreds of boxes from relatively simple and compact programs.

The drawback is that it is not easy to understand how a complex is shaped and generated; programs can generate thousands of them as part of a simulation or analysis; for example, a simulation produces a trace, a textual description of the time evolution of the program, that includes a description of complex structure. This textual output is complete, but can be difficult to understand because of the overwhelming quantity of data.

5.3.1 Need for classification

The task of visualizing a complex consists of rendering each box in the complex at a position, with a dimension and a colour that depends on the position of its neighbours and on its *class*.

Note however that in the Chapter 3 we have described what an entity is, but not what a *class* of entities is. We have already given a similar definition, when we introduced *species*: in the case of simulation using species-based methods, like the Gillespie SSA, a definition of species is necessary. As we have seen, we use classes of structural congruence for that purpose; however, here we want to tackle the problem in a more general way.

We can define a class of entities as the minimal set containing all the entities that are *equivalent* one to the other. Hoverer, introducing a notion of equivalence does not solve our problem, as it only moves our question from "what is a classes of entities?" to "when two entities are equivalent?"

There are various definition of equivalence, with increasing expressive power: from syntactical to structural to behavioural. Clearly, the latter would be the best one to represent a biological species: two biological entities that act in the exact same way should be considered equivalent, and therefore as belonging to the same species. However, due to computational constraints, in BlenX we group entities in the same species up to structural equivalence [183].

Moreover, talking about behaviour raises a whole new set of questions. Behaviour is a very general term, for which many different definitions are given in literature. In

computer science, especially in the process algebra field, behaviour is usually identified with the notion of *bisimulation*. However, for biological purposes, this definition is way too strict, and yet in some cases fails to capture different behaviours (as in the case of different rates, see [30]).

The definition of behaviour might even change, as it depends on the scientist goal: for different models, for different analyses or questions, it is perfectly valid to assert that two entities are equivalent if they have the same response to external stimuli, and/or they have the same interaction capabilities, and/or they have the same internal state and/or the same processing capability (even when their interaction capabilities are differnt) and so on.

Finally, for information extraction and processing from output data (e.g. simulation results) even a notion of equivalence based on behaviour may be too strict. For example, you might want to divide and represent differently some entities based on their structure.

For all these motivations, we introduce in our framework a *classification* module between *output data* and *visualization*. Thanks to this classification module, a user can decide exactly when two entities are equivalent by classifying them into categories.

Classification could be very powerful and serve other purposes than visualization; in this first iteration however we want to concentrate on the set of primitive necessary to our visualization goals.

5.3.2 Our approach

Our goal is to extract from the output of a processed BlenX model data in the form of graphs, and present them to the scientist in a form that helps him/her to understand the dynamics of the system.

When we deal with a big graph and we want to lay out its nodes in order to get an intuitive graphical representation of it, most of us instinctively follow a standard procedure. First of all we look for a node with particular characteristics, often a node with peculiarities that make it unique inside the graph. Once we have identified this node we take it as starting point and we proceed analysing its neighbours. Considering their characteristics we *classify* them in some way and we decide how to displace them. We explore the whole graph, iterating this process and obtaining a layout and overall picture that corresponds to what the graph actually represents in our mind.

This process is based on the fact that, even if we do not know a-priori the whole structure of the graph, we know which are its basic blocks and how they can be combined together. These are the observations that inspired our approach, that combines classification of nodes with local layout rules. Following this approach, we can draw in a dynamic way a graph on a node-per-node basis, taking into consideration only the current position

and direction, how it is classified, and its neighbours.

In the concrete application we chose as a case study, the graph (a BlenX complex) is already completely built; in general, our approach may be applied to dynamically created graphs.

5.3.3 Implementation

The process for displacing the nodes can be divided into separate steps, implemented by four blocks. The first block gives to the user an instrument to classify nodes depending on their characteristics, by means of a simple language for defining classes of nodes. The second block explores the input graph, labelling each node with one or more of the previously defined classes. The third block is used to define the visualization rules for each class of nodes. Using a simple set of instructions the user can specify how to draw a node belonging to a class and how to layout other nodes around it. The last block visualizes the input graph using as input the labelled graph and the user defined rules coming from the third part. The implementation schema is represented in Fig 5.16.

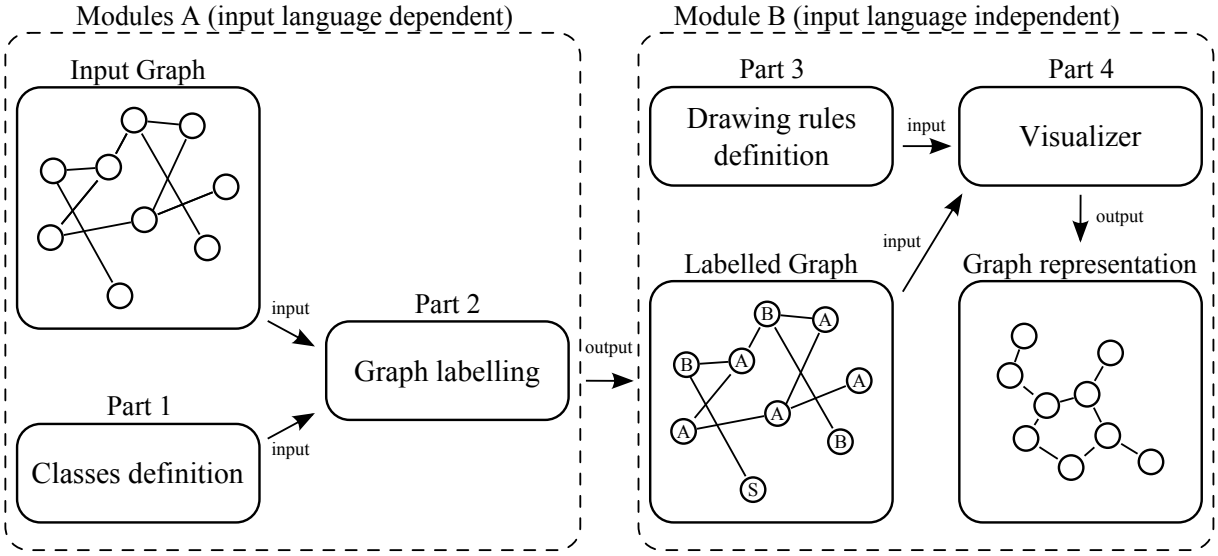


Figure 5.16: Organization of the process for disposing graph nodes

As shown in the picture it is possible to separate the implementation in two modules. The first one (A) is dependent from the language used for describing the input graph. The second one (B) instead is independent from such language as it depends only on the output of module A, which act as a proxy or facade for the graph.

This observation implies that module B can be used for any language, while module A has to be rewritten in order to act as a proxy that transform the considered language to

a labelled graph in the format consumed by B. In section 5.3.3 we will describe each part of the implementation separately. For the language dependent blocks, we use the BlenX language. Here we propose a short introduction to this language focusing on the aspects important in this context.

Classes definition

Each BlenX box has a set of binders, or interfaces; as a first approximation, the set of binders defines the interaction capabilities of that box[2]. Moreover, when a box changes its state it typically changes also its binders and thus the characteristics of some of its interfaces. As a result, in many cases it is enough to consider the binders in order to understand the kind of box we are dealing with and which state this box is in, and therefore to classify it.

For visualization purposes, we will classify BlenX boxes following this approach: using information on their interfaces only, avoiding to consider the structure of their internal program. Following this strategy the definition of a *class* corresponds to the specification of a set of interface descriptions. If a box has interfaces for satisfying all the interface descriptions of a class, it belongs (not exclusively) to such class. We provide a language for specifying the classes and here the generating BNF grammar follows:

[2]This is an approximation because a binder may be never used as a communication channel by the internal process, or it may be used to perform input in a box and output in another box. Clearly, these three kind of boxes give rise to different possible interactions

classList ::=
  class
| *classList class*

class ::= (*interfaceDescSet*) *Id*;

interfaceDescList ::=
  interfaceDesc
| *interfaceDesc*, *interfaceDescSet*
| *interfaceDescSet* ...

interfaceDesc ::=
  listOfType:*listOfState*
| *

listOfType ::=
| *type*
| *listOfType* **or** *type*

listOfState ::=
| *state*
| *listOfState* **or** *state*

type ::=
  Id
| *

state ::=
  hidden
| **bound**
| **free**
| *

According with the rules introduced by this grammar, the following is an example of rules describing two classes of boxes:

```
(A:free, B or c:*, *:hidden, *) classA;
(*:bound or hidden, B:* ...) classB;
```

The first row describes the rules to match a class with name `classA`. Boxes that belong to this class have exactly four interfaces; one of the interfaces has type `A` and is free, one has type `B` or `C` and can be in any state, another has to be `hidden` and can be of any

type, the last can be of any type in any state. The second class, called `classB`, does not specify an exact number of interfaces; the dots at the end of the interface description set mean that there can be more interfaces than those specified. Of course, the additional interfaces can be of any type in any state. The boxes that belong to this class must have at least two interfaces, one bound or hidden and the other on of type B.

Blocks description

Graph loading and labelling

We have seen that, in order to guarantee flexibility and extensibility, in our framework the input graph is loaded by a *graph adapter*. This block is conceptually very simple: it produces an in-memory representation of the graph that is shared and used by the other module. For this particular application, we wrote a BlenX complex graph adapter, which reads each complex generated from a BlenX model as a different graph.

The BlenX graph labelling block explores the BlenX output file and associates to each box that belongs to a complex a list of classes. By combining the in-memory graph and the list of labels we create the labelled graph that is used by module B.

The labelling procedure starts with a two steps pre-processing phase, in which rules are expanded and sorted, so that the algorithm can treat them in consistent way that speeds up the association of rules with binders:

- Rules explosion: classes described with rules containing one or more **or** are expanded into separate class description with basic rules. Example: `(bound or hidden:*, *:B ...) classB;` is transformed into a set of two rules with the same class name: `(bound:*, *:B ...) classB;` and `(hidden:*, *:B ...) classB;`

- Rules sorting: we sort the basic rules inside each class description in order to create a list for each class where the most restricting rules are at the beginning; therefore, rules that specify both type and state (*type*:*state*) are positioned at the head of the list, followed by those specifying only the type(*type*:*), than those specifying only the state *:*state* and at the end those specifying no restriction (* or *:*).

The algorithm then tries to match each node in the graph, in this case each box in the complex, with all the class descriptions. If a class matches, the name of that class is added to the list of labels associated with the analysed box. In particular, the matching phase of a class description with a box proceeds as follows:

- If the class is of the form (*interfaceDescSet* ...) *Id*, and the rules in the class description are less then the number of binder of the current box, we create a copy

of the class description where we substitute the ellipses with a correct number of unrestricted rules (*). This allows us to handle all the classes in the same way.

- If the number of class rules and the number of binders are not equal, the box does not belong to the current class.

- Each binder is associated to the list of all the rules it satisfies. If one of the binder dose not satisfy any rule the box does not belong to the current class.

- A different rule is associated with each binder, choosing among one of the rules satisfied by the binder. If this association is possible, the box belongs to the current class.

5.3.4 Graph layout

Along with the classification and labelling block, the layout part is a fundamental block for our framework. This block, designed to arrange nodes in the two or three dimensional space, is independent from the graph description and from the classification grammar, as it only depends on the normalized, labelled graph generated by the previous block.

The layout is not based on an automatic algorithm; general layout algorithms are very good for arranging nodes without human intervention, most of the time in a visually pleasant way. Some of them even emphasize some graph property, making it possible to understand some properties just by visualizing the graph. However, the generality of these algorithms have also a drawback: nodes disposition is based on some topological properties, not on the domain-specific features of the graph. This is impossible for a general algorithm that need to work on every graph type.

A possible alternative is to write algorithms for the layout and visualization of a particular kind of graphs; say, reaction networks graphs. In this case, the algorithm knows which kind of nodes, structures, connections to expect, and so arrange graphical elements on the screen making sensible and knowledgeable choices. Obviously, this knowledge has to be fixed inside the algorithm, encoded by the developer that wrote it. The drawback here is the opposite than in general layout algorithms: it lacks flexibility, and therefore it can be used only for the kind of networks it was designed for.

For our framework, we wanted to get the best of both worlds: an algorithm in which the end-user can exploit his (or her) domain knowledge to build a meaningful representation of the studied object, without being particularly tied to a specific one. This approach is not convenient as a general layout algorithm, because it needs user attention and interaction to specify how the layout should proceed, and is not as powerful as a specialized algorithm,

as we don't offer a full-fledged programming language to specify it. However, it have been proven to be a good compromise and to fulfil perfectly our current needs.

The user can insert his (or her) domain knowledge inside the algorithm by specifying how a single node should be displaced with respect to his predecessor, how its neighbours should be arranged, or even a mix of the two. Besides position, each node can be assigned a colour, a shape, a label, and many other properties. In particular, in our case, we use classes or combinations of classes as produced by the labelling block as the starting point to assign properties and rules for positioning.

Let's explain it better with an example. Suppose that the previous block classified the nodes of a graph in just three categories: `left`, `right` and `root`. From the names, is easy to understand that this graph should be laid down as a tree: starting from the root, the node connected to the root and marked as *right* (if any) should be placed 30 degrees to the right, while the one marked *left* (if any) should be placed 30 degrees to the left. Repeat the same process for the nodes attached to them. In our framework, this can be expressed using a *layout grammar*:

```
rootnode root {
   successor_adjustment = left:(translation(1,2,0)); right:(translation(-1,2,0));
   shape = cylinder;
   color = blue;
}

node left or right {
   successor_adjustment = left:(translation(1,2,0)); right:(translation(-1,2,0));
   shape = cylinder;
   color = red;
}
```

This example does exactly what we specified using words, in a grammar that our algorithm can understand. The complete grammar, shown in Table 5.2, is composed by a set of *drawing rules*, one for each node class or set of classes. Each drawing rules can specify a set of *attributes*. Attributes are properties to be applied to nodes belonging to that class(es), including properties used to transform the current node location, and the location passed to the neighbour nodes.

The layout algorithm uses these drawing rules (properties and transformations) to lay-out the labelled graph. The algorithm starts compiling a list of nodes whose class is marked as *rootnode* in the drawing rule. In our example, only nodes labelled with `root` will be put in that list. The algorithm takes one node from the list as a starting point for a visit of the graph. During the visit, the algorithm keeps track of the *current position*; when it finds a new node, it marks it as laid out using the current position, then applies

nodeList	::=	*node*
	|	*nodeList node*
node	::=	**root_node** *bodyNode*
	|	**node** *bodyNode*
bodyNode	::=	*IdCond***{** *attribute_list* **}**
attibuteList	::=	*attribute*
	|	*attibuteList*;*attribute*
attribute	::=	**shape** = *shapeVal*
	|	**color** = *colorVal*
	|	**size** = *dimension*
	|	**successors_direction_adj** = *IdCond***:(***adjastmenList***)**
	|	**transformation** = *adjastment*
dimenson	::=	***real***
	|	***real*, *real***
adjastmentList	::=	*adjastment*
	|	*adjastmentList*, *adjastment*
adjastment	::=	*rotation*(*real*, *real*, *real*, *real*)
	|	*translation*(*real*, *real*, *real*)
IdCond	::=	*IdCond* **or** *Cond*
	|	*IdCond* **and** *Cond*

Table 5.2: The layout grammar accepted by the graph layout block

to its attributes the specified properties (colour, etc..) and the transformations to the current position. Position transformations can apply to the current node or, like in the case of our example, can be applied to the current position passed to neighbours using the `successor_adjustment` attribute.

At the end of the process, the graph layout produces a mapping from nodes to points in space, that will be used by the last block in the tool-chain to display the results.

Visualizer

The last block in the tool-chain is a GUI application. Its task is really simple: it takes the mapping produced by the previous block and uses it to render the graph on the screen.

In order to understand the mechanism used to draw a complex think about the visualizer as a program that transforms a 3D Cartesian coordinate system. The visualizer uses this coordinate system to position the objects and as a base for the translations and rotations used to update the current drawing position. Every object drawn by the visualizer is centred on the current coordinate system origin.

The visualizer gets the information on how to draw a complex analysing the rule defined in the layout grammar; in particular, at each step it looks for a rule with a label compatible with the one marking the current node. The rule has some attributes that specify the visual cues of the node (i.e. its shape, size and colour). Using these attributes the visualizer draws the current node inside the 3D environment, then it updates the coordinate system and draws the neighbour nodes (if not already drawn). The instructions for updating the coordinate system are specified by the `successors_node_adj` attribute. This attribute holds a list of labelled transformations, specified as a composition of translations and rotations. The visualizer considers one of the successor nodes, reads its label and searches the `successors_node_adj` list for an entry with a label matching the one on the successor node. If an entry is found, the visualizer updates the coordinate system using the transformations on that entry. Then, in order to obtain the attributes needed to draw the node, the visualizer looks for a rule with a label matching the one on current node; finally it draws the node at the updated position using the correct set of attributes and re-iterate the process on the neighbours, making a marked traversal of the graph with a DFS or BFS algorithm.

The visualizer starts the visit of the graph from a *root node*. A root node belongs to a class whose matching drawing rule is marked as `root_node`. It is possible to have more than one root node per graph; in this case the visualizer starts randomly from one of them.

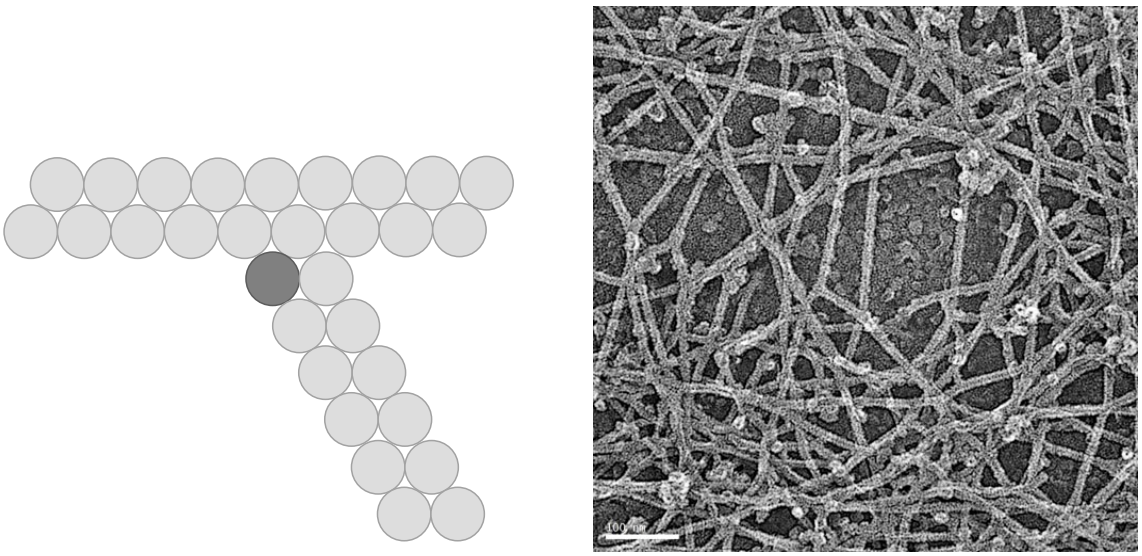
Note that as every other block in the chain the visualizer is completely replaceable. Indeed Larcher in [53, 134] replaced the visualizer presented in this thesis, which uses rich

graphics to display nodes but does not allow for transformations in the 3D plane, with a more complex 3D visualizer with stereographic display.

Examples

Larcher and Romanel [136] developed a model for actin complexation and growth. Actin monomers, when bound to certain molecules, can form filaments and tree-like structures that are important for processes like cell locomotion, phagocytosis and intracellular motility of lipid vesicles.

In the polymerization process actin monomers arrange themselves into two parallel, twisted strands that form a coiled structure. The exact 3D structure is difficult to reproduce on paper, and therefore in textbooks it is often simplified and represented as in Figure 5.17.



(a) (b)

Figure 5.17: (a) Bi-dimensional representation of an actin filament, (b) Microscope image of an actin filament (Copyright Dylan Burnette, NIH)

The complexes produced during the simulation of Larcher and Romanel model should have the same topology; however, as we already pointed out, inspecting the simulation results to understand how a complex is structured is not a trivial task.

Our goal is to produce actin representations similar to those depicted in Figure 5.17 starting from the model output. The first step is to define a set of classes for the classification of BlenX boxes:

```
(LB:free,*,*) first_monomer;
(L or LB:bound, *, *) monomer;
(*,*) arp;
```

This file defines three classes that can be used to distinguish boxes for the root of the filament, the actin monomers (with the exception of the root node), and for Arp2/3 (the branching points).

We define a set of layout rules for these classes:

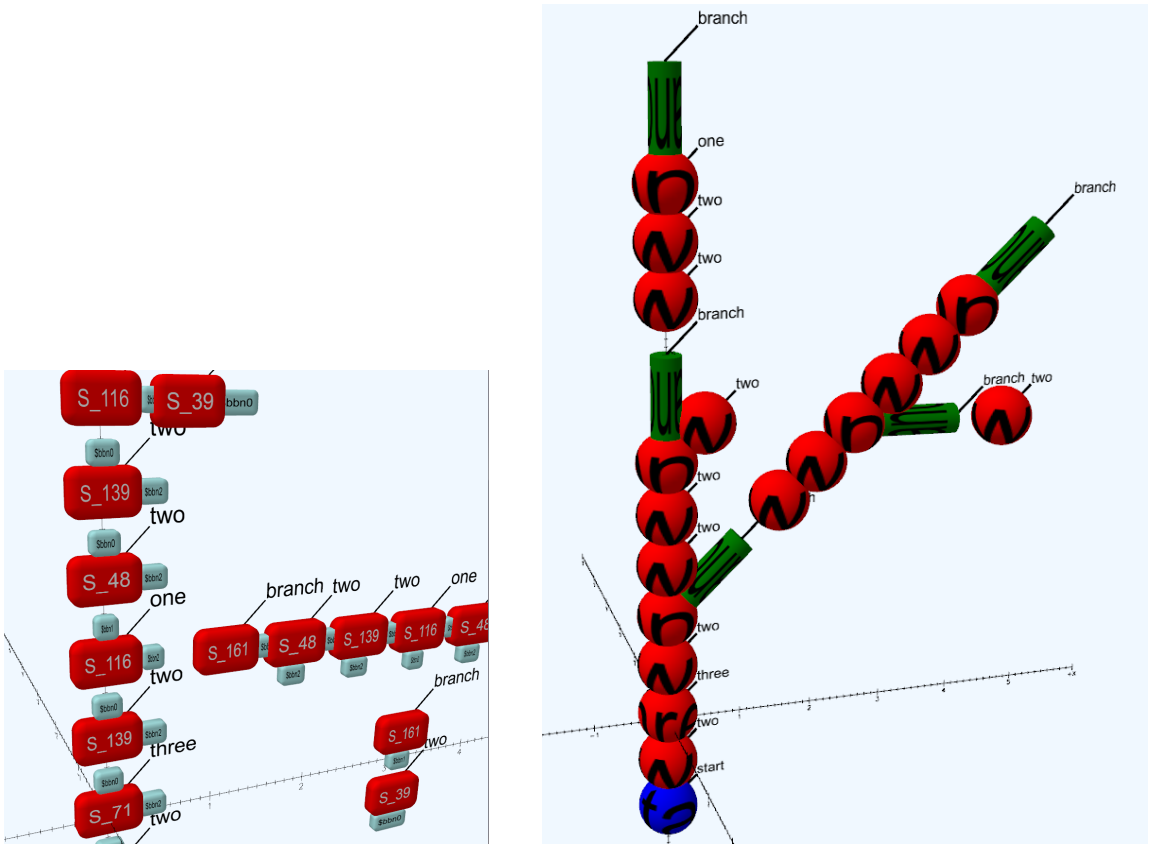
```
root_node first_monomer
{
  shape = sphere;
  size = 0.5;
  color = blue;
  successors_direction_adj = monomer:(translation(1,0,0)),
                             arp:(rotation(60,0,0,1),
                                  translation(1,0,0));
}

node monomer
{
  shape = sphere;
  size = 0.5;
  color = blue;
  successors_direction_adj = monomer:(translation(1,0,0)),
                             arp:(rotation(60,0,0,1),
                                  translation(1,0,0));
}

node arp
{
  shape = sphere;
  size = 0.5;
  color = orange;
  successors_direction_adj = monomer:(translation(1,0,0));
}
```

The visualizer uses the algorithm presented in Section 5.3.4 and this set of rules to draw BlenX complexes representing actin filaments. The result is presented in Figure 5.18

5.4 Related work and Future directions

Scientific visualization is a very broad area of research: even if we restrict ourselves to visualization of biochemical data, there are too many approaches to give a meaningful summary of the related work. We would concentrate only on some interesting tools and approaches from each of the areas we touched (visualization of space, networks, complexes).

One of the most active research groups on visualization of spatial simulations and volumetric data is the Volume Rendering group at the VRVis Zentrum fur Virtual Reality



(a) (b)

Figure 5.18: The actin filament visualized as Boxes and as geometrical shapes (spheres and cylinders)

und Visualisierung Forschungs. They work mainly on visualization of medical imaging, surgery simulation and volumes created by electron microscopy and confocal microscopy. These can be considered as related fields.

Their *High-Quality Real-Time Volume Rendering* project ([203, 96]) is focused on the study of algorithms and methods that combine volume rendering, isosurface extractions, and illumination techniques for advanced visualization of relevant characteristics. Many of the developed techniques and concepts can be transferred to our domain.

A related area of research involves the visualization of Flows [133], Tensor Fields [229] and Vector Fields, using techniques like pathlines, streamlines, Line Integral Convolution (LIC) [23] or a mix of LIC and Volume Rendering [102]. These techniques can be used to visualize not only the instant concentration of species in space, but also to directly visualize the flux, the movement and evolution of the diffusion gradient over time. In OpenCOVER, we already tried to use pathlines for this task, with mixed results. LIC, streamlines or streamshapes as used in the visualization of MT-RMI could lead to better

results.

With respect of network visualization, to the best of our knowledge, no tool or visualization technique has been studied specifically to visualize the reaction networks produced by systems biology simulations. On the other hand, the literature on visualization of biochemical pathways, a closely related field, is quite rich. Many software products, like Osprey [21], Biolayout [223] and Bio-Path [19], facilitate visualization and manipulation of complex interaction networks in large-scale datasets and biochemical pathways. Most of the tools display networks using both default layouts and ad-hoc algorithms. Other works concentrated on this aspect, developing context-aware layouts for biochemical pathways [12, 17, 145].

Finally, many software for the visualization of molecules in 3D exists. Just to cite the most famous, RasMol, JMol, Chemscape, MAGE, MolView and Chem3D are all very well know and established programs in the Molecular Biologist toolbox. Their purpose is to visualize the 3D structure of proteins and molecules. The novelty of our approach is to apply this kind of visualization to the rendering of *process structure*, in particular to the rendering of complexes, a unique feature of BlenX.

The small language we use to define the positioning of boxes in the 3D space, based on their class, is very similar to the ThreePi language developed by Cardelli et al. [33]. This similarity comes to no surprise, as the underlying concepts are the same: attach to each species (pi-calculus process or class of boxes) a set of affine transforms, in order to assign to each entity (i.e. to each instance of that species) a position in the 3D space.

Future research directions could proceed along two main lines: on the one hand, we should consolidate and make more usable our existing work; on the other hand, we could introduce new methods in our visualization prototypes – like the Tesor and Vector field visualization techniques we mentioned – or extend our existing work.

The first direction, for example, requires the implementation of a new, stand-alone Volume Rendering application for integration with BetaWB. In fact, one drawback of using COVISE for visualization is that it is necessary to take a dependency on the whole framework in order to use it: in order to develop COVISE modules and to run COVISE applications, it is necessary to set-up and deploy the whole framework. Besides, COVISE is a flexible but complex piece of software, that requires a very powerful hardware in order to fully express its capabilities. Furthermore COVISE, as the name implies, is a framework for collaborative visualization and simulation; we only made use of a limited set of its capabilities, limiting our research to the visualization part.

Therefore, while it was easy to use and experiment with it during my internship at HLRS, COVISE revealed to be un-practical to be used inside a small research group like the one at CoSBi. In order to further develop spatial visualization techniques, we plan to

re-produce the basic capabilities we need using smaller, simpler software.

That said, working with such a complex software was a valuable experience: visualization tools for Systems Biology are still in their infancy, and they can take great inspiration from related fields (mechanical simulations, medical imaging, and so on). Furthermore, as Systems Biology scales up and becomes more and more able to deal with complex systems, more powerful collaborative visualization capabilities will be needed.

We are also considering the extension of our work on complex visualization. The drawing process illustrated in this chapter draws a complex considering exclusively its structural characteristics, ignoring temporal information. Improving our method in order to exploit such information should make possible to enrich the representation of the rendered molecule and generate a movie that illustrates its formation process.

The classification of boxes can also be improved adding the possibility to classify them considering characteristics of their internal program. It is useful when modifications of the internal program do not cause any changes of the interface structures but are interesting to be recognized in the drawing procedure.

The classification method can be also used for other purposes: suggestions in this direction come from the Actin example by Larcher et al. [136]. Analysing simulation results obtained from the Actin model is not trivial: filaments can potentially assume infinite conformations, thus even a short simulation usually generates thousands of different complexes. Classification can be used to group together molecules that belong to distinct species but share common characteristics, in order to make the analysis and inspection of simulations results easier.

5.5 Summary

In this chapter we introduced visualization as a discipline for understanding the outcome of Systems Biology experiments. In particular, we focused on three different visualization techniques for simulations. Spatial visualization, through volume rendering and isosurface extraction, is key to understand spatial interactions in large reaction-diffusion simulation. Our approach to network visualization, where we extracted and rendered in an intuitive way network components, allowed us to show key information in a simpler and tidy way, even when working with possibly huge networks. Finally, visualization of complexes made simple to interpret and understand one of the key features for modular composition of BlenX models.

Chapter 6

Tools and Composition

Systems Biology is a field of research that heavily relies on software to carry on its studies. As we have seen in Chapter 2 and 3, in the last years this dependence grew even stronger: computer science theory deeply influenced research in Systems Biology, creating a whole new research branch known as *executable* systems biology. In this field of research creation of models and their analysis and simulation using computer programs is not only an useful instrument, but plays a fundamental central role.

In the last chapters we have seen how computer programs can help research in systems biology; in particular, we have seen how software is fundamental to create, simulate, and visualize biochemical models. Even if it is a common understanding that model simulation plays a central role in Systems Biology, the computational tools used by researchers in this field are by no means limited to simulators. In fact, many other programs and tools are needed, to pre-process and prepare input data, to analyse and visualize output data, to run statistical analyses, derive aggregate results and drive and control how data is processed and used in simulations. Different programs and tools need to be used together to create an *in-silico experiment*.

As we already discussed thoroughly in Chapter 1 and 2, experiments are central to the scientific method: hypotheses formed by scientists to answer a question must be tested before drawing any conclusions. A hypothesis is tested by means of an experiment in order to be confirmed, rejected, or refined, under the currently available knowledge. The composition of different software tools in a consistent way, in order to build an in-silico experiment, is therefore one of the central aspects in the day-by-day work of a systems biology researcher, probably second only to the construction and composition of models.

In our research group we experienced these difficulties quite often. Even if every member of our group have a good working knowledge of informatics, not everyone is familiar with programming. This is even more evident with people with a background in life sciences: the area of expertise of biologists does include the usage of informatics

instruments like simulators or on-line databases, but does not include the ability of dealing with difficult computer tasks (e.g. using command line scripts, building a database query, and so on).

On the other hand biologists and, more in general, natural scientists, are familiar with the concept of *protocol*. A *protocol* is a procedural method for the implementation of *experiments*. Protocols are present to standardize laboratory methods and ensure replication of results by others in the same laboratory or by other laboratories; they usually consists of detailed procedures, lists of required tools, how data is handled and treated and rules for generation of results and reports.

Up to now, there was a lack of a coherent framework for in-silico experiments; a single tool was commonly used to answer questions for one class of in-silico experiments; new research directions needed for a new tool in order to derive new results. Even inside our research group, most of the activities are implemented today by different tools written in disparate programming languages; the user has to use them in the correct order (enforcing the protocol *by hand*) and transfer data between them manually (i.e. copying and loading files or gathering data from databases). Despite some efforts from the community, tools lacked interoperability and composability, and generalizations was overlooked.

Well defined interfaces between tools are needed to enable composability and interoperability, so that a library of abstracted functionalities could be built. Then, users can consider these interoperable tools as basic *building blocks* and use them to compose a *protocol* to design and implement in-silico experiments.

6.1 Tools as Services

The list of required tools and the detailed procedure to follow in order to complete an experiment can be described as a series of steps; typically, these steps are described either through text or visually in a graphical way, using block diagrams, flowcharts or similar diagrams. Looking at these requirements it appears clear that an integrated software environment designed to help scientific research needs:

1. to expose software blocks and tools as components or *services*[1]; here we refer to a service as a software tool that represent a common operation (for example, execute a simulation, gather data from and online DB, and so on) in the scientist's domain;

2. easy composition/chaining of different services, preferably using a visual language.

[1]We choose to call our component services, in contrast with plug-ins or agents: a plug-in extends the capabilities of a larger program, and therefore it implies a strong dependence on a central program; the concept of agent, on the other hand, commonly implies a very high degree of intelligent and autonomous behaviour. Our components, instead, run independently but are orchestrated to obtain a richer behaviour, like servers in a microkernel OS or web-services in the cloud.

Services should be able to perform processes and exchange types of data linked to the domain of interest; in our case, Systems Biology data. Therefore, with the help of users and scientists from our research group, we defined the list of basic building blocks that a framework for Systems Biology experiment composition needs:

Composer services: user-driven model definition, load imported behaviours, load imported data, abstract behaviour, edit behaviours and data, compose submodels, define model variants, define behaviour mutation strategies.

Knowledge Inference services: automated and user assisted computation of rates and behaviours.

Runner services: simulation: chemical interactions, spatial/diffusion; numerical analysis; Markov chain extraction; model checking; static analysis; abstract Interpretation; type checking.

Abstraction services: statistics; analysis of components; Fourier analysis.

Visualization services: plotting; dynamic view; spatial view; structural views: graph of reactions.

Box 5 Ontology terms

Entity:	a biological entity, able to evolve its own state and the state of other entities. Possible evolutions are of finite types
Behaviour:	a set of evolution capabilities of a set of entities
Values:	an instantiation of the speed of evolutions and of the initial population of entities
Model:	a pair (Behaviour,Values)
Question:	a property of the model (or of the behaviour) whose satisfaction is to be quantified (probabilistically) over time
Service:	a processing block that takes as an input a model and a question (all optional) and returns a model and a question (all optional?)
Protocol:	a workflow of activities processing MODELS, under the control of QUESTIONS
Experiment:	the process of an execution of the workflow, from one defined START activity to a defined END activity

6.1.1 Workflow vs. Dataflow model

A *workflow*[2] is a pattern or composition of steps or *activities* designed to achieve processing intents of some sort. Steps are organized into a work process, that can be documented and learned, using some basic operators: sequencing of connected steps and operations, choice between multiple alternative steps, and so on.

The term workflow is also often used in computer programming to capture and develop processes and scenarios that may require human-machine interaction.

The most important programming and runtime frameworks provide libraries for creating, composing and executing workflows. This support can be more or less complete and integrated with the runtime environment. As an example, consider the Windows Workflow Foundation (WF), part of the .NET Framework.

The primary goal of WF is to support business processes; the discrete series of step that describes the activities of the people and software involved in the process is described as workflows. Once this workflow has been defined, an application can be built around that definition to support the business process. WF allows to build workflow-based applications, whether those applications coordinate interactions among software, interactions among people, or both. WF addresses the main challenges that arise when dealing with real-world business processes: for example, some business processes can take hours, days, or weeks to complete; state and resource need to be controlled and maintained, making applications persistable for example. Flexibility, the ability to change a business process on the fly, handling unpredictable behavior, allowing for domain-specific activities and coordination primitives are all requirements considered and fulfilled by WF.

At first sight, workflows in general, and WF in particular, could be a very good choice for our purposes. The Trident Scientific Workflow Workbench, for example, is a software from Microsoft Research built on the Windows Workflow Foundation that can be used to compose, run, and catalog experiments as workflows [210]. Being based on WF, the many strong points of WF (like, for example, fault tolerance, resilience, persistence, scheduling over HPC clusters or cloud computing resources, ...) are inherited automatically (for a more detailed overview of Trident and WF, see Section 6.2).

However, it also inherits the same weaknesses, or better, those characteristics that make it good for large-scale, large-data processing and computations but not so great for fast, short, data intensive computations. In other words, while Trident might be a great choice for some of our scenarios, usually we are dealing with a finer *level of granularity*. Moreover, we think that an approach focused on the flow of data rather then on the

[2]Note that here we are referring to workflows as a software technology to formalize and structure complex and possibly distributed processes, not as a language for modelling a simulation. The usage of workflows as modelling tools, although an interesting topic, is out of the scope of this thesis.

control flow might fit our needs best.

As an example, consider a scenario that occurred in our research group. Forlin et al. needed to build a quantitative model to make predictions on the effect of the removal or addition of some bio-molecules from a synthesis pathway present in fruit plants. The group only had a partial idea of the interaction network, and therefore part of it had to be *inferred* from existing knowledge. The available data included some experimentally measured time series, which indicated the variation in concentration over time of each species. Forlin processed the data, using first some statistical techniques and then some genetic programming techniques to guess the network structure and then refine it, until the network output matched the experimental measures.

In order to build this kind of application, the scientists reused some tools already developed in our group (simulators, time series and data readers) plus some third party tools (mainly packages from the R statistical analysis suite); then Forlin automated the procedure connecting the various pieces with Python, a scripting language.

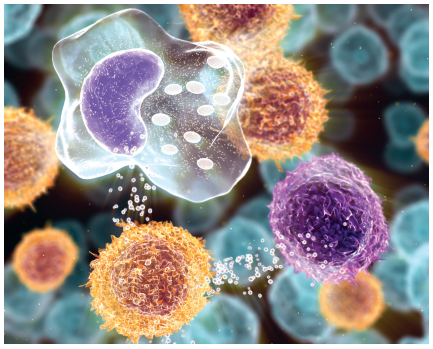
Obviously, not every scientist has the knowledge and time to craft an ad-hoc solution for every different experiment. Moreover, tools and solutions built in this way are usually targeted to the particular case and data under exam: they lack generality, making also hard to reproduce the same steps in a later phase or under slightly different situations. An easier and more dependable way of building and composing systems biology tools is therefore needed. Consider again our requirements: we would like users to consider our interoperable tools as basic *building blocks* and use them to compose a *protocol* to design and implement in-silico experiments. These requirements are very close to the ones needed to build biological models, that we introduced in Chapter 2 and 3.

We can consider software tools as interacting agents with well defined interfaces, like we did for bio-molecules. In this way, we can see how similar the solution to the two problems might be (Figure 6.1). This approach shifts the focus to the interaction capabilities of each agent, and hence towards a data flow model. The flow of data passed from one agent to the other and how this data is processed and transformed by the agents defines the interactions between tools or components, rather then the sequencing of actions and steps (Box 6).

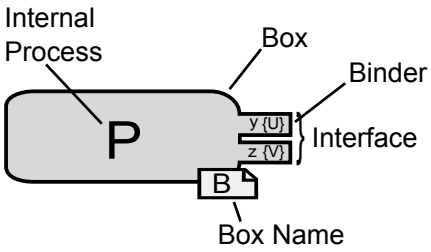
6.1.2 Dataflows for biological experiments

The BlenX language inspired our service model: the notion of typed interfaces, of reactive programming, and the focus on the exchange of data all derived from our knowledge of BlenX. There are four main differences however:

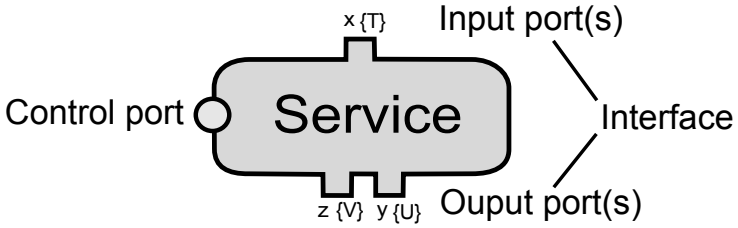
1. For simplicity, we divider the interface in input and output ports. We also added a



(a)



(b)



(c)

Figure 6.1: We represented the cell (a), with its receptors, as a BlenX box in (b), with its interaction sites (*binders*, or collectively *interface*); this idea influenced our definition of services (c) as autonomous computing units. Services, like BlenX boxes, have an interface and an internal process machinery; they react to inputs and produce output data as a result of some interaction.

control or configuration port (the round dot in the Figure); this ports sets up the service during initialization, configuring and controlling it so that it is conform to the question (see Box 5).

2. The internal process needs to be written in a real programming language; the BlenX internal process language is not enough. Although it is desirable that the language or framework for service programming has parallel features, which BlenX supports, we have to remember that services are complex pieces of software (simulators, data analysis toolkits, 3D rendering engines,...) which have to be written using a general purpose programming language.

3. Binder types are replaced by port types, which are the data types of the underlying programming language, or must be easily mapped onto them. The type of data exchanged and processed by services includes, but is not limited to:

   - Primitive data types (strings, integers, double)
   - Time series
   - Graphs

Box 6 Which is the difference between data- and workflows?

We already defined a *workflow* as a discrete series of steps that describes the activities of the people and software involved in a process. In this sense, a workflow is not different from the control flow model typically found in conventional programming: in fact, workflow frameworks like WF were strongly inspired by orchestration languages (like BPEL), which in turn are built upon process algebras. It is possible to recognize in WF many pi-calculus operators: replication, parallel, choice, sequencing.

On the other hand, a *dataflow-based* programming model focuses on the exchange of data, in which the dependencies between data is explicit, and the flow of the operations on that data is determined by these dependencies. Variations of values, for example, automatically triggers the computation of other values linked to it. Dataflow programs are often referred to as *reactive* programs: the program is built as a network of concurrent processes (or, in a simpler case, automata) connected by channels. Processes communicate by sending data over channels, and *react* to input data by triggering a new computation, that will lead to the production of new output messages. Dataflow-based program are like a series of workers on an assembly line, who do their assigned task as the materials arrive.

Both these models are well suited to programming a variety of concurrent or distributed processing scenarios. It is interesting to note that BlenX, introduced in Chapter 3, can easily embrace both programming models. Interaction between boxes, being based on types, message exchange, and interaction capabilities reflect more closely a dataflow-based programming approach; the definition of the internal processes, however, closely resembles a workflow programming model.

The difference between dataflow-based and workflow-based models resembles the difference between functional and imperative programming: it is not sharp and clear, and it is more a matter of style, discipline and focus (for example, it is possible to write functional-style code in an imperative language like C#, and imperative code in a functional language like ML) than of computational power (as we know, all mainstream programming languages are equivalent).

- Models (Behaviour, Data), our main data type

4. The affinity bases interaction model is replaced by an interaction model based on (a) type compatibility and (b) user-defined interactions. Ports can exchange messages only if their type is compatible (e.g. an input port of type double could receive messages from an output port of integers) and only if the user connected them (see Figure 6.2).

Obviously the last point is key to achieve our goals: users need to be able to compose service in a flow that examines, process and stores the data in the way they meant, in order to produce an in-silico experiment.

For example, consider the data flows in Figure 6.3. These composition of services allow the user to create store, run, and persist on permanent storage (for future usage or reference) the procedure to reproduce an experiment, something he previously needed to do by hand.

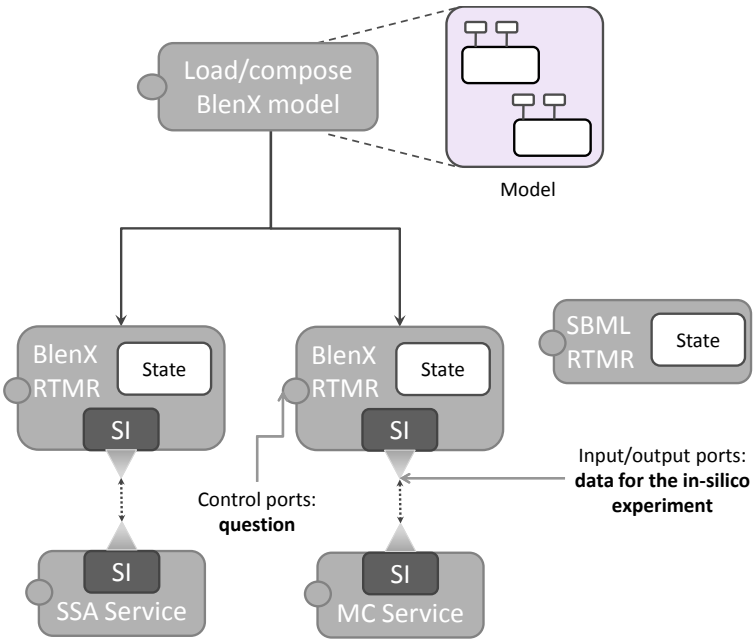


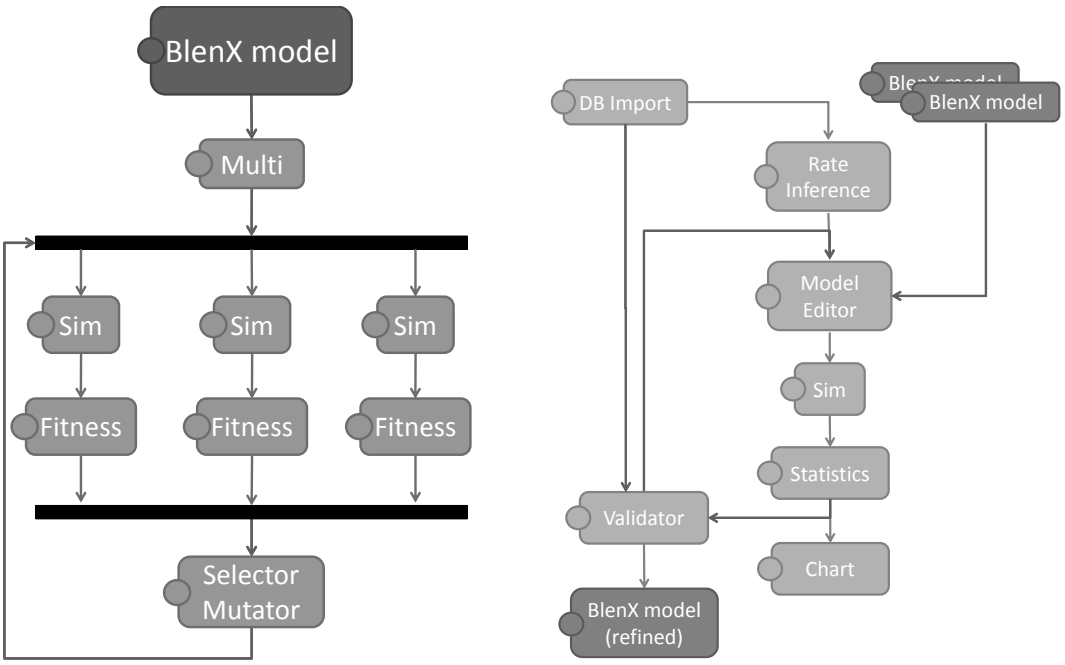
Figure 6.2: Visual composition of services

While workflow frameworks are rather diffused, mainly due to their wide adoption to model and implement business processes, dataflow frameworks are rather uncommon. On the other hand, we wanted to quickly experiment our ideas without the burden of building and testing a complete dataflow framework from scratch, at least not in this research phase in which we did not know if our approach will turn our to be effective. Therefore, we looked if we could borrow some existing technology from other fields of research.

6.1.3 Robotics Studio

Microsoft Robotics Studio is a software platform for the creation of programs to control and run small robots. Robot programs are by their nature reactive: robots reacts to stimuli from the environment to plan and execute their actions. You can even think of a robot as an agent or box: stimuli and message post to its interface are processed, in order to both update the robot's internal state and, possibly, to produce some output or reaction from the Robot.

Robotics studio adopts very interesting and innovative solutions to write this kind of programs. In particular, it is based on VPL (Visual Programming Language) and DSS (Distributed Software Services). VPL and DSS enable the design of applications starting from basic blocks. In this case, the basic blocks (also called activities) are a set of pre-defined data processing primitives and a set of user-definable activities called services.



(a) (b)

Figure 6.3: Data flows for the execution of in-silico (a) evolution experiment (see Chapter 7 for a complete explanation) and (b) model refinement experiment.

Services are implemented using C# and communicate using a lightweight protocol. VPL (Visual Programming Language) is a dataflow based visual orchestration language for activities, services and primitives.

Services are executed within the context of a node, a hosting environment that provides support for services to be created and managed. The DSS service model has been designed to facilitate reuse of services. It does this by making them easy to use and compose with each other while enforcing a very loose coupling between them. All DSS services consist of a common set of components as depicted in Figure 6.7.

It is easy to observe that DSS services are very similar both to BlenX boxes and to our own service concept: the major difference lays in the presence of only one port on which multiple message types are multiplexed. For every message type, it is possible to register one handler; when a message of that kind is received, the correct handler is called automatically. Multiple communication partners over the same port are also allowed: they can all post messages that will trigger the same handler, or alternatively invoke different handlers using a different message type. Notice that the message type is different from the transported type: you can have different message types (for example, PostError and PostResult) that carry the same payload (e.g. a string or an integer).

Internally, message handlers are written using a .NET language. It should come to no

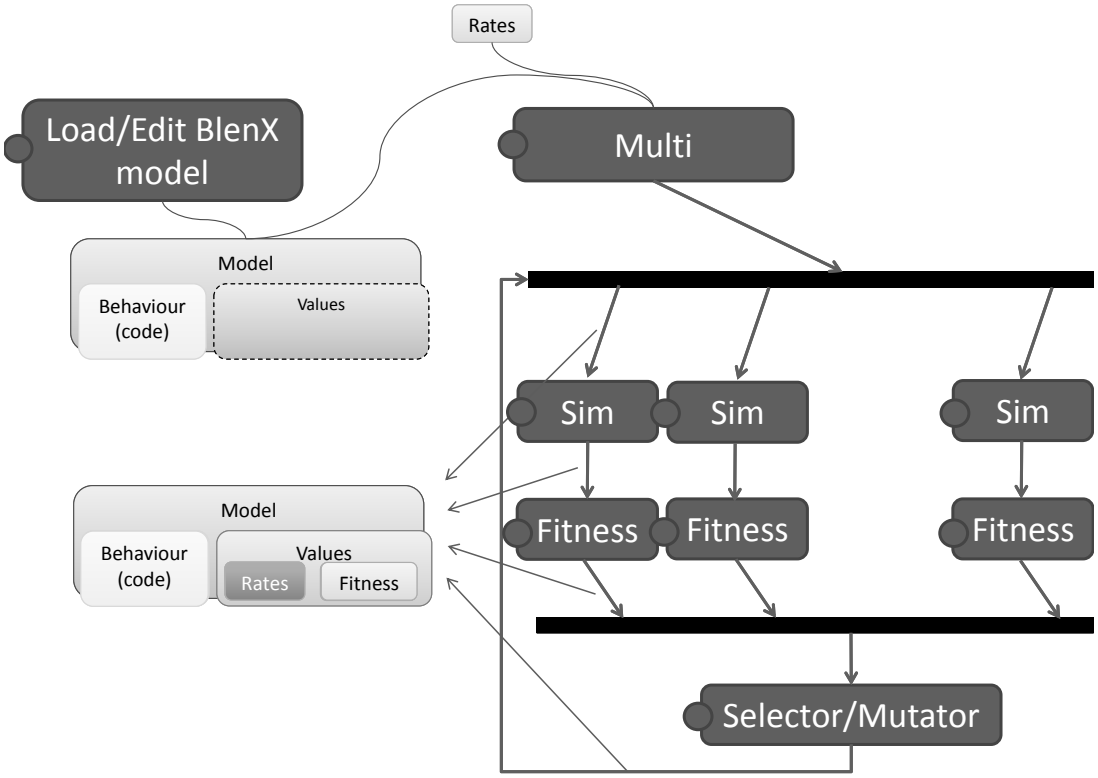


Figure 6.4: Details of the evolution data flow, with the types of data exchanged between services.

surprise that in order to help in writing message handlers and code for service processing, the framework includes a library for concurrent programming, called CCR (Coordination and Concurrency Runtime), a fast library for asynchronous parallel programming based on the join process calculus. Therefore, the analogy between our BlenX-based service model is even deeper than one can initially think. Therefore, we decided to try and implement our own service model on top of DSS, using the CCR to coordinate the flow of data and messages.

This choice proved to be successful: the DSS framework took care of the details (like serialization, object communication and linking, ...). This, together with the notion of contracts and the usage of well defined interfaces, made it easy to exchange complex data structures between services. It was easy to create new services and also to encapsulate existing code in DSS services, as we shall see in the case study in Chapter 7. Scalability proved to be a strong point, thanks to the ability of running services on different machines and making them communicate seamlessly. Performances were also pretty good, with some optimizations when services operated on same machine.

The Visual Programming Language was a good idea, and was very appealing to us: we

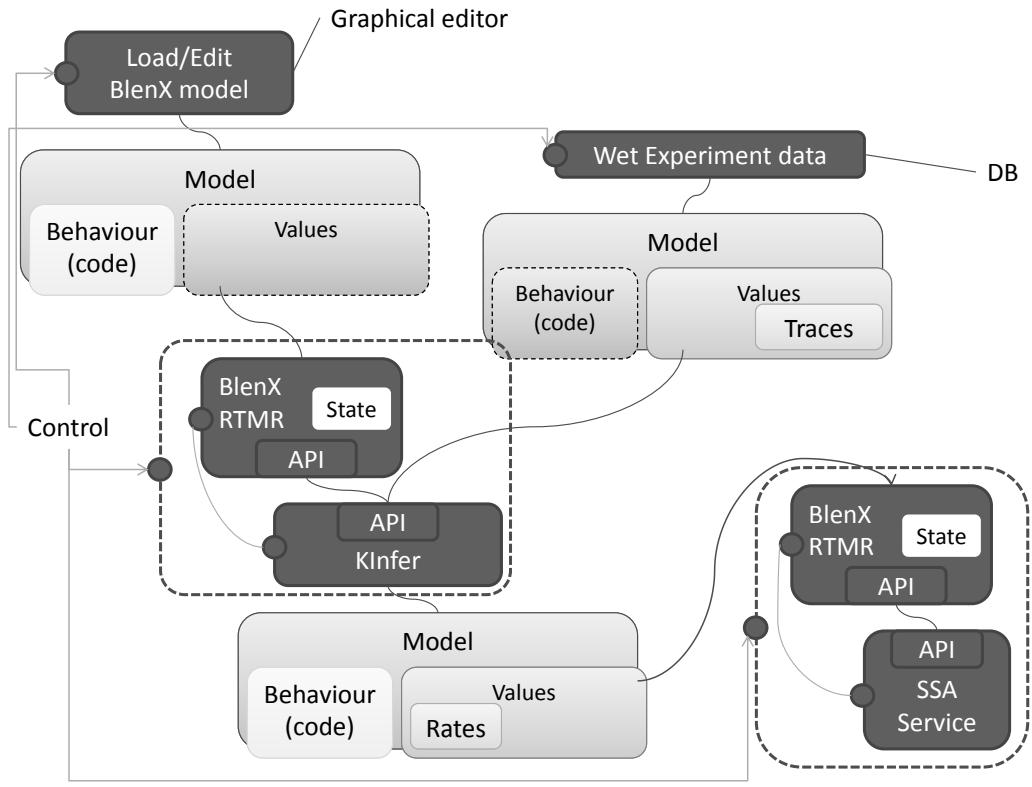


Figure 6.5: A detail of the model refinement data flow, with the types of data exchanged between services.

hoped that our target users could find easy to program interactions between services and activities in this way; unfortunately, it proved to be not flexible and counter-intuitive in many ways, above all in the way of connecting services in a data flow. Other drawbacks were the lack of flexibility in the definition of activities; extensibility is guaranteed only through the design and implementation of new services, which is not ideal in some scenario.

Finally, while DSS and CCR are designed to be extremely efficient and have great performance, they are however too slow to be used in some cases, where the level of granularity we have in mind is very fine (think about a Data Writer service that is invoked at every simulation step, or imagine a visualizer that performs real-time volumetric rendering). Again, lack of flexibility confirms to be the more relevant problem.

An example: Redi

As a simple example of how this framework can be applied to a real problem, consider a stochastic simulation with Redi (see Section 4.3).

Redi has a plugin structure that fits nicely into the DSS model: we encapsulated each

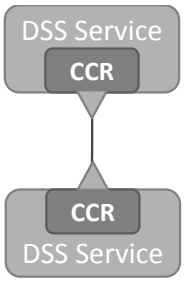


Figure 6.6: The basic DSS service composition model.

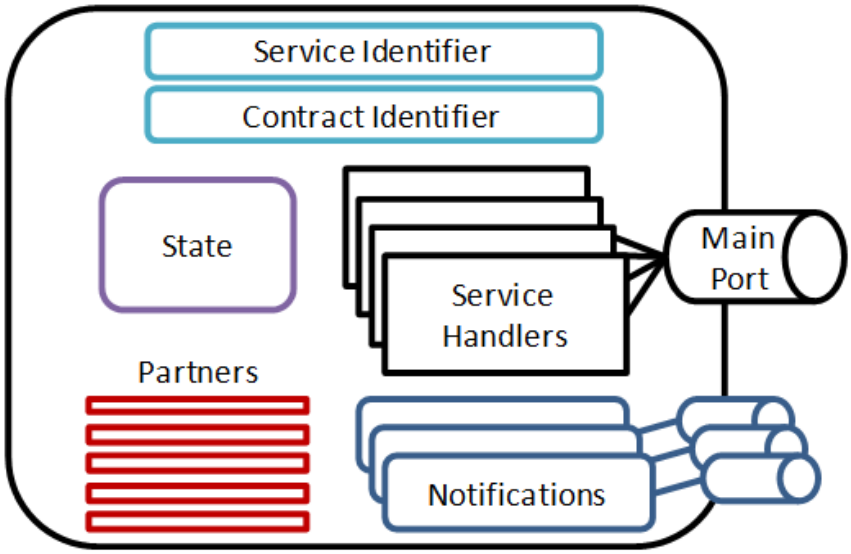


Figure 6.7: A DSS service (from msdn.microsoft.com)

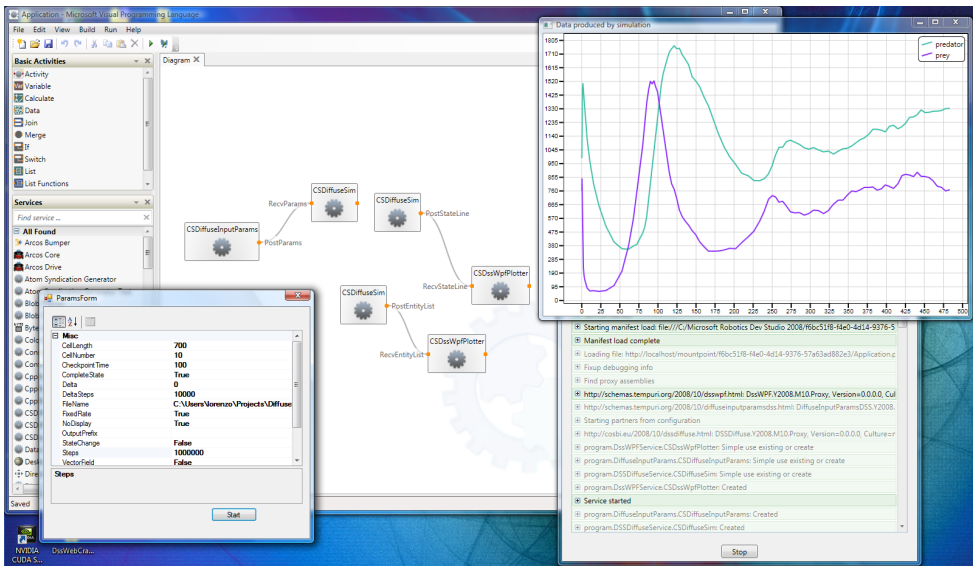


Figure 6.8: Composition and running of Redi as a network of DSS services

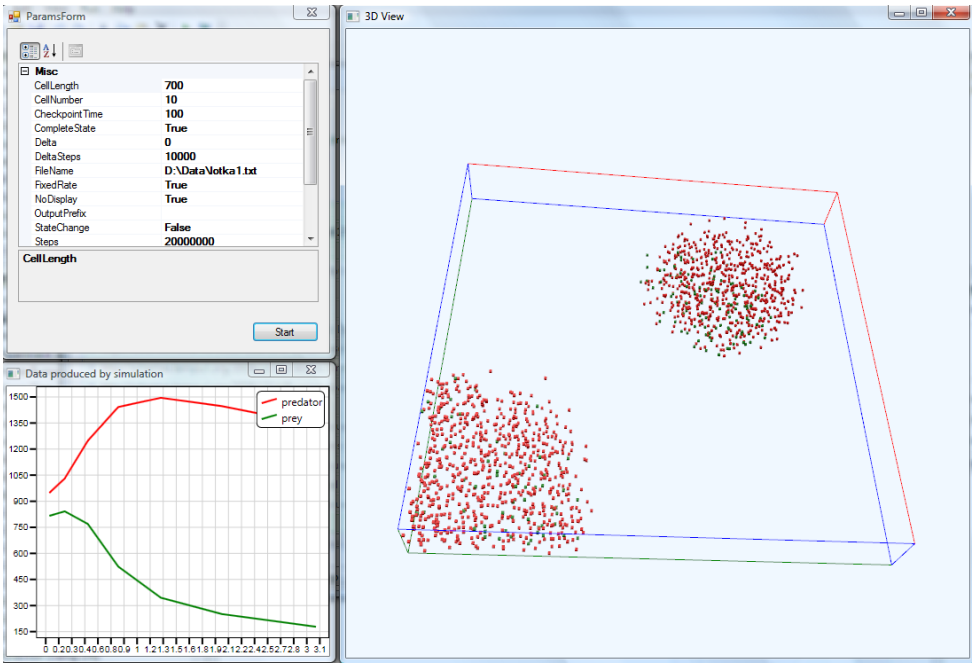


Figure 6.9: Redi and its plugin running as DSS services

plugin in a DSS service. We also build a re-usable DSS services that takes as input a class whose memebers are the simulation parameters, builds a grid view over it and exposes then to the user (Fig 6.8). In this way, we simulate the presence of the "control" port we mentioned in the previous section: simulation parameters are recorded and "published" to the interested services, triggering also the execution of the other services in the dataflow. Output and visualization plugins can be composed and configured using the VPL graphical editor, as depicted in Figure 6.9.

6.2 Related work and Future directions

We already mentioned Microsoft Research Trident previously in this Chapter as an example of a software built upon workflow technology [210]. Trident adds to WF domain specific activities, basic workflows, a library (which is a catalogue of existing activities and workflows), a registry (a catalogue of known services and data sets) and a visual composer part (called workbench) to support the creation and composition of scientific workflows. The goal is to provide basic elements that are useful and understandable by domain scientist, to let them compose, publish, track workflows without being developers or computer experts. Trident also adds to WF some valuable features, like the ability to schedule workflows on HPC clusters, the subdivision of activities and workflows by roles, services for automatic provenance and logging, and a web portal written in Silverlight that guarantees cross-platform access. It also has features what Trident authors define a "semantic" workflow discovery, which is a simple system to search for a particular workflow based on tags and annotations. While the underlying idea exposed by both Trident and our service-based framework is the same, the *granularity* and scope of application are different, making the two approaches complementary rather than opposite. In fact, for longer-running (coarser granularity) workflows, the runtime provided by WF (and therefore by Trident) offers support for checkpointing, persistence, resuming, which are desirable features. On the other end, it seems to be not suitable for fast, real-time message passing. Our service-based model, instead, aims at having better performance for message passing and communication of complex structures between services, making them suitable for composition fast, finer grained components. Our prototype based on DSS and the underlying CCR, despite some limitations highlighted in the previous Section, showed that this is achievable.

The Taverna Workbench [111] is an open source tool for designing and executing workflows created by the myGrid project. Taverna is very similar in design and philosophy to Trident: it is a family of tools for designing and executing workflows, that consists of

the Taverna Engine (workflow execution engine), the Taverna Workbench (desktop client) and Taverna Server (remote workflow execution server, which sits on top of the execution engine). Taverna allows for the automation of experimental methods through the use of a number of different services; notice that Taverna services, like Trident services, are not the small, fast components we introduced in our framework, but are mainly Web services. Services come from different domains like biology, chemistry but also meteorology and social sciences.

The Systems Biology Workbench (SBW) [109], instead, seems to be closer in spirit to our service model. Citing their documents *"SBW, is a software framework that allows heterogeneous application components to communicate and use each others' capabilities via a fast binary encoded-message system [...] a simple, high performance, open-source software infrastructure which is easy to implement and understand"*. The SBW message exchange system is implemented through a simple network protocol, which allows to potentially run different components on separate, distributed computers. It is, however, more oriented to programmers than to end users. In other words, it is more similar to a component or object broker[3] than to an orchestration framework. SBW enables interoperability and compatibility between components created using the framework, but components need to be instantiated an used inside a program or script; discoverability and self description capabilities limit somehow their ease of use (inside a visual programming and composition environment, for example).

On a related note, James II [226] provides a plug-in that adds an experimentation layer[4]. This experimental layer allows for the definition and execution of simulation experiments. In particular, it provides a framework for setting up and executing simulation experiments, focusing on correct handling of random number generators, replication, repeatability and choice of the simulation algorithm [104]. The layer greatly facilitates the configuration and usage of James; also, the plug-in system in James allows to develop more complex experiments as a composition of plug-ins [68]. However, the focus is on modelling and simulation: unlike Taverna, Trident or the proposed framework, the experimentation layer does not explicitly provide a way of composing different tools to build complex experiments, using an high-level, graphical language.

Finally, the *myExperiment* web portal is worth mentioning. myExperiment [51] is

[3]For example, like CORBA or DCOM

[4]Like many other modelling and simulation frameworks, James has a modular design based on plug-ins. However, as already noted, our focus goes beyond extensibility of a simulation framework, to cover all the different aspects of in-silico experiments, in which simulation is only one of the tools

not a workbench or a framework; it is rather a sort of "social network", a collaborative environment where scientists can publish their workflows in order to share expertise and avoiding reinvention. The myexperiment.org web site contains a large, public collection of workflows. Workflows span over multiple systems including Taverna and Trident.

In conclusion, Trident and Taverna are *workbenches* that allow a scientist with limited computing background to construct highly complex analyses reusing components and processes by the community; in some cases, they allow even groups with limited in-house computing resources to run those processes and analysis over public and private distributed computational resources.

On the other hand, SBW and James II are frameworks targeted at developers and designed to make easier for them to build interoperable tools. This is a great advantage for the community, as both developers and users benefit from this interoperability; the former group can build new solutions re-using or inserting into existing components; the latter can use software (for example, JDesigner) which, thanks to SBW, can provide many functionalities (different stochastic simulation algorithms, etc.) from different research groups. However, SBW and James II components need traditional programming techniques (e.g., writing plug-ins, instantiation and compose them from code,...) in order to be used; these techniques require skills that are not immediately acquired by every systems biology scientist.

Future directions

The BlenX-inspired service model was implemented, with some adaptations, on the Distributed Software Services framework. This prototype proved to be both quick to build, sufficiently easy to use and fast. As we previously highlighted in the Chapter, there were however some weak points; our future research and development will try to overcome these weak points.

Furthermore, we plan to investigate interoperability with workflow technologies, like Trident and Taverna: as we pointed out, these techniques share the same underlying ideas, but at a different level of granularity. Integration of our service model inside these workbenches would allow to use fast-little components for data crunching as well as workflows for lengthy, distributed processes.

Finally, as we mentioned in the introduction, the introduction of *versioning* and *reproducibility* in systems biology workbenches and toolkits could add immense benefits to researcher. These two features are related, as they both need to keep track and store information about models, experiments, steps and parameters used for simulation and analyses. The systems biology community has put a good effort in the definition of

standards dictating which information has to be recorded, that led to the creations of the MIASE (Minimum Information About a Simulation Experiment) standard with its formal representation SED-ML, and of the MIRIAM standard (Minimum Information Required in the Annotation of Models). However, software implementations of these standards are still scarce. A notable example is James, which through its plug-in system started to grow in this direction [68].

The integration of ideas and functionalities – in particular *versioning of models and results* and automatic *recording of changes, parameters and steps* of in-silico experiments – within the framework presented here would be another step in the direction of seamless management of systems biology experiments.

6.3 Summary

In this chapter we focused on the problem of the definition of complex in-silico experiments, where the user needs to move (*scale*) from a single simulation to more complex experiments where multiple tools are needed. In particular, we introduced a framework for the design and execution of general in-silico experiments. The framework allows to use different tools, exposed as services, and to combine them in a dataflow, allowing to build complex experimental protocols from basic building blocks. We compared our approach with other, related technologies: workflows, plug-in systems, simulation frameworks with support for experiments, highlighting similarities and differences.

Chapter 7

Case study: Evolution

In this chapter we examine how the techniques introduced and described in the previous Chapters helped our research group to create in-silico experiments of a scale larger than was possible previously. Moreover, we show how, using the techniques presented in this thesis, both the components and the process developed for this experiment can be reused, something that was difficult to achieve originally.

This case study is an updated version of [59] and [60], where some parts are updated using the results described in this thesis.

In [60], we presented an *evolutionary approach* to study biochemical networks. The approach is based on an *evolutionary algorithm*. Evolutionary algorithms are one of the first examples of cross fertilization between biology and computer science; algorithms and tools that mimic evolution are known from the pioneering work of Fogel in 1966 [76]. These algorithms are inspired by evolutionary biology concepts, like inheritance, mutation, selection and crossover. They have been applied to areas such as machine learning, optimization, search problems; however they soon lost their strict connection with biology and therefore brought advantages only to computer science.

On the other hand, approaches to study evolution are commonly based on comparative genomics or proteomics and on phylogenetic analysis [208, 9]. These studies compare networks from different organisms to infer how evolution affects the internal structure of the network of interactions.

Recently, however, there is an interest in understanding *how* networks emerged during evolution: knowledge of the process can help us to understand their basic properties, such as the role of complexity and the importance of topology and feedback loops. Therefore, alternative approaches emerged to simulate and re-create evolution *in-silico*; differently from *evolutionary algorithms*, they mimic the process in a very close and precise way [212, 169, 170, 163].

Up to now, these approaches have used ad-hoc tools and representations of network

dynamics, usually based on mathematical models, without exploiting the capability of the new computational and conceptual tools of systems biology.

Our goal is to blend evolution *in-silico* with computational models based on systems biology oriented languages, rejuvenating the mutual enrichment between biology and computer science. We develop a specific framework to allow straightforward study of network evolution based on the BlenX language and on the techniques described in Chapter 4 and 6. The great flexibility of BlenX in the definition of the structure of proteins allows us to introduce primitives for mutations used to build domain-based interaction and mutation models. Starting from the study of mutations at a biological level, we end up with some interesting program modifications that permit us to mutate the BlenX representation of biological entities in a meaningful and automatic way. Moreover, network dynamics can be easily modelled, and the interactions of emerged networks analysed.

7.1 Description of network dynamics

To study biological evolution of networks, we need a way to describe network dynamics and an algorithm to simulate network evolution from a generation to the next one.

7.1.1 A compositional model for signalling networks

A *signalling network* is any biological process that converts one kind of signal or stimulus into another; this conversion is also called *signal transduction*. In general, a signalling network results in a composition or cascade of biochemical reactions that are carried out by proteins and linked through second messengers. Biological signal transduction allows a cell or organism to sense its environment and react accordingly. Typically, a signalling network has one (or more) inputs, represented by any environmental stimulus, and one (or more) outputs, represented by an active protein.

We represent a protein in BlenX as a biological entity composed of a set of *sensing domains*, a set of *effecting domains* and an *internal structure*. Sensing domains are the places where the protein receives signals, effecting domains are the places that a protein uses for propagating signals, and the internal structure codifies for the mechanism that transforms an input signal into a protein conformational change, which can result in the activation or deactivation of another domain. This is inspired by the available knowledge of protein structure and function (see for example [215]).

Each biological entity is modelled with a box, which is a composition of an interface and an internal process unit. This gives an effective way for modelling proteins by decomposing the domains of interaction and the internal structure into two different constructs. Moreover, the compositional nature of the language allows us to design and apply

mutations on biological entities in an effective, simple and intuitive way.

We propose a general methodology for modelling proteins by providing patterns for modelling interaction domains and internal structures. Domains are represented using binders, i.e., interaction sites with an affinity. A *sensing domain* is represented by a binder, and the mechanism of message-passing is used to implement the reception of *activation* (e.g. phosphorylation) and *deactivation* (e.g. dephosphorylation) signals sent to the protein. The *effecting domain* is instead used to communicate, and so to activate or inhibit, other proteins. A general pattern for modelling a protein which can be activated by a single external signal was already presented in Sec. 3.1.2; we recall here the code for the pattern:

```
template pSTATE1s: pproc<<binder Base, binder Active>>  =
   rep state1?().(
      unhide(p).ch(recv,Active).
      recv?().hide(p).ch(recv,Base).state0!().nil);

template pSTATE0: pproc<<binder Base, binder Active>>  =
   rep state0?().(
      ch(recv,Base).recv?().ch(recv,Active).state1!().nil);

template SingleK: bproc<<binder K, binder Base, binder Active>> =
  #h(p, K), #(recv, Base)
[ recv?().state1!().nil |
  pSTATE0<<baseK3,activeK3>> |
  pSTATE1s<<baseK3,activeK3>> |
  rep p!(plus).nil ];
```

`SingleK` is a template for the initial (inactive) state of the protein. When an *inter-boxes communication* is executed through `recv` binder, the action `recv?()` is consumed, and the *intra-box communication* on channel `state1` is immediately executed. This triggers the sequence of actions `unhide(p)` and `ch(recv,Active)` (because their rates are `inf`). The obtained box has reached its active form, where the binder p is unhidden and the box can execute inter-communications through it. Now, if the box executes an *inter-boxes communication* through the binder `recv`, the reverse mechanism is executed and the protein returns back in its inactive form.

The pattern for modelling a protein which can be activated by receiving a signal twice is slightly different: five processes are put in parallel composition to represent the internal behaviour of the protein; one of them (`pSTATE0`) is the same as before, and another one (`pSTATE1`) needs a minor modification:

```
template pSTATE2: pproc<<binder Intermediate, binder Active>> =
   rep state2?().(
      unhide(p).ch(recv, Active).
      recv?().hide(p).ch(recv, Base).state1!().nil);

let pSTATE1: pproc =
```

```
   rep state1?().(
      recv?(what).what!().nil |
     (plus?().state2!().nil + minus?().state0!().nil));

template DoubleK: bproc<<binder K, binder Base,
                         binder Intermediate, binder Active>> =
  #h(p, K), #(recv, Base)
[ recv?().ch(recv, intK2).state1!().nil |
  pSTATE0<<Base,Intermediate>> |
  pSTATE1 |
  pSTATE2<<Intermediate,Active>> |
  rep p!(plus).nil ];
```

The `pSTATE0`, `pSTATE1` and `pSTATE2` processes encode the state machine that allows us to switch from the inactive to the active state and back. After receiving the first signal (with an activation mechanism similar to the one described for single signal activation), the process `pSTATE1`, representing an intermediate configuration, is activated. This process presents a choice behaviour: when a name *minus* is received, the process for the inactive state is enabled again; otherwise, if a name *plus* is received, the process representing the active state is enabled.

At the beginning, boxes are instantiated from the templates and are in the inactive form. Signals that enables the different internal processes are received when a box executes an *inter-boxes communication* through the binder `recv`. When the internal process `pSTATE2` is enabled, the box enclosing it has reached its active form. The reverse mechanism allows the protein to return back in its inactive form.

Obviously these patterns can be easily extended for modelling proteins with more than one sensing and effecting domains and for modelling mechanisms of activation based on the reception of more than two external signals.

As a final remark, note that the processes in these patterns can be seen as codifications for different *states*. The set of processes in these patterns are mutually exclusive, i.e. only one of the processes in the set is active at any given moment. Furthermore, upon a change each process enables exactly one process in the set before blocking on an input action. The set of processes act as a state machine; this behaviour will be useful for introducing *mutations* of processes in the next section.

7.2 Evolutionary framework

We propose a framework for simulating the evolution of networks *in-silico*. Evolution proceeds through selection acting on the variance generated by random mutation events. Individuals replicate in proportion to their performance, referred to as *fitness*. This pro-

cess can be modelled as shown in Tab. 7.1. This algorithm differs slightly from the generic

```
EvolutionAlgorithm ():
  Population := GenerateInitialPopulation();
  for i = 0 to generations do
    for each Individual in Population do
      output := Simulate(Individual);
      fitnesses[Individual] := ComputeFitness(output);
    NewPopulation := ReplicateAndMutate(fitnesses, Population);
    Population := NewPopulation;
```

Table 7.1: Generic EvolutionAlgorithm.

evolutionary algorithms used in computer science, being closer to real biological observations made for the asexual reproduction of organisms. Each individual in the population is codified using a BlenX program, and the boxes in each program are the abstraction of all the entities present in that individual. The interaction among these entities result in the behaviour of the network we want to study.

There are four main procedures in the algorithm:

- **GenerateInitialPopulation:** the initial population can be generated randomly, from a predefined network configuration to be used as a starting point, or it can be a network with no interactions. All the individuals in the initial population can be equal at the beginning, as they will be differentiated later by the mutation phase.
- **Simulate:** each individual in the population is simulated separately using the BetaWB stochastic simulator.
- **ComputeFitness:** the output of the simulation is used to compute the fitness value of the current individual. Note that the fitness value is problem-dependent; for an example, refer to Sec. 7.4.
- **ReplicateAndMutate:** this is the most important part of the algorithm; like in a real environment, individuals with the highest fitness values are more likely to survive, replicate and produce a progeny that resembles them, being not, however, completely equal to them.

The ReplicateAndMutate algorithm (Tab. 7.2) creates a new population with the same number of individuals of the current generation, using as a base the current individuals. At each step it chooses one individual, with probability proportional to its

```
ReplicateAndMutate (fitnesses, Population):
  for  i = 0  to  i < Population.Size  do
    Individual := ChooseOneIndividual(Population, fitnesses);
    for each  Protein  in  Individual.Proteins  do
      if  Random() < DuplicationProbabily
        Protein2 := Protein.Duplicate();
        Individual.Proteins.Add(Protein2);
      for each  Domain  in  Protein.Domains  do
        if  Random() < MutationProbability
          MutationType := GetRandomMutation();
          if  IsMutationFeasible(Domain, MutationType)
            Domain2 := Individual.PickCompatibleDomain(Domain, MutationType);
            Individual.Mutate(Domain, Domain2, MutationType);
    NewPopulation.Add(Individual);
  return  NewPopulation;
```

Table 7.2: The ReplicateAndMutate algorithm.

fitness (ChooseOneIndividual in the code above). This is achieved by constructing a cumulative probability array a from the *fitness* array, generating a random number in the range $0...a[Population.Size]$, and then finding the index into which the random number falls.

The selected individual will replicate and pass to the next generation. During the replication, each protein in the "genome" of the individual is given the chance to mutate, according to a probability.

A mutation is selected among all the possible types by the GetRandomMutation function, and this mutation is applied. Finally the individual, which can be either equal to its predecessor or mutated, is added to the new population. We now define in more detail the mutations that we consider in our framework.

7.2.1 Mutations

Here we consider the end-effects of point mutations occurring at the DNA level. These mutations ultimately affect network dynamics. For example, mutations in a DNA sequence can change the protein amino-acid sequence, leading to changes in its tertiary

structure with implications on the affinity of this protein with other proteins or substrates. Similarly, events at DNA level as gene duplication or domain shuffling can alter network structure and dynamics.

A computer program which is used to mimic evolution of a species must implement random mutations in individuals during replication as well. Here, we can easily implement these molecular processes using the domain and network model we discussed in Sect. 7.1.1.

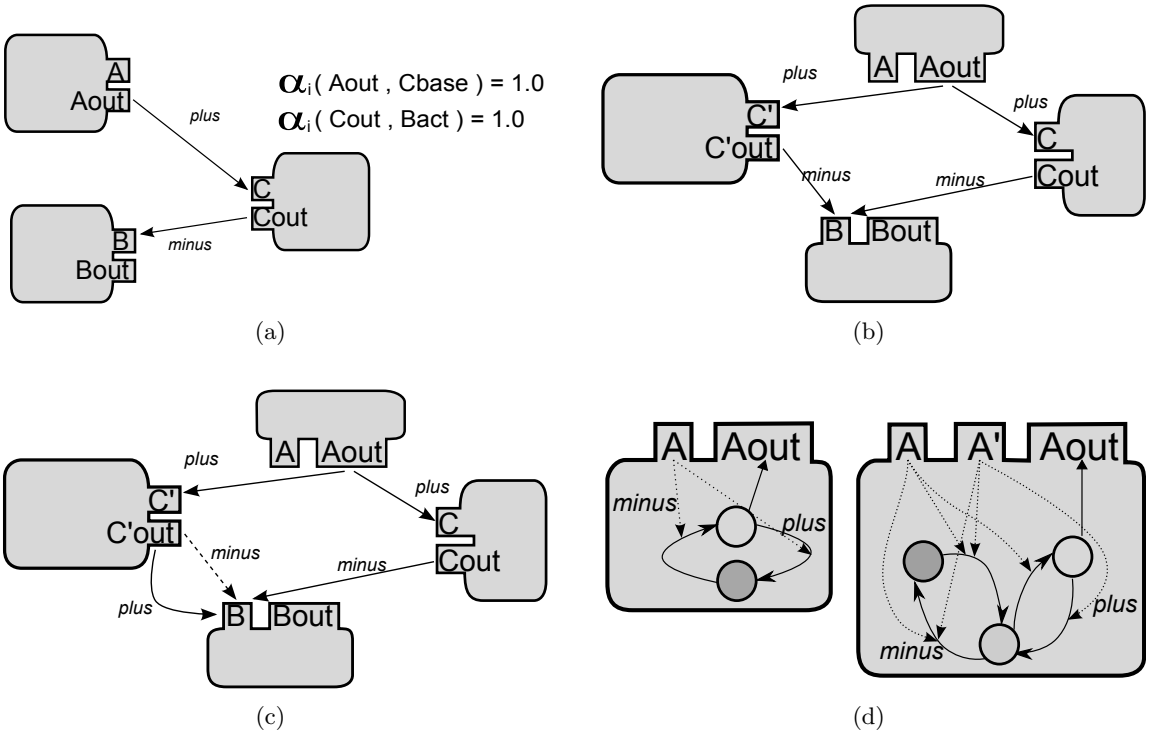


Figure 7.1: Different kinds of mutations: in (a) the initial configuration, displaying the α function as a list of tuples; in (b) duplication of protein C followed by mutation of domain Δ^{C}_{out} in (c). Finally, (d) displays how the internal structure could change to accommodate the duplication of a domain.

We will take as an example the three-protein network represented in Fig. 7.1(a) and we will illustrate how different mutations can be modelled in BlenX.

Duplication and deletion of proteins

Gene duplication at DNA level is implemented with a duplication of the box associated to the protein the gene codifies for. The new box will have the same internal structure and the same binder names, while binder identifiers will be new but will have the same interaction capabilities. This is achieved by copying the affinities of the original binder identifiers. Duplication of binder identifiers is needed because subsequent mutations on one of the binders of the duplicated protein must not affect the original one. Furthermore, since the new protein is a new distinct entity, it must not be structural equivalent to the

original one. The same considerations hold for the internal processes: duplication and deletion of domains may lead to a modification of the internal structure (see next section); the internal processes must be duplicated so that each box has its own, distinct internal behaviour. Using the templates presented in Sec. 7.1.1, the box for protein `C`

```
let C: bproc = DoubleK<<outC, baseC, intC, activeC>>;
```

will be duplicated to

```
let C: bproc = DoubleK<<outC, baseC, intC, activeC>>;
let C1: bproc = DoubleK<<outC1, baseC1, intC1, activeC1>>;
```

Deletion of a protein is accomplished by deleting the associated box, the internal process it refers to and the appropriated entries in the α function.

Mutation of domains

Point mutations in DNA can change the protein amino-acid sequence, and consequently lead to the mutation of a domain and to changes in the interaction capabilities of the protein to which it belongs. In our formalism, this is achieved by changing the α function on the two domains that take part in the interaction. More specifically, the mutation on a domain can be a *change of interaction*, for which we modify the affinity adding a number sampled from a normal distribution, an *addition of an interaction* between two domains $d1$ and $d2$, modelled as the addition of an affinity $\alpha(d1, d2) = x$, with $x > 0$, and finally a *removal of an interaction* between two domains $d1$ and $d2$ setting $\alpha(d1, d2) = 0$. For example, the mutation on domain `outC` that can be observed in Fig.7.1(c) is obtained by changing the compatibility from $\alpha(outC, activeB) = 1.0$ to $\alpha(outC, activeB) = 0.0$, $\alpha(outC, baseB) = 0.9$; in this way the internal process of `C` is now allowed to send a *plus* message when the `B` process is in an inactive state, represented by the binder identifier `baseB`.

Duplication and deletion of domains

Domain duplication or deletion is more complex as it involves not only interfaces or rates, but requires also modification of the internal behaviour in response to stimuli.

Duplicating or removing domains can be easily done acting on the binders list and on the affinity function α; however, for these domains to act as sensing or effecting domains in cooperation or in antagonism with the existing ones, the internal behaviour of the process must also be changed. We devised several possible modifications of the behaviour when a domain is added.

As an example, consider the case of a sensing domain: when a signal arrives -by means of a ligand binding, or by phosphorylation of a residue- the internal behaviour of the

protein changes bringing it to a different *state*. If that domain is duplicated, the internal behaviour must be changed accordingly. The second domain may act *concurrently* with the old domain, with the result that the activation of this second domain will bring the protein in the same state as the old one, acting in parallel. This is the case, for example, of a receptor that can bind to two different signal molecules. Alternatively, the duplicated domain can affect the capability of the protein to reach that state, and so must act *in coordination* with the original one; this is the case of kinases that must be phosphorylated twice to activate (double phosphorylation, as in Fig. 7.1(d)).

These mutations are obtained by manipulating the structure of the internal process to *transform* their behaviour. In both cases, we assume that the internal process have a standard structure, as described in Sec. 7.1. This process is built through parallel composition of different processes, each representing a different *state*. The set of processes in parallel is a set of *mutually exclusive* ones: at any given time only one of the processes can be active (e.g. not blocked waiting for a communication). Moreover, each process in the set *enables* another one by issuing a communication immediately before blocking itself.

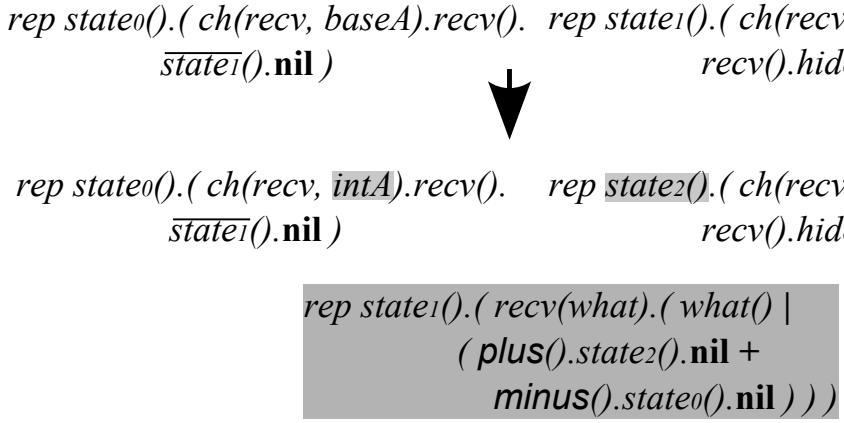


Figure 7.2: Transformation for the modification of a sensing domain, introducing a new *state*. Light gray highlights the modified actions, dark gray the newly introduced ones.

In the case of "cooperative" domains, where a signal on both is required to reach the desired internal configuration, the transformation can be accomplished by substituting the process codifying for the current active state with a new process, adjusting the channel names used for the intra-communications and binder identifiers accordingly. In Fig. 7.2, for example, it is shown how it is possible to manipulate an internal process to transform a protein activated (or deactivated) by a single phosphorylation into a protein that is activated (or deactivated) by a double phosphorylation, encoding an intermediate step of "half-activation".

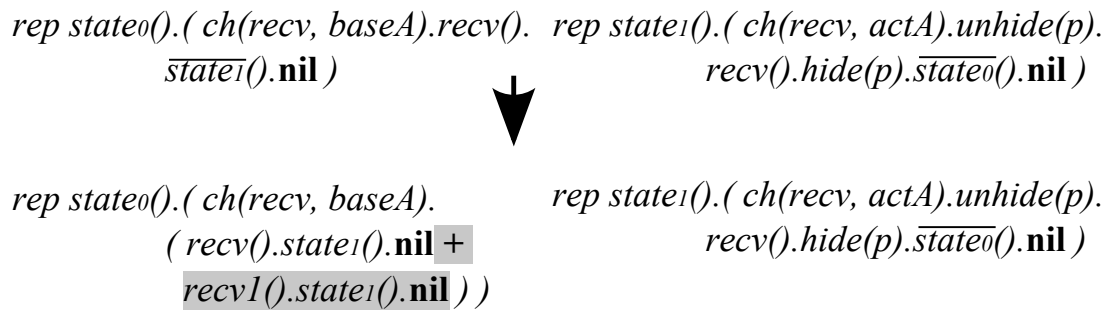


Figure 7.3: Transformation for the modification of a sensing domain. Light gray highlights the introduced actions.

The case of *concurrent*, or *competitive* domains, where each of the signals can lead to the desired internal configuration, can be handled in a similar way; in this case however the process representing the state is substituted with a different process (see Fig. 7.3).

Deletion of a domain requires to undo the steps done while duplicating it. This task is accomplished again by transformation of the internal process, restoring the behaviour to the original one.

7.2.2 Measure of fitness

When analysing evolution of specific biological systems, one needs to consider the "fitness" benefit of that system to the organism (i.e. to its reproductive success). While it is usually complicated to define and measure such fitness contribution, network dynamics can provide a good proxy in case of biological networks. As the concentrations of the proteins involved in such networks will define the proper functioning of the network, how these concentrations fit a specific time course would determine how well the network "operates".

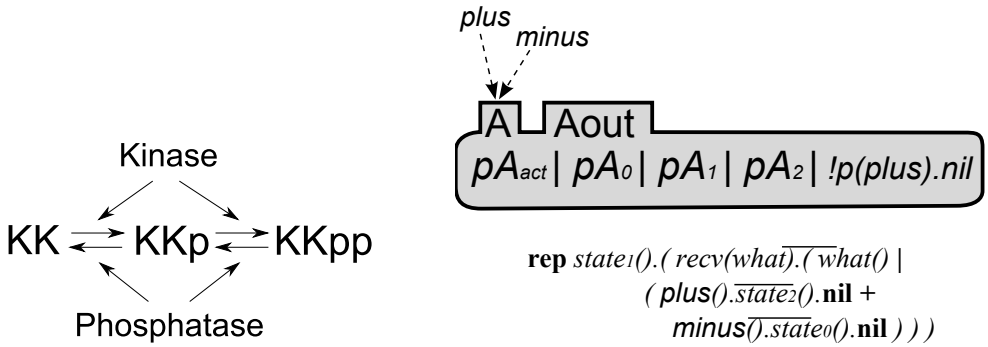
We will illustrate in our example how fitness can be computed using the integration of a response.

7.2.3 Constraints

We understand that with our framework it is possible to generate countless combinations, interactions and mutations. Many interactions or mutation can be possible and have sense from the point of view of a program syntax and semantics, but have little or no sense from the biological perspective. We addressed this issue by providing a configurable way of specifying constraints on mutations, their probability and which class, or type, of protein or domain they can affect.

7.3 Implementation

We implemented our evolutionary framework using the BlenX language and its supporting tools (The Beta Workbench, introduced in Sec. 3.2).BetaWB was modified to run within the experimental service framework introduced in Chapter 6.



(a) (b)

```
new PiReplication(
   new PiAction(new ActionIO("state1", "", ActionType.INPUT)),
   new PiParallel(
      new PiAction(new ActionIO("recv", "what", ActionType.INPUT),
         new PiAction(new ActionIO("what", "", ActionType.OUTPUT),
            new PiNil())),
         new PiChoice(
            new PiAction(new ActionIO("plus", "", ActionType.INPUT),
               new PiAction(new ActionIO("state2", "", ActionType.OUTPUT),
                  new PiAction(new PiNil()))),
            new PiAction(new ActionIO("minus", "", ActionType.INPUT),
               new PiAction(new ActionIO("state0", "", ActionType.OUTPUT),
                  new PiAction(new PiNil()))))));
```

(c)

Figure 7.4: Representation of a protein -a kinase with two phosphorylation sites-: (a) its textbook description; (b) as a box, with the part of its internal process that encodes its intermediate configuration; (c) the program code to generate programmatically part of the internal process.

Each *individual* in our evolutionary framework is represented by a BlenX program. We used the BetaWB simulator to execute the models and obtain their time courses; we built a new tool to compute the fitness based on the simulator output files, and a new tool to manipulate and mutate the BlenX programs, based on the BetaWB libraries.

The challenging part was to implement mutations of BlenX programs. The first three kind of mutations introduced in Sec. 7.2 act on the α function, and so it is possible to obtain them by manipulating the BlenX binder definition file, a file that stores in a tabular way the α function. However, mutations of the fourth kind (duplication and deletion of

domains) lead to changes in the internal behaviour. These changes are done directly on the program, by exploiting the executable nature of BlenX. Models written in BlenX are not meant to be solved (like other formalisms, for example those based on ODEs); instead, the model code is compiled into a format that is understood and executed by the simulator *virtual machine*.

The BetaWB compiles the model *Just in Time*: the simulator takes the source code for the model and compiles it into an Abstract Syntax Tree (AST). This tree is the object model of the processes discussed in Sec. 7.2. *Transformations* discussed in that section are implemented by manipulating and navigating in a programmatic way the Abstract Syntax Tree. Our libraries allow us to access the AST and write it to the disk, generating a new and perfectly valid BetaWB textual model.

As an example, consider Fig. 7.4, that reports three representations of the same biological entity, a two-level kinase. In the upper left corner, the kinase is depicted using the standard representation used in biological papers and textbooks. The figure in the upper right corner represents the same object as a box with an internal process codifying for different *states*. The BlenX code immediately under the box represents one of those processes, that codifies for the intermediate state. The process is recurring (using the `rep` operator) and reacts to signals on the binder representing the sensing domain. When a signal arrives, the name passed through the communication channel is used to understand if the protein represented by the box was phosphorylated or de-phosphorylated; a message is then sent to exactly one other process in the set, the one representing the active state in the first case and the one representing the completely inactive case in the second. The lower part of Fig. 7.4 depicts how the code for the internal process representing the intermediate state can be generated programmatically using our library.

We built an AST analyser that is able to recognize some patters in the tree and modify them accordingly. The possibility of recognizing patterns on the tree, of generating programmatically parts of it and adjusting the remaining part can be used to produce all the possible mutations at the level of internal behaviour. Transformations introduced in Figures 7.2 and 7.3, for example, are implemented in the way shown in Figures 7.5 and 7.6 respectively.

In the original paper [60], all the tools in BetaWB are orchestrated by a *driver*. The driver application runs all the simulations on a single machine, in a sequential way.

In Chapter 6, we introduced a modular framework for in-silico experiments; one of the driving motivations in its design is the ability of writing small components to perform specific functionalities, and then being able to compose them using a graphical environment in a seamless way. As Figure 7.7 shows, this programming model suits very well to our evolutionary simulation scenario.

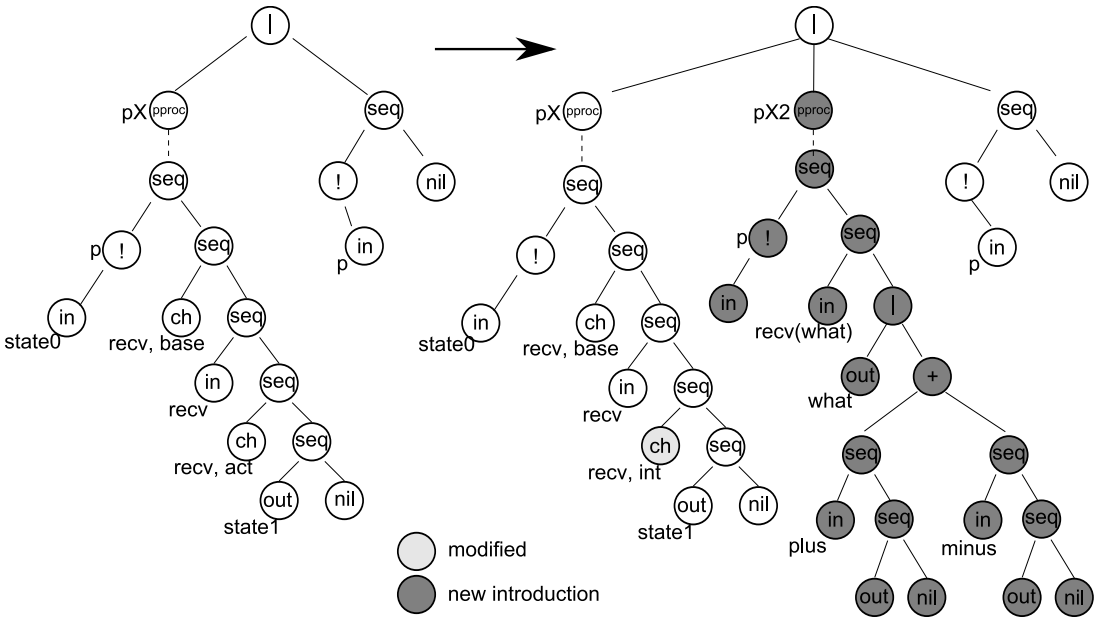


Figure 7.5: AST transformation for the *cooperative* modification of a sensing domain.

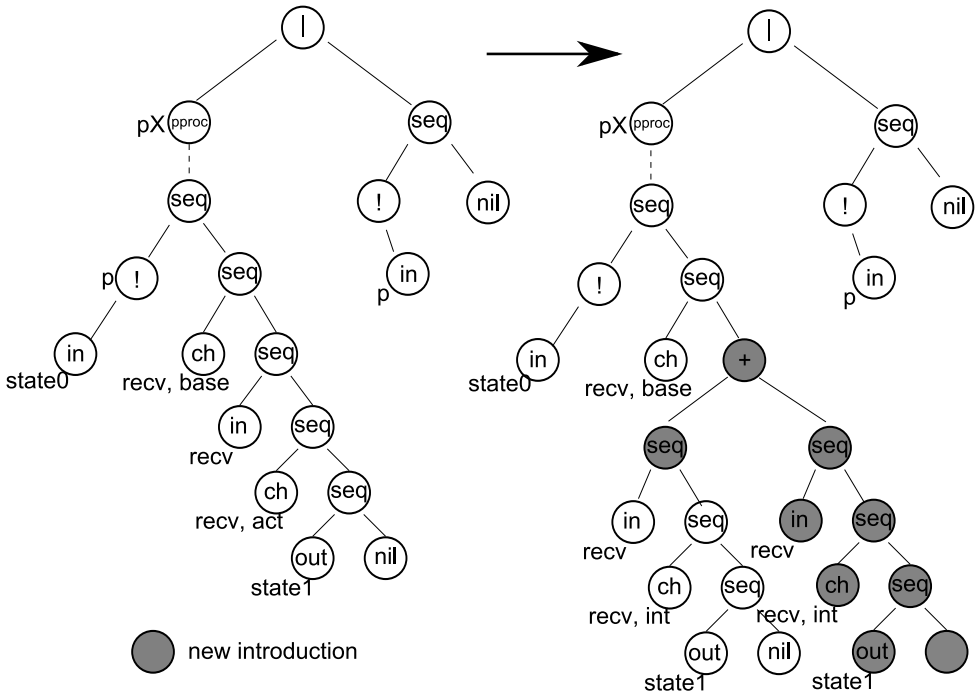
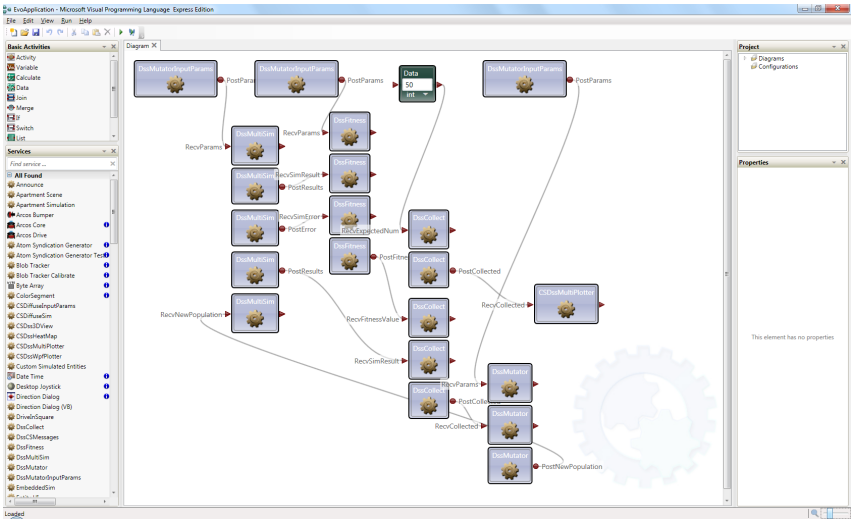


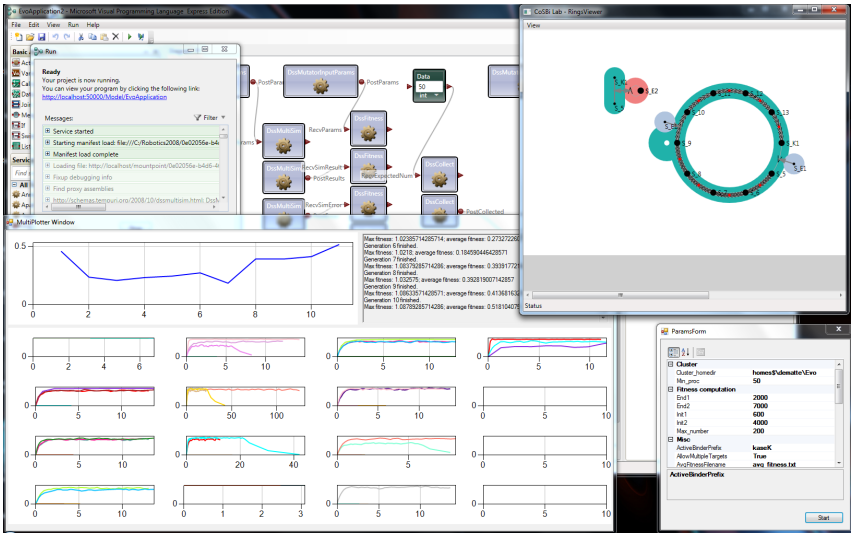
Figure 7.6: AST transformation for the *competitive* modification of a sensing domain, with a full gain in performance.

We updated the original project by dividing the application into multiple services: a *runner* service encapsulating the BetaWB simulator; a couple of *composer* services

to host the mutation and replication code; a *visualization* service to trace the progress of the simulation; another *visualization* service, implementing the network visualization introduced in Section 5.2.1, to inspect the resulting networks and other services for support and bookkeeping.



(a)



(b)

Figure 7.7: The evolution application running as an ensemble of DSS services. Notice the "Rings" service, displaying a network with high fitness value.

Furthermore, evolutionary simulations are a trivially parallelisable problem: the MRIP approach described in Chapter 4 is sufficient in this case. Each generation depends on the previous one, but within each generation each individual *lives* (e.g. is simulated) independently from the other. The *foreach* block of the evolutionary algorithm in Tab. 7.1

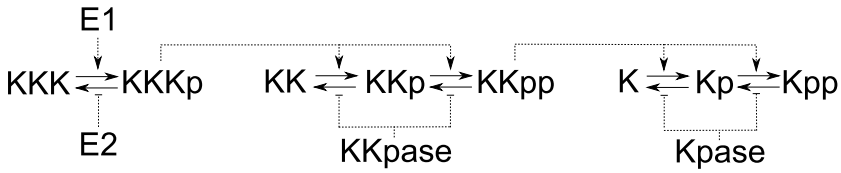


Figure 7.8: MAPK cascade as described in [107]. KKK denotes $MAPKKK$, KK denotes $MAPKK$ and K denotes $MAPK$. The signal $E1$ transforms KKK to $KKKp$, which in turn transforms KK to KKp to $KKpp$, which in turn transforms K to Kp to Kpp. In particular, when an input $E1$ is added, the output of Kpp increases rapidly. The transformations in the reverse direction are the result of the signal $E2$, the $KKpase$ and the $Kpase$. In particular, by removing the signal $E1$, the output level of Kpp reverts back to zero.

can be split in concurrent threads of execution.

Our services takes advantage of this fact by running simulations on different computation nodes on a cluster of PCs, distributing the load between nodes and even, thanks to their underlying usage of the Coordination and Concurrency runtime, among multiple cores. Results are then gathered and combined by another service at the end of each generation.

7.4 An example: MAPK cascade

The mitogen-activated protein kinase cascade (MAPK cascade) is a series of three protein kinases which is responsible for cell response to growth factors. In [107], a model for the MAPK cascade was presented (Fig.7.8) and analysed using ODEs; the cascade was shown to perform the function of an ultra-sensitive switch and the response curves were shown to be steeply sigmoidal. A process calculi based analysis of the MAPK cascade was presented in [173]. For simplicity, in this paper we rely on a simplified version of the model, where all the enzymatic reactions of the form:

$$E + S \rightleftharpoons_{K_{ES}^{-1}}^{K_{ES}} ES \multimap^{K_{EP}} EP \multimap^{\infty} E + P$$

are substituted with simplest reactions of the form:

$$E + S \multimap^{K_{EP}} E + P$$

Using the design patterns presented in 7.1, a model for the MAPK cascade has been developed (see Tab.7.4). Notice that differently from [60], the model has been updated to the latest version of BlenX, using the constructs introduced in Sec. 3.1.4; therefore, the model is more compact and readable.

```
// Signal template
template Signal: bproc<<binder S>> = #(e,S)
   [ rep e!().nil ];

// Phospatase template
template Phospatase: bproc<<binder P>> = #(x, P)
  [ rep x!(minus).nil; ]

// Kinase templates: with single or double activation
template pSTATE2: pproc<<binder Intermediate, binder Active>> =
   rep state2?().(unhide(p).ch(recv, Active).
      recv?().hide(p).ch(recv, Base).state1!().nil);

template pSTATE1s: pproc<<binder Base, binder Active>>  =
   rep state1?().(unhide(p).ch(recv,Active).
      recv?().hide(p).ch(recv,Base).state0!().nil);

let pSTATE1: pproc =
   rep state1?().(recv?(what).what!().nil |
     (plus?().state2!().nil + minus?().state0!().nil));

template pSTATE0: pproc<<binder Base, binder Active>>  =
   rep state0?().(ch(recv,Base).recv?().ch(recv,Active).state1!().nil);

template SingleK: bproc<<binder K, binder Base, binder Active>> =
  #h(p, K), #(recv, Base)
[ /*... see Chapter text */ ];

template DoubleK: bproc<<binder K, binder Base,
                         binder Intermediate, binder Active>> =
  #h(p, K), #(recv, Base)
[ /*... see Chapter text */ ];

// Definition: activation and deactivation signals
let E1: bproc = Signal<<signalE1>>;
let E2: bproc = Signal<<signalE1>>;

// The three kinases (one with single phosporilation, two
// with double phosphorilation)
let K3: bproc = SingleK<<kaseK3, baseK3, activeK3>>;
let K2: bproc = DoubleK<<kaseK2, baseK2, intK2, activeK2>>;
let K1: bproc = DoubleK<<kaseK1, baseK1, intK1, activeK1>>;

// The 2 phosphatases
let P1: bproc = Phospatase<<paseP1>>;
let P2: bproc = Phospatase<<paseP2>>;

when (E1 : step = 1500 : ) delete(2);

run 20 K3 || 200 K2 || 200 K1 || 2 E1 || 2 E2 || 2 P1 || 2 P2
```

Table 7.3: Complete model of MAPK in BlenX

Following [173], we set all the reaction rates to a nominal value of 1.0 and we initialize the system with two of $E1$, $E2$, $KKPase$ and $KPase$, 20 of KKK and 200 of KK and K. Simulating the MAPK system with the BetaWB simulator, similar response profiles (modulo timescale) were observed for the output of Kpp with respect to the model presented in [107], despite the differences in the simulation parameters; the system still behaves as an ultra-sensitive switch.

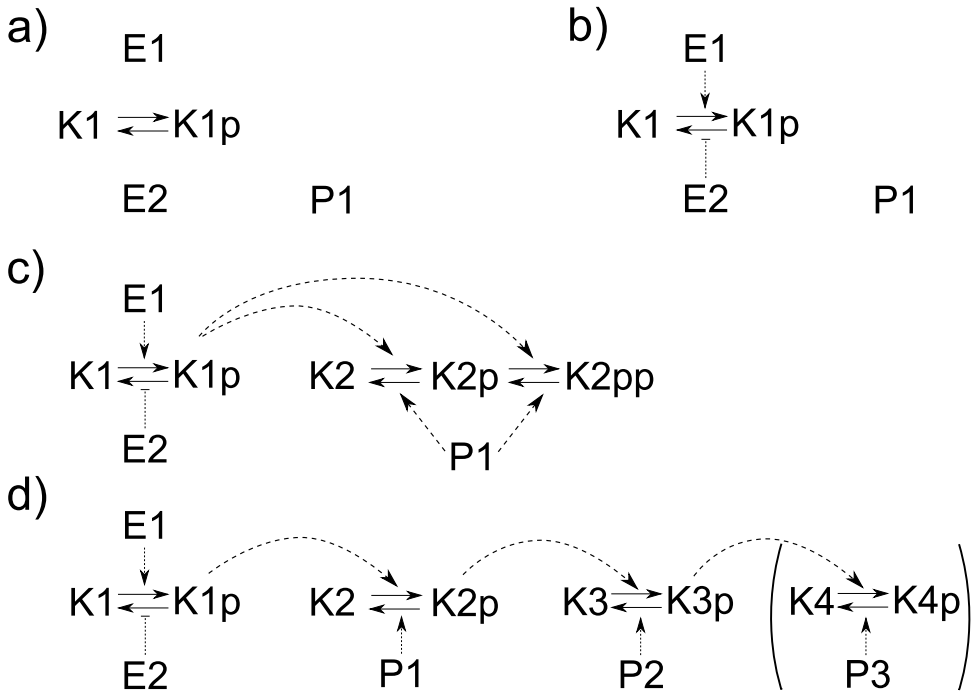
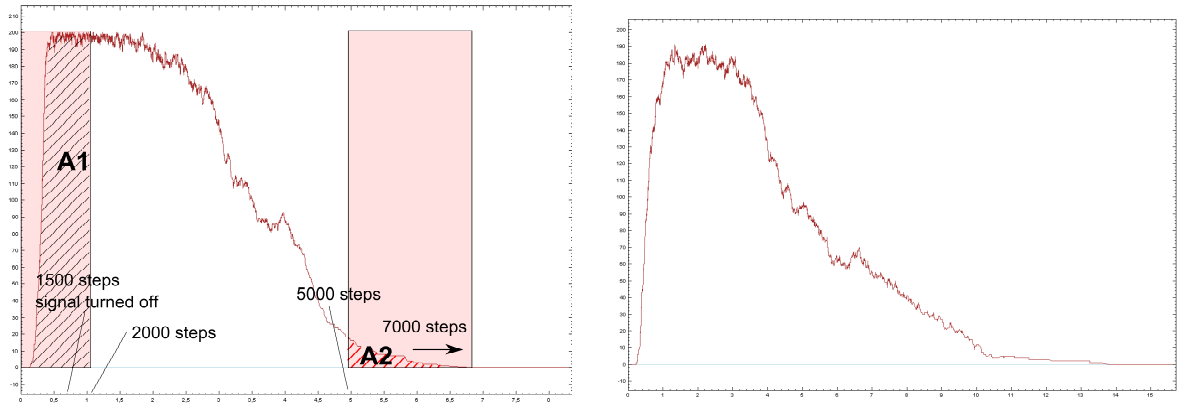


Figure 7.9: (a) Basic individual of the initial configuration. (b) Only signals $E1$ and $E2$ are enabled. (c) A particular individual we obtained, with a two-level phosphorilation. (d) An alternative evolution, with single phosphorilation kinases but a longer cascade.

We use this simplified MAPK cascade system as a starting point for testing our evolutionary framework. In particular, we want to analyse the evolution of a population according to a fitness function which captures the essential behaviour of our MAPK cascade model.

In detail, we generate an initial population of 500 individuals containing the network shown in Fig.7.9a. We set up very general initial conditions, with a single kinase $K1$, a single phosphatase $P1$, an activation signal $E1$ and a deactivation signal $E2$; the model lacks any interactions among entities. In other words, we consider an ancestral organism that possessed all the base proteins but lacked a signalling system similar to the MAPK cascade as observed today. The dynamic of each individual is then simulated; we run each individual for 7000 simulation steps and we remove the signal $E1$ at the step 1500 using a

time-triggered *delete* event, introduced in Sec. 3.1.1. Using the output of the simulation, we then measure for each individual the corresponding fitness. The fitness function we



(a) (b)

Figure 7.10: (a) Time course of the Kpp concentration over the simulation time, superimposed to the integral areas for the fitness function we implemented. The fitness parameters are $i_1 = 0$, $e_1 = 2000$, $i_2 = 5000$ and $e_2 = 7000$. (b) Time course of Kpp for a network with high fitness.

implemented measures how rapidly the output of an active kinase increases, how much the output of the same kinase persists after removing the signal $E1$ before returning back to the initial condition. Let $out = \{n_0, n_1, ..., n_{7000}\}$ be the tuple representing the active kinase K^* dynamics in time of an individual, then the fitness for out is formally computed by the following formula:

$$fitness(out) = \mu + \left(\frac{\sum_{j=i_1}^{e_1} n_j}{K_M^* * (e1 - i1)} - \left(\gamma * \frac{\sum_{j=i_2}^{e_2} n_j}{K_M^* * (e2 - i2)} \right) \right)$$

The two sums, that we denote respectively with $A1$ and $A2$, represent discrete integrals and are normalised with respect to their possible maximum values (see Fig.7.10). The values i_1, e_1, i_2 and e_2 are changeable parameters that define the boundaries for the computation of the two discrete integrals present in the formula, and the value K_M^* represents the maximum value for the K^* response. Moreover, μ represents the minimum fitness and γ controls the relative importance to responding to a signal and turning the response off after its removal. The reported results are for $i_1 = 0$, $e_1 = 2000$, $i_2 = 5000$, $e_2 = 7000$, $K_M^* = 200$, $\mu = 0.1$ and $\gamma = 0.75$.

Notice that we implemented the fitness computation as a separate, customizable service. Configuration information provided through the *control* port can be used to adjust the aforementioned parameters; if a completely different fitness measure is needed, the service can be replaced altogether with a different one, without the need to change or recompile the application.

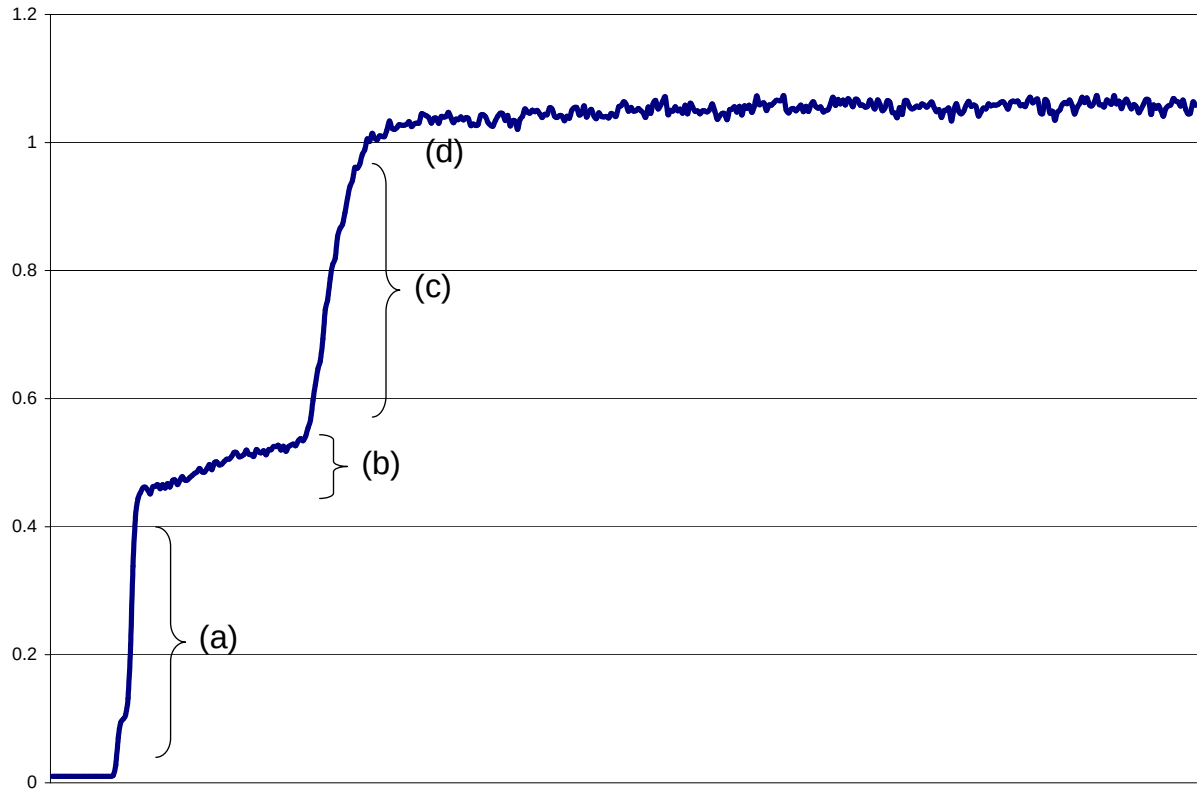
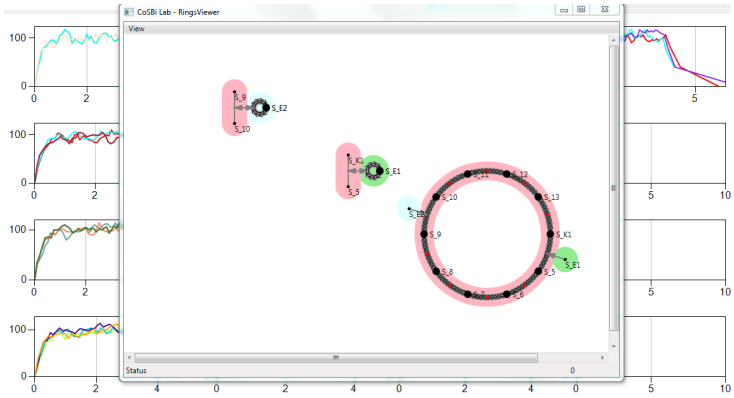


Figure 7.11: Changes in fitness during a typical evolutionary simulation.

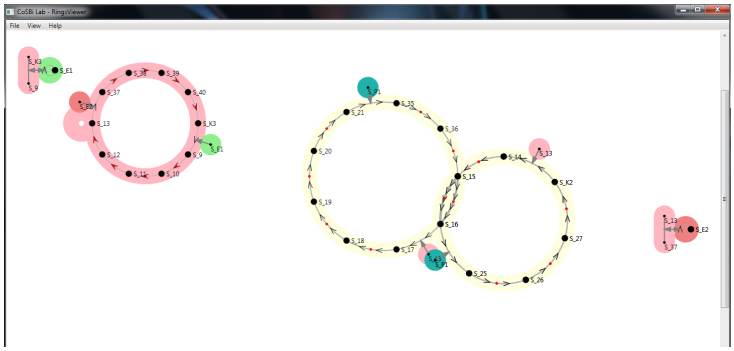
According to the algorithms presented in the previous section, the population is evolved. Differently form previous studies we made on the same model, we do not limit mutations to *mutations of domains*, but we included also duplication and deletion of proteins and domains. In order to maintain a biological validity for the new individuals, possible mutations are the one that satisfies the following constraints: (1) signals $E1$ and $E2$ cannot be removed; (2) a kinase can only activate other kinases or itself; (3) kinases are specific (e.g. they do not phosphorilate multiple proteins); (4) phosphatases are not specific but can only deactivate kinases.

We iterated the evolution algorithm for 2000 generations, for different values of fitness function parameters.

We then inspected the generated models using the Plotter and Designer tools from BetaWB (see Sec. 3.2) and the "Rings" network visualizer introduced in Chapter 5. The dynamic behaviour of one of the obtained networks is shown in Fig.7.10(b); examples of obtained individuals are in Fig. 7.9. The network visualizer (Sec. 5.2.1) allowed us to quickly inspect the result networks, to capture at a glance interesting activation patters. For example, in Figure 7.12 (a) and (b) we can see how the networks for individuals



(a)



(b)

Figure 7.12: Individuals displayed in the Rings network visualizer

Fig. 7.9(b) and (c) respectively, look like in our application.

In particular, we did not obtain individuals with a perfect MAPK cascade network, but individuals in (c) and (d) have very good fitness values and show the two directions in which evolution went to build an ultra-sensitive switch, namely forming longer cascades with multiple kinases or having multiple phosphorylation sites. We did not obtain in our runs individuals whose networks combined the two characteristics; we suspect that this fact may be due to the fitness function, that reached its maximal values with the two configurations in (c) and (d). We will conduct more experiments with a more sensitive fitness function. As a final note, interactions within kinases and phospatases shown in (c) and (d) are only an example: we obtained also individuals with very complex relations (self-activations, "reverse" activations -where for example $K1$ activated $K2$ and $K2$ activated $K1$- and so on).

The variation of fitness during a simulation is depicted in Fig. 7.11. Note the "steps" in the fitness. We observed this typical behaviour in almost all our runs. In the first generations, individuals have to find the correct signal: the *jump* in (a) is realized when the

the activation signal $E1$ hits one of the kinases. In (b) instead we have the slow adaption to the introduction of the deactivation signal: the presence of the signal allows the cascade to be switched off, but reduces the gain of the switch in response to an activation signal. The second *jump*, in (c), is where double phosphorylations or more kinases are added to the cascade, allowing the network to re-gain the lost efficacy and react in a steep way to the activation signal. The last phase, (d), is where more phosphatases are added in order to switch off the response in a quicker way.

Interesting individuals, selected using fitness and complexity measures (e.g. number of proteins and/or domains and/or interactions) were chosen for further inspection. The network visualizer introduced in Section 5.2.1 allowed us to quickly inspect visually these networks: it was possible, for example, to establish at a glance how many components were involved in a cascade, how signals and kinases interacted (acting in a single or in multiple points), how fast were the reactions and how many intermediate steps were formed.

The Rings application helped us in finding that particularly fit individuals showed either a good number (¿=4) of single-phosphorilated kinases on a cascade (showing as many simple interacting rings) or a few number of kinases (2-3) with many independent phosporilation sites (showing as few multi-rings or complex networks). Again, components for selection and visualization were inserted in the tool-chain in a very simple way, using our framework.

7.5 Related work and Future directions

This chapter presented how the tools introduced in this thesis can be combined and used to construct a formal approach and a working implementation for simulating the evolution of networks *in-silico*.

Network dynamics are described with the BlenX language, which allowed us to develop a modular description of signalling networks. The programming language approach adopted for modelling allowed us to describe mutations in a novel and modular way as well: BlenX proved to be well-suited for this task, allowing us to implement mutations as program transformations.

Following the approach described in Chapter 6, we used a service-oriented framework to develop various services, and then we composed them together to create an evolutionary algorithm. We also used the MRIP approach mentioned in Chapter 4 to speed up the evolutionary simulation. The small example in Sec. 7.4 showed the potential of our approach; by simulating evolution we can gain interesting insights in network topology, proteins structures and interactions and on the role of different processes.

Future work in this area proceeds along two main lines. Other researcher in our group

are currently exploring the usage of the same framework from an optimization perspective, to help discovering and inferring biochemical network topologies from incomplete data. At the same time, we plan to use the features in the new version of BlenX to perform evolutionary computations and simulation of evolution inside a single BlenX model, as we mentioned at the end of Chapter 3. Our aim is to be able to express mutations, replications and multiple generations within the language, using first order functions operating on binders with higher order types.

7.6 Summary

In this chapter we presented a realistic case study that uses some of the methods introduced in this thesis. In particular, we showed how the modular features of BlenX (Chapter 3) allow to express complex models in a simple, composable way; we showed how the framework introduced in Chapter 6 allows to orchestrate different tools in a simple, graphical way to obtain an in-silico experiment that is more complex than a single simulation; we used a simple but effective parallel execution technique, introduced in Chapter 4. We then examined the results of the in-silico experiment using the network visualization technique introduced in Section 5.2.1.

Chapter 8

Conclusions

In this thesis we faced the problem of applying systems biology to large scale systems; while Systems Biology promises to be able to cope with problems, hypotheses and research at a system level, some of the methodologies, theories and underlying techniques are yet in a non-mature state.

Research done during my PhD studies, summarized by the contents of this thesis, defined five main areas of intervention: modelling languages, spatial simulation algorithms, parallel simulation algorithms, tools for the interpretation of results of *in-silico* experiments and tools for the composition of *in-silico* experiments.

Work on modelling languages include my research as part of the team that designed, formalized and implemented the BlenX language. A programming language approach promises to enable the construction of very complex models: after all, software systems are widely recognized as the most complex thing human beings created, more complex then aircrafts, cars, or computer hardware itself. Therefore, the idea is that programming languages and their related methods, tools and practices, could be used to scale up the size of the problems we are able to model and analyse, and help to master the extreme complexity of biological processes. BlenX was built upon the beta-binders process algebra, inheriting concepts like affinity-based interactions and encapsulation, to create a language that can be used to model biological systems in a very compact and modular way. During the process of creating the language, several other concepts found their way into BlenX; we designed a language more tailored to the modeller needs in terms of ease of modelling and expressiveness. Features like events, variables, rate expressions, continuous functions, made the language easier to use in models of realistic biological systems, where some details need to be abstracted away in order to concentrate on the central aspects.

Scaling up the size of the biological problems we are able to treat requires advances in the simulation methods. On the one hand, explicit handling of space in models and simulation algorithm is necessary if the model goes beyond the level of a single pathway:

the most used family of simulation algorithm, Gillespie SSA and its variants, do not take space into account. On the other hand, parallel execution is key to obtain acceptable simulation performances, especially when we consider that we are living a paradigm shift in both hardware architectures and software development, from high speed single core machines and sequential programs to many-core architectures and the parallel software.

Therefore, we devised a method and an algorithm to simulate movement and diffusion though a crowded, inhomogeneous medium, like the cell cytosol. Then, we analysed the techniques that can be employed to parallelise the execution of stochastic simulations. We presented the the practical implementation of a species-based method and of an individual-based method.

Visualization, especially when dealing with large spatial model (like in developmental biology), or with very complicated reaction networks, is key to understand and interpret simulation and analysis experiments. This thesis focuses on visualization techniques to interpret the results of simulations, with the development of a new technique to visualize reaction graphs, a new framework to visualize biological complexes -with a special application to BlenX- and of various tools and GUIs to visualize spatial simulations.

Finally, some initial research on ways to design protocols for *in-silico experiments* is presented. During our research, we realized that even if simulation is very important for systems biology, there are a lot of different techniques that can be used; moreover, simulation is often only a step in a more complex set of operations necessary to investigate some problem. This thesis propose a method and programming practice to develop tools in a way that can be easily composed; each tool is encapsulated in a *component* or *service*, each representing a single step to be taken during an *in-silico* experiment. A composition of tools, realized in a graphical way thanks to existing technology borrowed from other fields of research, can be seen as a *protocol* reflecting a real wet-lab experimental protocol.

Acknowledgements

I would like to thank you all the people that, in one way or another, helped me during the years as a PhD student.

First of all, Alessandro Romanel, my room-mate from the beginning: during the first two years we worked together on a daily basis on BlenX and BetaWB. We shared ideas, discussed about problems, and also had a very good time working on tricky subjects and sharing a laugh. Without his help, my work and this thesis will simply not be here.

I want also to thank you my other PhD mates: Roberto Larcher, Alida Palmisano and Michele Forlin. Working with them enriched me both professionally and personally.

Thank you to Tommaso Mazza and Davide Prandi; they reviewed this thesis providing useful suggestions and corrections, making it much more enjoyable to read.

Last but not least, my supervisor, Corrado Priami. He was very motivating, assigning me challenging tasks but letting also do my own research on the subjects I liked, without forcing me to stick to a particular task. He also followed me carefully, giving useful advice when needed.

Bibliography

[1] P.S. Agutter, P.C. Malone, and D.N. Wheatley. Intracellular transport mechanisms: a critique of diffusion theory. *I. Theor. Biol.*, 176:261–272, 1995.

[2] P.S. Agutter and D.N. Wheatley. Random walks and cell size. *BioEssays*, 22:1018–1023, 2000.

[3] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular biology of the cell.* Garland Science, 2002.

[4] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular biology of the cell.* Garland Science, 4th ed. edition, 2003.

[5] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series).* Oxford University Press, 1977.

[6] Joshua A. Anderson and Alex Travesset. Molecular dynamics on graphic processing units: Hoomd to the rescue. *Computing in Science & Engineering*, 10(6), 2008.

[7] P. W. Anderson. More is different. *Science*, 177(4047):393, 1972.

[8] Steven S. Andrews and Dennis Bray. Stochastic simulation of chemical reactions with spatial resolution and single molecule detail. *Phys. Biol.*, 1(3-4):137–151, 2004.

[9] M.M. Babu, S.A. Teichmann, and L. Aravind. Evolutionary dynamics of prokaryotic transcriptional regulators. *J. Mol. Biol.*, (358):614–633, 2006.

[10] W. L. Bain and D. S. Scott. An algorithm for time synchronization in distributed discret event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 30–33, 1988.

[11] Paolo Ballarini, Rosita Guido, Tommaso Mazza, and Davide Prandi. Taming the complexity of biological pathways through parallel computing. *BRIEFINGS IN BIOINFORMATICS*, 10(3):278–288, 2009.

[12] Moritz Y. Becker. A graph layout algorithm for drawing metabolic pathways. *Bioinformatics*, 17(5), 2001.

[13] Marco Bernardo and Stefania Botta. A survey of modal logics characterising behavioural equivalences for non-deterministic and stochastic systems. *Mathematical Structures in Computer Science*, 18(1):29–55, 2008.

[14] D. Bernstein. Exact stochastic simulation of coupled chemical reactions. *PHYSICAL REVIEW E*, 71, April 2005.

[15] D. Bernstein. Exact stochastic simulation of coupled chemical reactions. *PHYSICAL REVIEW E*, 71, April 2005.

[16] L. Bononi, M. Bracuto, G. D'Angelo, and L. Donatiello. Concurrent replication of parallel and distributed simulation. In *Proceedings of the 19th ACM/IEEE/SCS PADS Workshop*, pages 430–436, 2005.

[17] Romain Bourqui, Ludovic Cottret, Vincent Lacroix, David Auber, Patrick Mary, Marie-France Sagot, and Fabien Jourdan. Metabolic network visualization eliminating node redundance and preserving metabolic pathways. *BMC Systems Biology*, 2007.

[18] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *Annals of Math. Statist*, 29(2):610–611, 1958.

[19] Franz J. Brandenburg, Michael Forster, Andreas Pick, Marcus Raitner, and Falk Schreiber. Biopath: Exploration and visualization of biochemical pathways. In M. Juenger and P. Mutzel, editors, *Graph Drawing Software*, pages 215–236. Springer Mathematics and Visualization Series, 2004.

[20] M. Bravetti and G. Zavattaro. Service oriented computing from a process algebraic perspective. *Journal of Logic and Algebraic Programming*, 70(1):3–14, 2007.

[21] BJ. Breitkreutz, C. Stark, and M. Tyers. Osprey: A network visualization system. *Genome Biology*, 4(3), 2003.

[22] K. Burrage, P. M. Burrage, N. Hamilton, and T. Tian. Computer-intensive simulations for cellular models. In *Parallel Computing in Bioinformatics and Computational Biology*, pages 79–119, 2006.

[23] Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. In *SIGGRAPH '93: Proceedings of the 20th annual conference on*

Computer graphics and interactive techniques, pages 263–270, New York, NY, USA, 1993. ACM.

[24] W. Cai and S. J. Turner. An algorithm for distributed discrete-event simulation - the 'carrier null message' approach. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 3–8, 1990.

[25] M. Calder, S. Gilmore, and J. Hillston. Automatically deriving odes from process algebra models of signalling pathways. In *Proc. Computational Methods in Systems Biology 2005*, pages 204–215, 2005.

[26] Muffy Calder, Stephen Gilmore, and Jane Hillston. Modelling the influence of RKIP on the ERK signalling pathway using the stochastic process algebra PEPA. In Anna Ingolfsdottir and Hanne Riis Nielson, editors, *Proceedings of the BioConcur Workshop on Concurrent Models in Molecular Biology*, London, England, August 2004.

[27] Y. Cao, D. Gillespie, and L. Petzold. Multiscale stochastic simulation algorithm with stochastic partial equilibrium assumption for chemically reacting systems. *J. Comp. Phys.*, 206(2):395–411, 2005.

[28] L. Cardelli. Brane calculi. In *CMSB*, pages 257–278, 2004.

[29] L. Cardelli. Brane calculi - interactions of biological membranes. In *Proceedings of Workshop on Computational Methods in Systems Biology (CMSB'04)*, volume 3082, pages 257–278. Lecture Notes in Computer Science, Springer, 2005.

[30] Luca Cardelli. On process rate semantics. *Theor. Comput. Sci.*, 391(3):190–215, 2008.

[31] Luca Cardelli. Artificial biochemistry. In Anne Condon, David Harel, Joost N. Kok, Arto Salomaa, and Erik Winfree, editors, *Algorithmic Bioprocesses*, pages 429–462. Springer Publishing Company, Incorporated, 2009.

[32] Luca Cardelli. Can a systems biologist fix a tamagotchi? In Yves Bertot, Grard Huet, Jean-Jacques Lvy, and Gordon Plotkin, editors, *From Semantics to Computer Science - Essays in Honour of Gilles Khan*, pages 501–512. Cambridge University Press, 2009.

[33] Luca Cardelli and Philippa Gardner. Processes in space. Technical Report Technical Report DTR09-4, Imperial College London, Department of Computing, 2009.

[34] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Electr. Notes Theor. Comput. Sci.*, 10, 1997.

[35] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *IEEE Trans. on Software Engineering*, SE-5(5):440–452, 1979.

[36] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *Comm. ACM*, 24(11):198–206, 1981.

[37] Min Chen, D. Ebert, H. Hagen, R.S. Laramee, R. van Liere, K.-L. Ma, G. Ribarsky, W. Scheuermann, and D. Silver. Data, information, and knowledge in visualization. *Computer Graphics and Applications, IEEE*, 29(1):12–19, 2009.

[38] Min Chen, David Ebert, Hans Hagen, Robert S. Laramee, Robert van Liere, Kwan-Liu Ma, William Ribarsky, Gerik Scheuermann, and Deborah Silver. Data, information, and knowledge in visualization. *IEEE Comput. Graph. Appl.*, 29(1):12–19, 2009.

[39] R. De Chiara, U. Erra, and V. Scarano. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In *Proceedings of the Vision, Modeling, and Visualization Conference*, pages 233–240, Stanford (California), USA, 2004.

[40] Federica Ciocchetta and Jane Hillston. Process algebras in systems biology. In *Formal Methods for Computational Systems Biology*, volume 5016, pages 265–312. Springer Berlin / Heidelberg, 2008.

[41] Federica Ciocchetta and Jane Hillston. Bio-pepa: A framework for the modelling and analysis of biological systems. *Theor. Comput. Sci.*, 410(33-34):3065–3084, 2009.

[42] Federica Ciocchetta, Corrado Priami, and Paola Quaglia. Modeling kohn interaction maps with beta-binders: An example. *T. Comp. Sys. Biology*, pages 33–48, 2005.

[43] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.

[44] R. Colwell. *Intel's P6 Microarchitecture*, chapter 7. McGraw-Hill Science, 2004.

[45] B. A. Cota and R. G. Sargent. A framework for automatic lookahead computation in conservative distributed simulations. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 56–59, 1990.

[46] Pierre-Louis Curien, Vincent Danos, Jean Krivine, and Min Zhang. Computational self-assembly. *Theor. Comput. Sci.*, 404(1-2):61–75, 2008.

[47] Michele Curti, Davide Prandi, and Linda Brodo. Formal executable descriptions of biological systems. In *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, page 2, Washington, DC, USA, 2005. IEEE Computer Society.

[48] Michele Curti, Davide Prandi, and Linda Brodo. Formal executable descriptions of biological systems. In *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, page 2, Washington, DC, USA, 2005. IEEE Computer Society.

[49] V. Danos and C. Laneve. Formal molecular biology. *TCS*, 2004.

[50] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theor. Comput. Sci.*, 325(1):69–110, 2004.

[51] D. De Roure, C. Goble, and R. Stevens. The design and realisation of the myexperiment virtual research environment for social sharing of workflows. *Future Generation Computer Systems*, 25:561–567, 2009.

[52] P. Degano, D. Prandi, C. Priami, and P. Quaglia. Beta-binders for biological quantitative experiments. In *4rd Int. Workshop on Quantitative Aspects of Programming Languages (QAPL 06)*, 2006. to appear.

[53] Lorenzo Dematté, Roberto Larcher, Alida Palmisano, Corrado Priami, and Alessandro Romanel. *Programming Biology in BlenX*. Springer, 2010.

[54] Lorenzo Dematté and Tommaso Mazza. On parallel stochastic simulation of diffusive systems. In *Proceedings of the sixth International Conference on Computational Methods in Systems Biology (CMSB2008)*, volume LNBI 5307, page 191210, 2008.

[55] Lorenzo Dematté, Corrado Priami, and Alessandro Romanel. Betawb: modelling and simulating biological processes. In *SCSC*, pages 777–784, 2007.

[56] Lorenzo Dematté, Corrado Priami, and Alessandro Romanel. The Beta Workbench: a computational tool to study the dynamics of biological systems. *Briefings in Bioinformatics*, 2008.

[57] Lorenzo Dematté, Corrado Priami, and Alessandro Romanel. The blenx language: A tutorial. In *SFM*, pages 313–365, 2008.

[58] Lorenzo Dematté, Corrado Priami, and Alessandro Romanel. Modelling and simulation of biological processes in BlenX. *SIGMETRICS Performance Evaluation Review*, 4(35):32–39, 2008.

[59] Lorenzo Dematté, Corrado Priami, Alessandro Romanel, and Orkun Soyer. A formal and integrated framework to simulate evolution of biological pathways. In *Proceedings of CMSB2007, LNBI 4695*, pages 106–120. Springer-Verlag, 2007.

[60] Lorenzo Dematté, Corrado Priami, Alessandro Romanel, and Orkun Soyer. Evolving blenx programs to simulate the evolution of biological networks. *Theor. Comput. Sci.*, 408(1):83–96, 2008.

[61] Cristian Dittamo and Davide Cangelosi. Optimized parallel implementation of gillespie's first reaction method on graphics processing units. In *International Conference on Computer Modeling and Simulation*, pages 156–161, Macau, China, 2009.

[62] Maciej Dobrzyński, Jordi Vidal Rodríguez, Jaap A. Kaandorp, and Joke G. Blom. Computational methods for diffusion-influenced biochemical reactions. *Bioinformatics*, 23(15):1969–1977, 2007.

[63] R. M. D'Souza, M. Lysenko, and K Rahmani. Sugarscape on steroids: simulating over a million agents at interactive rates. In *Agent2007*, Chicago, IL, 2007.

[64] Roshan M. D'Souza, Simeone Marino, and Denise Kirschner. Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. In *SpringSim'09*, San Diego, CA, 2009.

[65] J. Elf and M. Ehrenberg. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Syst. Biol.*, 1(2), December 2004.

[66] M. Elowitz, A. Levine, E. Siggia, and P. Swain. Stochastic gene expression in a single cell. *Science*, 297:1183–1186, 2002.

[67] Bjrn Engquist, Per Ltstedt, and Olof Runborg. *Multiscale Modeling and Simulation in Science*. Springer Publishing Company, Incorporated, 2009.

[68] R Ewald, J Himmelspach, M Jeschke, S Leye, and A. M. Uhrmacher. Flexible experimentation in the modeling and simulation framework james ii implications for computational systems biology. *Briefings in Bioinformatics*, 2010.

[69] G. C. Ewing, D. McNickle, and L. Pawlikowski. Multiple replications in parallel: Distributed generation of data for speeding up quantitative stochastic simulation. In

Proceedings of the 15th Congress of Int. Association for Matemathics and Computer in Simulation, pages 397–402, 1997.

[70] François Fages and Sylvain Soliman. Abstract interpretation and types for systems biology. *Theor. Comput. Sci.*, 403(1):52–70, 2008.

[71] Jordan Ferenc and Roberto Valentini. Cosbilab graph: the network analysis module of cosbilab. *Environmental Modelling & Software*, To Appear, 2010.

[72] Nvidia's next generation cuda compute architecture: Fermi. Technical report, NVIDIA Corporation, 2009.

[73] A Ferscha. *Parallel and Distributed Simulation of Discrete Event Systems*. McGraw-Hill, 1996.

[74] A. Fick. *Poggendorff's Annalen der Physik und Chemie*, 94:59–86, 1855.

[75] Jasmin Fisher and Thomas A Henzinger. Executable cell biology. *Nature Biotechnology*, 25:1239 – 1249, 2007.

[76] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley, New York, 1966.

[77] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[78] R. M. Fujimoto. Parallel discrete event simulation. *Comm. ACM*, 33(10):30–53, 1990.

[79] D. Fusco, N. Accornero, B. Lavoie, S. Shenoy, J. Blanchard, R. Singer, and E. Bertrand. Single mrna molecules demonstrate probabilistic movement in living mammallian cells. *Curr. Biol.*, 13:161–167, 2003.

[80] M.A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem.*, 104:1876–1889, 2000.

[81] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *The Journal of Computational Physics*, 22(4):403–434, 1976.

[82] Daniel T. Gillespie. The chemical langevin equation. *The Journal of Chemical Physics*, 113(1):297–306, 2000.

[83] Daniel T. Gillespie. Simulation methods in systems biology. In *SFM*, pages 125–167, 2008.

[84] D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, December 1977.

[85] L. Glass and S.A. Kauffman. The logical analysis of continuous, non-linear biochemical control networks. *J. Theor. Biol.*, 39:103129, 1973.

[86] P. W. Glynn and P. Heidelberger. Analysis of parallel replicated simulations under a completion time constraint. *ACM TOMACS*, 1(1):3–23, 1991.

[87] P. W. Glynn and P. Heidelberger. Analysis of initial transient deletion for parallel steady-state simulations. *SIAM J. Scientific Stat. Computing*, 13(4):904–922, 1992.

[88] P. W. Glynn and P. Heidelberger. Experiments with initial transient deletion for parallel, replicated steady-state simulations. *Management Science*, 38(3):400–418, 1992.

[89] Scott Le Grand. Broad-phase collision detection with cuda. In *GPU Gems 3*. Addison Wesley, 2007.

[90] Simon Green. Cuda particles. NVIDIA Whitepaper, 2008.

[91] B. Groselj and C. Tropper. The time-of-next-event algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 25–29, 1988.

[92] Helmut Grubmueller and Klaus Schulten. Special issue on advances in molecular dynamics simulations. *Journal of Structural Biology*, 157(3), March 2007.

[93] Maria Luisa Guerriero, Davide Prandi, Corrado Priami, and Paola Quaglia. Process calculi abstractions for biology. Technical Report TR-13-2006, The Microsoft Research - University of Trento Centre for Computational and Systems Biology, 2006.

[94] Maria Luisa Guerriero, Davide Prandi, Corrado Priami, and Paola Quaglia. Process calculi abstractions for biology. In Anne Condon, David Harel, Joost N. Kok, Arto Salomaa, and Erik Winfree, editors, *Algorithmic Bioprocesses*, pages 463–486. Springer Berlin Heidelberg, 2009.

[95] Maria Luisa Guerriero, Corrado Priami, and Alessandro Romanel. Modeling static biological compartments with beta-binders. In *AB*, pages 247–261, 2007.

[96] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Buehler, and Markus Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proceedings of Eurographics*, pages 303–312, 2005.

[97] S.E. Harding and P. Johnson. The concentration dependence of macromolecular parameters. *Biochemical Journal*, 231:543–547, 1985.

[98] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[99] Mark Harris. Mapping computational concepts to gpus. In Matt Pharr, editor, *GPU Gems 2*, pages 493–508. NVIDIA Corporation, 2005.

[100] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118, Saarbrucken, Germany, 2002.

[101] P. Heidelberger. Discrete event simulations and parallel processing: statistical properties. *SIAM Journal Stat. Comput.*, 9:1114–1132, 1988.

[102] Anders Helgeland and Oyvind Andreassen. Visualization of vector fields using seed lic and volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 10:673–682, 2004.

[103] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[104] Jan Himmelspach, Roland Ewald, and Adelinde M. Uhrmacher. A flexible and scalable experimentation layer. In *WSC '08: Proceedings of the 40th Conference on Winter Simulation*, pages 827–835. Winter Simulation Conference, 2008.

[105] C.A.R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, 1978.

[106] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, 1972.

[107] C. Y. Huang and J. E. Ferrell. Ultrasensitivity in the mitogen-activated protein kinase cascade. *Proc Natl Acad Sci U S A*, 93(19):10078–10083, September 1996.

[108] C. Y. Huang and J. E. Ferrell. Ultrasensitivity in the mitogen-activated protein kinase cascade. *Proc Natl Acad Sci U S A*, 93(19):10078–10083, September 1996.

[109] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. Doyle, and H. Kitano. The erato systems biology workbench: Enabling interaction and exchange between software tools for computational biology. In *In Proceedings of the Pacific Symposium on Biocomputing*, pages 450–461, 2002.

[110] Michael Hucka, Andrew Finney, and Herbert M. et al. Sauro. The systems biology markup language (sbml): A medium for representation and exchange of biochemical network models. *Bioinformatics*, 9(4):524531, 2003.

[111] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, , and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34:729–732, 2006.

[112] D. Hume. Probability in transcriptional regulation and its implications for leukocyte differentiation and inducible gene expression. *Blood*, 96:2323–2328, 2000.

[113] M. Hybinette and R. M. Fujimoto. Cloning parallel simulations. *ACM TOMACS*, 11(4):378–407, 2001.

[114] T. Ideker, T. Galitski, and L. Hood. A new approach to decoding life: Systems biology. *Annual Review of Genomics and Human Genetics*, 2(1):343372, 2001.

[115] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. Volume rendering techniques. In Randima Fernando, editor, *GPU Gems*. Addison-Wesley Professional, 2004.

[116] M. Januszewski and M. Kostur. Accelerating numerical solution of stochastic differential equations with cuda. *Computer Physics Communications*, 181:183–188, 2010.

[117] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Computer Systems*, 7(3):404–425, 1985.

[118] M Jeschke, R Ewald, A Park, R Fujimoto, and AM Uhrmacher. Parallel and distributed spatial simulation of chemical reactions. In *Proceedings of the 22nd ACM/IEEE/SCS PADS Workshop*, 2008.

[119] Mathias John, Roland Ewald, and Adelinde M. Uhrmacher. A spatial extension to the π calculus. *Electron. Notes Theor. Comput. Sci.*, 194(3):133–148, 2008.

[120] Mathias John, Roland Ewald, and Adelinde M. Uhrmacher. A spatial extension to the pi calculus. *Electron. Notes Theor. Comput. Sci.*, 194(3):133–148, 2008.

[121] Mathias John, Cédric Lhoussaine, Joachim Niehren, and Adelinde M. Uhrmacher. The attributed pi calculus. In *CMSB '08: Proceedings of the 6th International Conference on Computational Methods in Systems Biology*, pages 83–102, Berlin, Heidelberg, 2008. Springer-Verlag.

[122] S. E. Jorgensen. *Integration of Ecosystem Theories: A Pattern*. Kluwer Academic Publishers, 2002.

[123] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo methods. Vol. 1: basics*. Wiley-Interscience, 1986.

[124] E. R. Kandel. The molecular biology of memory storage: a dialogue between genes and synapses. *Science*, 294:1030–1038, 2001.

[125] S.A Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *J. Theor. Biol.*, 22:437467, 1969.

[126] M. W. Kirschner. The meaning of systems biology. *Cell*, 121(4):503504, May 2005.

[127] H. Kitano, editor. *Foundations of Systems B iology*. The MIT Press, 2001.

[128] Hiroaki Kitano. Systems biology: A brief overview. *Science*, 295(5560):1662–1664, March 2002.

[129] William J. Knottenbelt and Jeremy T. Bradley. Tackling large state spaces in performance modelling. In *SFM*, pages 318–370, 2007.

[130] M. Kwiatkowska and J. Heath. Biological pathways as communicating computer systems. *Journal of Cell Science*, 122(16):2793–2800, 2009.

[131] Marta Kwiatkowska, Gethin Norman, and David Parker. Using probabilistic model checking in systems biology. *SIGMETRICS Perform. Eval. Rev.*, 35(4):14–21, 2008.

[132] K.J. Laidler, J.H. Meiser, and B.C. Sanctuary. *Physical Chemistry*. Houghton Mifflin Company, 2003.

[133] Robert S. Laramee, Helwig Hauser, Helmut Doleisch, Benjamin Vrolijk, Frits H. Post, and Daniel Weiskopf. The state of the art in flow visualization: Dense and texture-based techniques. *Computer Graphics Forum*, 23(2), 2004.

[134] Roberto Larcher and Lorenzo Dematté. Custom visualization of biological structures: an application to blenx complexes. Technical Report TR-16-2009, CoSBi, 2009.

[135] Roberto Larcher, Adaoha Ihekwaba, and Corrado Priami. A betawb model for the nfkb pathway. Technical Report TR-25-2007, CoSBi, 2007.

[136] Roberto Larcher, Corrado Priami, and Alessandro Romanel. Modelling self-assembly in blenx. *Transactions on Computational Systems Biology*, To Appear, 2009.

[137] L. Laursen. Computational biology: Biological logic. *Nature*, 462:408–410, November 2009.

[138] Christophe Lavelle, Hugues Berry, Guillaume Beslon, Francesco Ginelli, Jean-Louis Giavitto, Zoi Kapola, Andr Le Bivic, Nadine Peyrieras, Ovidiu Radulescu, Adrien Six, Vronique Thomas-Vaslin, and Paul Bourgine. From molecules to organisms: towards multiscale integrated models of biological systems. *Theoretical Biology Insights*, 1(1):13–22, 2008.

[139] Y. Lazebnik. Can a biologist fix a radio? or, what i learned while studying apoptosis. *Cancer Cell*, 2:179–182, 2002.

[140] Paola Lecca and Lorenzo Dematté. Stochastic simulation of reaction-diffusion systems. *Int. Journal of Medical and Biological Engineering*, 1(4):211–231, December 2008.

[141] Paola Lecca, Lorenzo Dematté, Michela Lecca, and Corrado Priami. Stochastic modelling of diffusion systems. video image simulation of tubulin diffusion in cytoplasm: a case study. In *II Eccomas Thematic Conference on Computaional Vision and Medical Image Processing*, Porto, Portugal, October 2009.

[142] Paola Lecca, Lorenzo Dematté, and Corrado Priami. Modeling and simulating bio-molecule diffusion in non-homogeneous solutions. diffusive spatial effects on chaperone-assisted protein folding: a case study. Technical Report TR-16-2008, CoSBi, 2008.

[143] Paola Lecca, Adaoha Ihekwaba, Lorenzo Dematté, and Corrado Priami. Spatio-temporal dynamics of reaction diffusion systems: stochastic simulation of the bicoid gradient in drosophila embryo. Technical Report TR-5-2010, CoSBi, 2010.

[144] Hong Li and Linda Petzold. Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the graphics processing unit. *The International Journal of high Performance Computing Applications*, 00:1–10, 2009.

[145] W. Li and H. Kurata. A grid layout algorithm for automatic drawing of biochemical networks. *Bioinformatics*, 21(9):2036–42, May 2005.

[146] Y-B. Lin. Parallel independent replicated simulation on a network of workstations. *ACM SIGSIM Simulation Digest*, 24(1):73–80, 1994.

[147] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, pages 39–55, 2008.

[148] Weiguo Liu, Bertil Schmidt, Gerrit Vossa, and Wolfgang Mller-Wittiga. Accelerating molecular dynamics simulations using graphics processing units with cuda. *Computer Physics Communications*, 179(9):634–641, November 2008.

[149] Margaret L. Loper and Richard M. Fujimoto. Pre-sampling as an approach for exploiting temporal uncertainty. In *PADS 00*, pages 157–164, 2000.

[150] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[151] T. Mazza and R. Guido. Guidelines for parallel simulation of biological reactive systems. In *Proceedings of NETTAB 2008, Bioinformatics Methods for Biomedical Complex System Applications*, 2008. To appear.

[152] H.H. McAdams. It is a noisy business! genetic regulation at the nanomolar scale. *Trends Genet*, 15:65–69, 1999.

[153] Emanuela Merelli, Giuliano Armano, Nicola Cannata, Flavio Corradini, Mark d'Inverno, Andreas Doms, Phillip Lord, Andrew Martin, Luciano Milanesi, Steffen Mller, Michael Schroeder, and Michael Luck. Agents in bioinformatics, computational and systems biology. *Briefings in Bioinformatics*, pages 45–59, 2007.

[154] John Stuart Mill and Ernest Nagel. *John Stuart Mill's Philosophy of Scientific Method*, chapter Systems of Logics, Book III. Hafner Press, New York, 1950.

[155] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[156] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[157] Robin Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1980.

[158] M. E. J. Newman and G. T Barkema. *Monte Carlo Methods in Statistical Physics*. Oxford University Press, 2000.

[159] J. Nickolls, I. Buck, and M. Garland. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, 2008.

[160] D. M. Nicol. Parallel discrete-event simulation of fcfs stochastic queueing networks. *SIGPLAN Not.*, 23(9):124137, 1988.

[161] Denis Noble. *The Music of Life: Biology beyond the genome*. Oxford University Press, 2006.

[162] B. Novak, A Sveiczer, and J J Tyson. A stochastic, molecular model of the fission yeast cell cycle: role of the nucleocytoplasmic ratio in cycle time regulation. *Biophys. Chem*, 2001.

[163] P. Franois P and V. Hakim. Design of genetic networks with specified functions by evolution in silico. *Proc. Natl. Acad. Sci.*, 2(101):580–5, 2004.

[164] Alida Palmisano, Ivan Mura, and Corrado Priami. From odes to language-based, executable models of biological systems. In *Pacific Symposium on Biocomputing (PSB 2009)*, pages 239–250, January 2009.

[165] J. Pearson. Complex patterns in a simple system. *Science*, 261(5118):189–192, 1993.

[166] Kalyan S. Perumalla and Brandon G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on gpus. In *SpringSim '08: 2008 Spring simulation multiconference*, pages 116–123, Ottawa, Canada, 2008. Society for Computer Simulation International.

[167] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[168] Linda Petzold. Multiscale simulation algorithms for biochemical systems. Technical report, University of California Santa Barbara, 2007.

[169] T. Pfeiffer and S. Schuster. Game-theoretical approaches to studying the evolution of biochemical systems. *Trends Biochem Sci.*, 1(30):20–5, Jan 2005.

[170] T. Pfeiffer, O.S. Soyer, and S. Bonhoeffer. The evolution of connectivity in metabolic networks. *PLoS Biol.*, 7(3), 2005.

[171] Andrew Phillips and Luca Cardelli. A correct abstract machine for the stochastic pi-calculus. In *In Bioconcur04. ENTCS*, 2004.

[172] Andrew Phillips and Luca Cardelli. A graphical representation for the stochastic pi-calculus. In *Proceedings of Concurrent Models in Molecular Biology (Bioconcur'05)*, 2005.

[173] Andrew Phillips, Luca Cardelli, and Giuseppe Castagna. *A Graphical Representation for Biological Processes in the Stochastic pi-Calculus*. Number 4230 in LNCS. Springer, 2006.

[174] Victor Podlozhnyuk. Parallel mersenne twister. NVIDIA Whitepaper, 2007.

[175] A. Prakash and C. V. Ramamoorthy. Hierarchical distributed simulations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 341–348, 1988.

[176] D Prandi, C Priami, and P Quaglia. Shape spaces in formal interactions. *Complexus*, 2:128–139, 2005.

[177] Davide Prandi. A formal approach to molecular docking. In *CMSB*, pages 78–92, 2006.

[178] Davide Prandi and Lorenzo Dematté. Gpu computing for systems biology. *BRIEFINGS IN BIOINFORMATICS*, To Appear, 2010.

[179] Davide Prandi, Corrado Priami, and Paola Quaglia. Communicating by compatibility. *J. Log. Algebr. Program.*, 75(2):167–181, 2008.

[180] C. Priami and P. Quaglia. Beta binders for biological interactions. In *CMSB*, pages 20–33, 2004.

[181] C. Priami and P. Quaglia. Operational patterns in beta-binders. *T. Comp. Sys. Biology*, 1:50–65, 2005.

[182] C. Priami, A. Regev, W. Silverman, and E. Shapiro. Application of a stochastic name passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80, 2001.

[183] C. Priami and A. Romanel. The decidability of the structural congruence for beta-binders. In *MeCBIC 2006*, 2006.

[184] Corrado Priami. Algorithmic systems biology. *Communications of the ACM*, 52(5):8088, 2009.

[185] Corrado Priami, Paola Quaglia, and Alessandro Romanel. Blenx - static and dynamic semantics. In *Proceedings of 20th International Conference on Concurrency Theory (CONCUR)*. Springer-Verlag, 2009.

[186] D. Rantzau, K. Frank, U. Lang, D. Rainer, and U. Woessner. Covise in the cube: An environment for analyzing large and complex simulation data. In *Proc. 2nd Workshop on Immersive Projection Technology (IPTW '98)*, Ames, Iowa, 1998.

[187] A. Regev, E. Panina, W. Silverman, E. Shapiro, and L. Cardelli. Bioambients: an abstraction for biological compartments. *Theoretical computer science*, 2003.

[188] A. Regev, E.M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.

[189] A. Regev and E. Shapiro. Cells as computation. *Nature*, 2002.

[190] A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. *Pac Symp Biocomput*, pages 459–470, 2001.

[191] A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Proceedings of the Pacific Symposium of Biocomputing 2001 (PSB2001)*, pages 459–470, June 2001.

[192] Paul Richmond, Simon Coakley, and Daniela M. Romano. A high performance agent based modelling framework on graphics card hardware with cuda. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 1125–1126., Budapest, Hungary, 2009.

[193] A.M. Ridwan, A. Krishnan, and P. Dhar. A parallel implementation of gillespies direct method. *Computational Science - ICCS 2004*, 3037/2004:284–291, 2004.

[194] Elijah Roberts, John E.Stone, Leonardo Sepulveda, Wen-Mei W. Hwu, and Zaida Luthey-Schulten. Long time-scale simulations of in vivo diffusion using gpu hardware. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, 2009.

[195] Mathias Röhl and Adelinde M. Uhrmacher. Composing simulations from xml-specified model components. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 1083–1090. Winter Simulation Conference, 2006.

[196] A. Romanel and C. Priami. On the decidability and complexity of the structural congruence for beta-binders. *Theor. Comput. Sci.*, 404(1-2):156–169, 2008.

[197] Alessandro Romanel. *Dynamic Biological Modelling: a language-based approach*. PhD thesis, University of Trento - DISI / Dipartimento di Ingegneria e Scienza dell'Informazione, 2010.

[198] I. Ross, C. Browne, and D. Hume. Transcription of individual genes in eukaryotic cells occurs randomly and infrequently. *Immunol Cell Biol*, 1994.

[199] M. Rukoz. Hierarchical deadlock detection for nested transactions. *Distributed Computing*, 4:123–129, 1991.

[200] Stefan Rybacki, Jan Himmelspach, and Adelinde M Uhrmacher. Experiments with single core, multi-core, and gpu based computation of cellular automata. In *SIMUL '09. First International Conference on Advances in System Simulation*, pages 62–67, Porto, 2009.

[201] Uwe Sauer, Matthias Heinemann, and Nicola Zamboni. Getting closer to the whole picture. *Science*, 316(5824):550–551, 2007.

[202] Simon Scarl. Implications of the turing completeness of reaction-diffusion models, informed by gpgpu simulations on an xbox 360: Cardiac arrhythmias, re-entry and the halting problem. *Computational Biology and Chemistry*, 33(4):253–260, August 2009.

[203] Henning Scharsach, Markus Hadwiger, André Neubauer, Stefan Wolfsberger, and Katja Buehler. Perspective isosurface and direct volume rendering for virtual endoscopy applications. In *Proceedings of Eurovis/IEEE-VGTC Symposium on Visualization*, pages 315–322, 2006.

[204] Rinaldo B. Schinazi. Predator-prey and host-parasite spatial stochastic models. *The Annals of Applied Probability*, 7(1):1–9, 1997.

[205] J. Schulze-Doebold, U. Woessner, S.P. Walz, and U. Lang. Volume rendering in a virtual environment. In *Proceedings of 5th IPTW and Eurographics Virtual Environments*, pages 187–198. Springer Verlag, 2001.

[206] M. Schwehm. Parallel stochastic simulation of whole-cell models. In *Proceedings of ICSB*, pages 333–341, 2001.

[207] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[208] R. Sharan, S. Suthram, R. M. Kelley, T. Kuhn, S. McCuine, P. Uetz, T. Sittler, R. M. Karp, and T. Ideker. Conserved patterns of protein interaction in multiple species. *Proc Natl Acad Sci*, 6(102):1974–79, 2005.

[209] T.S. Shimizu, S. V. Aksenov, and D. Bray. A spatially extended stochastic model of the bacterial chemotaxis signalling pathway. *Journal of Molecular Biology*, 329:291–309, 2003.

[210] Yogesh Simmhan, Roger Barga, Catharine van Ingen, Ed Lazowska, and Alex Szalay. Building the trident scientific workflow workbench for data management in the cloud. In *International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP)*. IEEE, October 2009.

[211] A. Solovyova, P. Schuck, L. Costenaro, and C. Ebel. Non ideality of sedimantation velocity of halophilic malate dehydrogenase in complex solvent. *Biophysical Journal*, 81:1868–1880, 2001.

[212] Orkun Soyer and Sebastian Bonhoeffer. Evolution of complexity in signaling pathways. *PNAS*, (103):16337–16342, 2006.

[213] J. Spudich and DEJ Koshland. Non-genetic individuality: Chance in the single cell. *Nature*, 1976.

[214] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proceedings of Volume Graphics 2005*, pages 187–195, New York, USA, 2005.

[215] Ann M. Stock, Victoria L. Robinson, and Paul N. Goudreau. Two-component signal transduction. *Annu. Rev. Biochem*, (69):183–215, 2000.

[216] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, (28):2618–2640, 2007.

[217] S. Streltsov and P. Vakili. Parallel replicated simulation of markov chains: implementation and variance reduction. In *Proceedings of the 25th conference on Winter simulation*, pages 430–436, 1993.

[218] M. Sugimoto, K.Takahashi, T. Kitayama, D. Ito, and M. Tomita. Distributed cell biology simulations with E-Cell system. In *First international workshop on life science grid (LSGRID04)*, 2004.

[219] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[220] K. Takahashi, S.N.V. Arjunan, and M. Tomita. Space in systems biology of signaling pathways - towards intracellular molecular crowding in silico. *FEBS Letters*, 579(8):1783–1788, 2005.

[221] Koichi Takahashi. An exact brownian dynamics method for cell simulation. In *Computational Methods in Systems Biology*, volume 5307/2008, pages 5–6. Springer Berlin / Heidelberg, 2008.

[222] Kouichi Takahashi, Kazunari Kaizu, Bin Hu, and Masaru Tomita. A multi-algorithm, multi-timescale method for cell simulation. *Bioinformatics*, 20(4):538–546, March 2004.

[223] A. Theocharidis, S. van Dongen, A.J. Enright, and T.C. Freeman. Network visualisation and analysis of gene expression data using biolayout express3d. *Nature Protocols*, 4(10):1535–50, 2009.

[224] T. Tian and K. Burrage. Parallel implementation of stochastic simulation for large-scale cellular processes. In *Proceedings of of Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, pages 621–626, 2005.

[225] M. P. Tombs and A. R. Peacocke. *The Osmotic Pressure of Biological Macromolecules*. Monograph on Physical Biochemistry, Oxford University Press, 1975.

[226] Adelinde Uhrmacher, Daniela Degenring, and Bernard P. Zeigler. Discrete event multi-level models for systems biology. *T. Comp. Sys. Biology*, 1:66–89, 2005.

[227] Adelinde M. Uhrmacher, Roland Ewald, Mathias John, Carsten Maus, Matthias Jeschke, and Susanne Biermann. Combining micro and macro-modeling in devs for computational biology. In *WSC '07: Proceedings of the 39th conference on Winter simulation*, pages 871–880, Piscataway, NJ, USA, 2007. IEEE Press.

[228] J. S. van Zon and P. R. Ten Wolde. Green's-function reaction dynamics: A particle-based approach for simulating biochemical networks in time and space. *J Chem Phys*, 123(23), December 2005.

[229] Joachim Weickert and Hans Hagen, editors. *Visualization and Processing of Tensor Fields*. Springer, 2005.

[230] K. R. Wood and S. J. Turner. A generalized carrier-null method for conservative parallel simulation. In *Proceedings of the 8th PADS Workshop*, pages 50–57, 1994.

[231] V. Yau. Automating parallel simulation using parallel time streams. *ACM TOMACS*, 9(2):171–201, 1999.

Appendix A

Computed diffusion coefficients

A.1 The model of diffusion

If solutions of different concentrations are brought into contact with each other, the solute molecules tend to flow from regions of higher concentration to regions of lower concentration, and there is ultimately an equalisation of concentration. The driving force of the diffusion is the Gibbs energy difference between regions of different concentration, i. e. the gradient of chemical potential μ. Consider a solution containing N different solutes. The chemical potential μ_i of any particular chemical species i is defined as the partial derivative of the Gibbs energy G with respect to the concentration of the species i, with temperature and pressure held constant. Species are in equilibrium if their chemical potentials are equal.

$$\mu_i \equiv \frac{\partial G}{\partial c_i} = \mu_i^0 + RT \ln a_i \tag{A.1}$$

where c_i is the concentration of the species i, μ_i^0 is the standard chemical potential of the species i (i .e. the Gibbs energy of 1 mol of i at a pressure of 1 bar), $R = 8.314$ J $\cdot$ K^{-1} $\cdot$ mol^{-1} is the ideal gas constant, and T the absolute temperature. The quantity a_i is called *chemical activity* of component i, and it is given by

$$a_i = \frac{\gamma_i c_i}{c^0} \tag{A.2}$$

where γ_i is the *activity coefficient*, c^0 being a reference concentration. The activity coefficients express a deviation of a solution from the ideal thermodynamic behaviour and in general they may depend on the concentration of all the solutes in the system. For ideal solution, a limit which is recovered experimentally at high dilutions, $\gamma_i = 1$. If the concentrations of species i varies from point to point in space, then so does the chemical

potential. For simplicity, we treat here the case in which there is only a chemical potential gradient in the x direction only. Chemical potential is the free energy per mole of substance i, free energy is the negative of the work W which a system can perform, and work is connected to force F by $dW = Fdx$. Therefore an inhomogeneous chemical potential is related to a virtual force per molecule of

$$F_i = -\frac{1}{N_A}\frac{d\mu_i}{dx} = -\frac{k_B T c^0}{\gamma_i c_i}\sum_j \frac{\partial a_i}{\partial c_j}\frac{\partial c_j}{\partial x} \tag{A.3}$$

where $N_A = 6.022 \times 10^{23}$ mol^{-1} is the Avogadro's number, $k_B = 1.381 \times 10^{-23}$ J $\cdot$ K^{-1} is the Boltzmann's constant, and the sum is taken over all species in the system other than the solvent. This force is balanced by the drag force experienced by the solute ($F_{drag,i}$) as it moves through the solvent. Drag forces are proportional to the speed. If the speed of the solute is not too high in such a way the solvent does not exhibit turbulence, we can assume that the drag force is

$$F_{drag,i} = f_i v_i \tag{A.4}$$

where $f_i \propto c_i$ is the frictional coefficient, and v_i is the mean drift speed.

Again, if the solvent is not turbulent, we can assume that the *flux*, defined as the number of moles of solute which pass through a small surface per unit time per unit area, is

$$J_i = c_i v_i \tag{A.5}$$

i. e. the number of molecules per unit volume multiplied by the linear distance travelled per unit time.

Since the virtual force on the solute is balanced by the drag force (i. e. $F_{drag,i} = -F_i$), we obtain the following expression for the mean drift velocity

$$v_i = \frac{F_i}{f_i}$$

so that Eq. (A.5) becomes

$$J_i = -\frac{k_B T}{\gamma_i f_i}\sum_j \frac{\partial a_i}{\partial c_j}\frac{\partial c_j}{\partial x} \equiv -\sum_j D_{ij}\frac{\partial c_j}{\partial x} \tag{A.6}$$

where

$$D_{ij} = \frac{k_B T c^0}{\gamma_i f_i}\frac{\partial a_i}{\partial c_j} \tag{A.7}$$

are the diffusion coefficients. The Eq. (A.7) states that, in general, the flux of one species depends on the gradients of all the others, and not only on its own gradient. However, here we will suppose that the chemical activity a_i depends only weakly on the concentrations of the other solutes, i. e. we assume that $D_{ij} \approx 0$ for $i \neq j$ and the Fick's laws still holds. Let D_i denote D_{ii}. It is still generally the case that D_i depends on c_i in sufficiently concentrated solutions since γ_i (and thus a_i) has a non trivial dependence on c_i. In order to find an analytic expression of the diffusion coefficients D_i in terms of the concentration c_i, let us consider that the rate of change of concentration of the substance i due to diffusion is given by

$$D_i = -\frac{\partial J_i}{\partial x} \tag{A.8}$$

Substituting Eq. (A.7) into Eq. (A.6), and then substituting the obtained expression for J_i into Eq. (A.8), gives

$$D_i = -\frac{\partial}{\partial x}\left(- D_i(c_i)\frac{\partial c_i}{\partial x}\right) \tag{A.9}$$

so that

$$D_i = \left(\frac{\partial D_i(c_i)}{\partial x}\right)\frac{\partial c_i}{\partial x} + D_i(c_i)\frac{\partial^2 c_i}{\partial x^2} = \frac{\partial D_i(c_i)}{\partial c_j}\frac{\partial c_j}{\partial x}\frac{\partial c_i}{\partial x} + D_i(c_i)\frac{\partial^2 c_i}{\partial x^2} \tag{A.10}$$

Let $c_{i,k}$ denote the concentration of a substance i st coordinate x_k, and $l = x_k - x_{k-1}$ the distance between adjacent mesh points. The derivative of c_i with respect to x calculate in $x_{k-\frac{1}{2}}$ is

$$\left.\frac{\partial c_i}{\partial x}\right|_{x_{k-\frac{1}{2}}} \approx \frac{c_{i,k} - c_{i,k-1}}{l} \tag{A.11}$$

By using Eq. (A.11) into Eq. (A.6) the diffusive flux of species i midway between the mesh points $J_{i,k-\frac{1}{2}}$ is obtained

$$J_{i,k-\frac{1}{2}} = -D_{i,k-\frac{1}{2}}\frac{c_{i,k} - c_{i,k-1}}{l} \tag{A.12}$$

where $D_{i,k-\frac{1}{2}}$ is the estimate of the diffusion coefficient midway between the mesh points.

The rate of diffusion of substance i at the mesh point k is

$$D_{ik} = -\frac{J_{i,k+\frac{1}{2}} - J_{i,k-\frac{1}{2}}}{l}$$

and thence

$$D_{ik} = \frac{D_{i,k-\frac{1}{2}}}{l^2}(c_{i,k-1} - c_{i,k}) - \frac{D_{i,k+\frac{1}{2}}}{l^2}(c_{i,k+1} - c_{i,k}) \tag{A.13}$$

To determine completely the right-hand side of Eq. (A.13) is now necessary to find an expression for the activity coefficient γ_i and the frictional coefficient f_i, contained in the formula (A.7) for the diffusion coefficient. In fact, by substituting Eq. (A.2) into Eq. (A.7) we obtain an expression of the diffusion coefficient in terms of activity coefficients γ_i

$$D_{ii} = \frac{k_B T}{f_i}\Big(1 + \frac{c_i}{\gamma_i}\frac{\partial \gamma_i}{\partial c_i}\Big) \tag{A.14}$$

Let focus now on the calculation of the activity coefficients, while a way to estimate the frictional coefficients will be present in Section A.1.1. By using the subscript '1' to denote the solvent and '2' to denote the solute, we have

$$\mu_2 = \mu_2^0 + RT \ln\left(\frac{\gamma_2 c_2}{c^0}\right) \tag{A.15}$$

where γ_2 is the activity coefficient of the solute and c_2 is the concentration of the solute. By differentiating with respect to c_2 we obtain

$$\frac{\partial \mu_2}{\partial c_2} = RT\Big(\frac{1}{c_2} + \frac{1}{\gamma}\frac{\partial \gamma_2}{\partial c_2}\Big) \tag{A.16}$$

The chemical potential of the solvent is related to the osmotic pressure (Π) by

$$\mu_1 = \mu_1^0 - \Pi V_1 \tag{A.17}$$

where V_1 is the partial molar volume of the solvent and μ_1^0 its standard chemical potential. Assuming V_1 to be constant and differentiating μ_1 with respect to c_2 we obtain

$$\frac{\partial \mu_1}{\partial c_2} = -V_1 \frac{\partial \Pi}{\partial c_2} \tag{A.18}$$

Now, from the Gibbs-Duhen relation, the derivative of the chemical potential of the solute with respect to the solute concentration is

$$\frac{\partial \mu_2}{\partial c_2} = -\frac{M(1 - c_2\overline{v})}{V_1 c_2}\frac{\partial \mu_1}{\partial c_2} = \frac{M(1 - c_2\overline{v})}{c_2}\frac{\partial \Pi}{\partial c_2} \tag{A.19}$$

where M is molecular weight of the solute and $\overline{v}$ is the partial molar volume of the solute divided by its molecular weight. The concentration dependence of osmotic pressure is usually written as

$$\frac{\Pi}{c_2} = \frac{RT}{M}\Big[1 + BMc_2 + O(c_2^2)\Big] \tag{A.20}$$

where B is the second virial coefficient (see Section A.1.2), and thence the derivative with respect to the solute concentration is

$$\frac{\partial \Pi}{\partial c_2} = \frac{RT}{M} + 2RTBc_2 + O(c_2^2) \tag{A.21}$$

Introducing Eq. (A.21) into Eq. (A.19) gives

$$\frac{\partial \mu_2}{\partial c_2} = RT(1 - c_2\overline{v})\Big(\frac{1}{c_2} + 2BM\Big) \tag{A.22}$$

From Eq. (A.16)and Eq. (A.22) we have

$$\frac{1}{\gamma_2}\frac{\partial \gamma_2}{\partial c_2} = \frac{1}{c_2}\Big[RT(1 - c_2\overline{v})(1 + 2BMc_2) - 1\Big]$$

so that

$$\int_1^{\gamma_2'} \frac{d\gamma_2}{\gamma_2} = \int_{c^0}^{c_2'} \frac{1}{c_2}\Big[RT(1 - c_2\overline{v})(1 + 2BMc_2) - 1\Big]dc_2$$

On the grounds that $c_2\overline{v} \ll 1$ [225], by solving the integrals we obtain

$$\gamma_2' = \exp[2BMRT(c_2' - c^0)] \tag{A.23}$$

The molecular weight $M_{i,k}$ of the species i in the mesh k can be expressed as the ratio between the mass $m_{i,k}$ of the species i in that mesh and the Avogadro's number $M_{i,k} = m_{i,k}/N_A$. If p_i is the mass of a molecule of species i and $c_{i,k}l$ is the number of molecules of species i in the mesh k, then the molecular weight of the solute of species i in the mesh k is given by

$$M_{i,k} = \frac{p_i \;\; l}{N_A} c_{i,k} \tag{A.24}$$

Substituting this expression in Eq. (A.23) we obtain for the activity coefficient of the solute of species i in the mesh k ($\gamma_{i,k}$), the following equation

$$\gamma_{i,k} = \exp\Big(2B\;\frac{p_i\;l}{N_A}\;c_{i,k}^2\Big) \tag{A.25}$$

A.1.1 Intrinsic viscosity and frictional coefficient

The diffusion coefficient depends on the ease with which the solute molecules can move. The diffusion coefficient of a solute is a measure of how readily a solute molecule can push aside its neighboring molecules of solvent. An important aspect of the theory of diffusion is how the magnitudes of the frictional coefficient f_i of a solute of species i and, hence, of the diffusion coefficient D_i, depend on the properties of the solute and solvent molecules. Examination of well-established experimental data shows that diffusion coefficients tend to decrease as the molecular size of the solute increases. The reason is that a larger solute molecule has to push aside more solvent molecules during its progress and will therefore move slowly than a smaller molecule. A precise theory of the frictional coefficients for the diffusion phenomena in biological context cannot be simply derived from the elementary assumption and model of the kinetic theory of gases and liquids. The Stokes's theory considers a simple situation in which the solute molecules are so much larger than the solvent molecules that the latter can be regarded as a continuum (i. e. not having molecular character). For such a system Stokes deduced that the frictional coefficient of the solute molecules is $f_i = 6\pi r_i^H \eta$, where r_i^H is the hydrodynamical radius of the molecule and η is the viscosity of the solvent. For proteins diffusing in the cytosol, the estimate of frictional coefficient through the Stokes's law is hard, for several reasons. First of all, the assumption of very large spherical molecules in a continuous solvent is not a realistic approximation for a protein moving through the cytosol: the protein may be not spherical and the solvent is not a continuum. Furthermore, in the protein-protein interaction in the cytosol water molecules should be included explicitly, thus complicating the estimation of the hydrodynamical radius. Finally, the viscosity of the solvent η within the cellular environment cannot be approximated either as the viscosity of liquid or the viscosity of gas. In both cases, the theory predict a strong dependence on the temperature of the system, that has not been found in the cell system, where the most significant factor in determining the behavior of frictional coefficient is the concentration of solute molecules. To model the effects of non-ideally on the friction coefficient we assume that its dependence on the concentration of the solute is governed by expression similar to the one used to model friction coefficient in sedimentation processes [211]

$$f_{i,k} = k_f c_{i,k} \tag{A.26}$$

where k_f is an empirical constant, whose value can be derived from the knowledge of the ratio $R = k_f/[\eta]$. Accordingly to the Mark-Houwink equation, $[\eta] = kM^\alpha$ is the intrinsic viscosity coefficient, α is related to the shape of the molecules of the solvent, and M is still the molecular weight of the solute. If the molecules are spherical, the intrinsic viscosity is

independent of the size of the molecules, so that $\alpha = 0$. All globular proteins, regardless of their size, have essentially the same $[\eta]$. If a protein is elongated, its molecules are more effective in increasing the viscosity and $[\eta]$ is larger. Values of 1.3 or higher are frequently obtained for molecules that exist in solution as extended chains. Long-chain molecules that are coiled in solution give intermediate values of α, frequently in the range from 0.6 to 0.75 [132]. For globular macromolecule, R has a value in the range of 1.4 - 1.7, with lower values for more asymmetric particles [97].

A.1.2 Calculated second virial coefficient

The mechanical statistical definition of the second virial coefficient is given by the following

$$B = -2\pi N_A \int_0^\infty r^2 \exp\Big[-\frac{u(r)}{k_B T}\Big] dr \tag{A.27}$$

where $u(r)$ is the interaction free energy between two molecules and r is the intermolecular center-center distance. In this work we assume for $u(r)$ the Lennard-Jones pair (12,6)-potential (Eq. A.28), that captures the attractive nature of the Van der Waals interactions and the very short-range Born repulsion due to the overlap of the electron clouds.

$$u(r) = 4\Big[\Big(\frac{1}{r}\Big)^{12} - \Big(\frac{1}{r}\Big)^{6}\Big] \tag{A.28}$$

and expanding the term $\exp\left(\frac{4}{k_B T}\frac{1}{r^6}\right)$ into an infinite series, the Eq. (A.27) becomes

$$B = -2\pi N_A \sum_{j=0}^{\infty} \frac{1}{j!}(T^*)^j \int_0^\infty r^{2-6j} \exp\Big[-T^*\frac{1}{r^2}\Big] dr$$

where $T^* \equiv 4/(k_B T)$ and thus

$$B = -\frac{\pi N_A}{6} \sum_{j=0}^{\infty} \frac{1}{!j} 4^j (k_B T)^{-\frac{1}{4}+\frac{1}{2}j}\ \Gamma\Big(-\frac{1}{4}+\frac{1}{2}j\Big) \tag{A.29}$$

In our model the estimate of B is given by truncating the infinite expansion of the Γ function to $j = 4$. Taking into account the additional terms for $j > 4$ does not influence the simulation results in a significant way.

Appendix B

BlenX language reference

A BlenX program is made of an optional *declaration* file for the declaration of user-defined constants and functions, a *binder definition* file that associates unique identifiers to binders of entities used by the program and a *program* file, that contains the program structure.

All the BlenX files share the syntax definition of identifiers, numbers and rates as reported below:

$$
\begin{array}{lcl}
Letter & ::= & [a-zA-Z] \\
Digit & ::= & [0-9] \\
Exp & ::= & [Ee][+]?\{Digit\} \\
real1 & ::= & \{Digit\}^{+}\{Exp\} \\
real2 & ::= & \{Digit\}^{*}\text{"."}Digit^{+}(\{Exp\})? \\
real3 & ::= & \{Digit\}^{+}\text{"."}Digit^{*}(\{Exp\})?
\end{array}
$$

$$
\begin{array}{lcl}
Real & ::= & real1 \mid real2 \mid real3 \\
Decimal & ::= & \{Digit\}^{+} \\
Id & ::= & (\{Letter\}|\_)(\{Letter\}|\{Digit\}|\_)^{*}
\end{array}
$$

$$
number \quad := \quad Real \mid Decimal
$$

$$
rate \quad := \quad number \mid \textbf{rate} \ (\ Id\ ) \mid \textbf{inf}
$$

Note that in the following sections, during the description of the programming constructs, we prefix qualifying words to Id in order to clarify the kind of identifier that can occur in a given position. We will write $boxId$, $binderId$, $funcId$ and $varId$ to specify identifiers referring to boxes, binders, functions and variables respectively. Syntactically, they are all equal to Id; the disambiguation is done by the BlenX compiler using a symbol table. For examples, if an identifier Id is used in a function declaration, it will be stored as a $funcId$ in the symbol table.

B.1 The declaration file

A declaration file is a file with *.func* or *.decl* extension that contains the definition of variables, constants and functions. Since these constructs are optional, it is possible to skip the definition of the whole file. The declaration file has the following syntax:

declarations ::=
    decList

decList ::=
    dec
  | *dec decList*

dec ::=
    let *Id* : **function** = *exp* ;
  | **let** *Id* : **var** = *exp* ;
  | **let** *Id* : **var** = *exp* **init** *number*;
  | **let** *Id* (*number*) : **var** = *exp* ;
  | **let** *Id* : **const** = *exp* ;

exp ::=
    number
  | *Id*
  | | *Id* |
  | **log** (*exp*)
  | **sqrt** (*exp*)
  | **exp** (*exp*)
  | **pow** (*exp* , *exp*)
  | *exp* + *exp*
  | *exp* − *exp*
  | *exp* × *exp*
  | *exp* / *exp*
  | −*exp*
  | +*exp*
  | (*exp*)

An *expression* is made up of operators and operands. The syntax for the expression *exp* and the possible algebraic operators that can be used is given in the previous table. Operator precedence follows the common rules found in every programming language. + and − have the precedence when used as unary operators, while × and / have the precedence w.r.t. + and − when used as binary operators.

A *state variable* or simply *variable* is an identifier that can assume real modifiable val-

ues (Real value). The content of a variable is automatically updated when the defining expression *exp* changes; The content of the variable can also be changed by an **update** event (see Sec. B.6). In this case, the function associated with the event is evaluated and the variable is updated with the resulting value. After the variable identifier and the **var** keyword, the user has to specify the expression used to control the value of the variable and an optional initial value after the **init** keyword. Examples of variable declarations follows:

```
let v1 : var = 10 * |A|;
let mCycB : var = 2 * |X| * log(v1) init 0.1;
```

In addition, we define another type of variables, called *continuous variables*. A continuous variable x represent a differential equation on x. The equation is discretised and integrated over time; the variable value is evaluated at fixed time steps Δt, specified by the user. In a *continuous variable* declaration the user has to specify the Id of the variable, immediately followed by the Δt value:

```
let x(delta_t): var = expr;
```

Where `delta_t` is a real number. An optional initial value can be specified with the `init` keyword. For example:

```
let x(0.1): var = x * (1/y) init 0.2;
```

The expression after the = sign is used to compute the delta value, with Δt implicit. Therefore, the declaration `let x(delta_t): var = expr;` corresponds to the differential equation

$$\frac{\delta x}{\delta t} = expr$$

. Every Δt, the simulator updates the variable value using the following formula:

$$x_{t(i)} = x_{t(i-1)} + expr \cdot \Delta t$$

.

A *constant* is an identifier that assumes a value that cannot be changed at run-time and specified through a constant expression (an expression that does not rely on any variable or concentration $|Id|$ to be evaluated). As an extension, BlenX allows the use of *constant expressions*. Examples of constant declarations and of constant expressions follow:

```
let c1 : const = 1.0;
let pi : const = 3.14;
let c2 : const = (2.5 + 1) / (2.5 - 1);
let c3 : const = (4.0/3.0) * pi * pow(c1, 3);
let e: const = exp(1.0);
```

In the current version of BlenX, *functions* are parameterless and always return a Real value. As is, a function is only a named expression that can be used to evaluate a rate or to update the content of a state variable. An example of function definition follows:

```
let f1 : function =
   (k5s / alpha) / (pow( (J5 / (m * alpha * |X|) ) , 4) + 1);
```

Notice that when a program contains continuous variables, then the CTMC generation is not allowed.

B.2 The binder definition file

The binder definition file is a file with *.types* extension that stores all the binder identifiers that can be used in the declaration of binders (see Sec. B.4) and the affinities between binders associated with a particular identifier.

Affinities are a peculiar feature of BlenX. The interaction mechanism of many biological modelling languages is based on the notion of exact complement of communication channel names, as in computer science modelling where two programs can interact only if they know the exact address of the interacting partners. In BlenX instead interactions are guided by affinities between a pair of binder identifiers. There are three advantages in this approach: it allows us to avoid any global policy on the usage of names in order to make components interact; it relaxes the exact, or *key-lock*, style of interaction of exact name pairing; it permits a better separation of concerns, as it allows us to put interaction information in a separate file that can be modified or substituted without altering the program. The usage of affinities in a separate file is comparable to program interactions guided by *contracts* or service definitions, like in some web-service models (see [20]).

$$
\begin{array}{lll}
\textit{affinities} & ::= & \\
 & & \{\ \textit{binderIdList}\ \} \\
 & | & \{\ \textit{binderIdList}\ \}\%\%\{\ \textit{affinityList}\ \} \\
\textit{binderIdList} & : & \\
 & & \textit{binderId} \\
 & | & \textit{binderId},\ \textit{binderIdList} \\
 & & \\
\textit{affinity} & ::= & \\
 & & (\ \textit{binderId},\ \textit{binderId},\ \textit{rate}\ ) \\
 & | & (\ \textit{binderId},\ \textit{binderId},\ \textit{funcId}\ ) \\
 & | & (\ \textit{binderId},\ \textit{binderId},\ \textit{rate},\ \textit{rate},\ \textit{rate}\ ) \\
\textit{affinityList} & := & \\
 & & \textit{affinity} \\
 & | & \textit{affinity},\ \textit{affinityList}
\end{array}
$$

An *affinity* is a tuple of three or five elements. The first two elements are binder identifiers declared in the *binderIdList*, while the other elements can either be rate values or a single function identifier. If the affinity tuple contains a single rate value, then the value is interpreted as the base rate of *inter-communication* (Sec. B.4.1) between binders with identifier equal to the first and second $binderId$ respectively.
If the affinity tuple contains three rate values, these values are interpreted as the base rate for *complex*, *decomplex* (see Sec. B.5 for the definition of complexes) and *inter-complex communication* between binders with identifier equal to the first and second $binderId$ respectively.
When the element after the two $binderId$s is a function identifier, the expression associated to the function will be evaluated to yield a value, then interpreted as the rate of *inter-communication*.

B.3 The program file

The central part of a BlenX program is the program file. The program file has a *.prog* extension; it is generated by the following BNF grammar:

$$
\begin{array}{lll}
\textit{program} & ::= & \\
 & & \textit{info}\ \langle\langle\ \textit{rateDec}\ \rangle\rangle\ \textit{decList}\ \textbf{run}\ \textit{bp} \\
 & | & \textit{info}\ \textit{decList}\ \textbf{run}\ \textit{bp} \\
 & & \\
\textit{info} & ::= & \\
 & & [\ \textbf{steps} = \textit{decimal}\ ] \\
 & | & [\ \textbf{steps} = \textit{decimal},\ \textbf{delta} = \textit{number}\ ] \\
 & | & [\ \textbf{time} = \textit{number}\ ] \\
 & & \\
\textit{rateDec} & ::= & \\
 & & \textit{Id} : \textit{rate} \\
 & | & \textbf{CHANGE} : \textit{rate} \\
 & | & \textbf{EXPOSE} : \textit{rate} \\
 & | & \textbf{UNHIDE} : \textit{rate} \\
 & | & \textbf{HIDE} : \textit{rate} \\
 & | & \textbf{BASERATE} : \textit{rate} \\
 & | & \textit{rateDec}, \textit{rateDec} \\
 & & \\
\textit{decList} & ::= & \\
 & & \textit{dec} \\
 & | & \textit{dec}\ \textit{decList}
\end{array}
$$

$$
\begin{array}{lll}
\textit{dec} & ::= & \\
 & & \textbf{let } \textit{Id} : \textbf{pproc} = \textit{process} \ ; \\
 & | & \textbf{let } \textit{Id} : \textbf{bproc} = \textit{box} \ ; \\
 & | & \textbf{let } \textit{Id} : \textbf{complex} = \textit{complex} \ ; \\
 & | & \textbf{let } \textit{Id} : \textbf{prefix} = \textit{actSeq} \ ; \\
 & | & \textbf{let } \textit{Id} : \textbf{bproc} = \textit{Id} \ \langle\langle \ \textit{invTempList} \ \rangle\rangle \ ; \\
 & | & \textbf{when } (\ \textit{cond} \ ) \ \textit{verb} \ ; \\
 & | & \textbf{template } \textit{Id} : \textbf{pproc} \ \langle\langle \ \textit{decTempList} \ \rangle\rangle = \textit{process} \ ; \\
 & | & \textbf{template } \textit{Id} : \textbf{bproc} \ \langle\langle \ \textit{decTempList} \ \rangle\rangle = \textit{box} \ ; \\
 & & \\
\textit{bp} & ::= & \\
 & & \textit{Decimal Id} \\
 & | & \textit{Decimal Id} \ \langle\langle \ \textit{invTempList} \ \rangle\rangle \\
 & | & \textit{bp} \ || \ \textit{bp}
\end{array}
$$

A *prog* file is made up of an header *info*, an optional list of rate declarations (*rateDec*), a list of declarations *decList*, the keyword **run** and a list of starting entities *bp*.

The *info* header contains information used by the BWB simulator that will execute the program. A stochastic simulation can be considered as a succession of time-stamped steps that are executed sequentially, in non-decreasing time order. Thus, the duration of a simulation can be specified as a **time**, intended as the maximum time-stamp value that the simulation clock will reach, or as a number of **steps** that the simulator will schedule and execute. The **delta** parameter can be optionally specified to instruct the simulator to record events only at a certain frequency (and not every time and event is simulated).

A BlenX program is a stochastic program: every single step that the program can perform has a *rate* associated to it, representing the frequency at which that step can, or is expected to, occur. The *rateDec* specifies the global rate associations for individual channel names or for four particular *classes* of actions that a program can perform. In addition, a special class **BASERATE** can be used to set a common basic rate for all the actions that do not have an explicit rate set. The explicit declaration of a rate in the definition of an *action* has the precedence on this global association (see Sec. B.4).

The list of declarations *decList* follows. Each declaration is a small, self-contained piece of code ended by a ';'. A declaration can be named, e.g. it can have an *Id* that designates uniquely the declaration unit in the program, or it can be nameless. Declarations of boxes, processes, sequences of prefixes and complexes must be named[1], while events are *nameless*.

[1]Note that some language constructs, i.e. processes and sequences, can appear throughout a program without a name; they must be named only when they appear as a declaration

B.4 Processes and Boxes

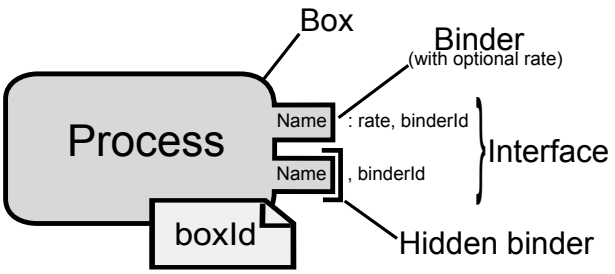
Boxes are generated by the following BNF grammar:

$$
\begin{array}{lll}
\textit{box} & ::= & \\
 & & \textit{binders}\ [\ \textit{process}\ ] \\
 & & \\
\textit{binders} & ::= & \\
 & & \#\ (\ Id\ :\ rate,\ Id\ ) \\
 & |\ & \#\ (\ Id,\ Id\ ) \\
 & |\ & \#\mathbf{h}\ (\ Id\ :\ rate,\ Id\ ) \\
 & |\ & \#\mathbf{h}\ (\ Id,\ Id\ ) \\
 & |\ & \textit{binders},\ \textit{binders} \\
 & & \\
\textit{process} & ::= & \\
 & & \textit{par} \\
 & |\ & \textit{sum}
\end{array}
$$

The intuition is that a box represents an autonomous biological entity that has its own control mechanism (the *process*) and some interaction capabilities expressed by the *binders*.

A *binders* list is made up of a non empty list of *elementary binders* of the form $\#(Id : rate, Id)$ (active with rate), $\#(Id, Id)$ (active without rate), $\#\mathbf{h}(Id : rate, Id)$ (inactive with rate), $\#\mathbf{h}(Id, Id)$ (inactive without rate), where the first Id is the *subject* of the binder, *rate* is the stochastic parameter that quantitatively drives the activities involving the binder (hereafter, stochastic rate) and the second Id represents the identifier of the binder. Binder identifiers cannot occur in processes while subjects of binders can The subject of an elementary beta binder is a binding occurrence that binds all the free occurrences of it in the process inside the box to which the binder belongs. Hidden binders are useful to model interaction sites that are not available for interaction although their status can vary dynamically. For instance a receptor that is hidden by the shape of a molecule and that becomes available if the molecule interacts with/binds to other molecules. Given a list of binders, we denote the set of all its subjects with $sub(binders)$. A box is considered *well-formed* if the list of binders has subjects and identifiers all distinct. Well-formedness of each box defined in a BlenX program is checked statically at compile-time. Moreover, well-formedness is preserved during the program execution.

The BlenX graphical representation of a box is:



Boxes are generated by the following BNF grammar:

$$
\begin{array}{lll}
\textit{process} & ::= & \\
 & & \textit{par} \\
 & | & \textit{sum} \\
 & & \\
\textit{par} & ::= & \\
 & & \textit{parElem} \\
 & | & \textit{sum} \mid \textit{sum} \\
 & | & \textit{sum} \mid \textit{par} \\
 & | & \textit{par} \mid \textit{sum} \\
 & | & \textit{par} \mid \textit{par} \\
 & | & (\ \textit{par}\ ) \\
 & & \\
\textit{sum} & ::= & \\
 & & \textit{sumElem} \\
 & | & \textit{sum} + \textit{sum} \\
 & | & (\ \textit{sum}\ ) \\
 & & \\
\textit{sumElem} & ::= & \\
 & & \textbf{nil} \\
 & | & \textit{seq} \\
 & | & \textbf{if}\ \textit{condexp}\ \textbf{then}\ \textit{sum}\ \textbf{endif} \\
 & & \\
\textit{parElem} & ::= & \\
 & & \textit{Id} \\
 & | & \textit{Id}\ \langle\langle\ \textit{invTempList}\ \rangle\rangle \\
 & | & \textbf{rep}\ \textit{action}\ .\ \textit{process} \\
 & | & \textbf{if}\ \textit{condexp}\ \textbf{then}\ \textit{par}\ \textbf{endif}
\end{array}
$$

$$
\begin{array}{lll}
\textit{seq} & ::= & \\
 & & \textit{action} \\
 & | & \textit{action}\ .\ \textit{process} \\
 & | & \textit{Id}\ .\ \textit{process}
\end{array}
$$

A process can be a *par* or a *sum*. The non-terminal symbol *par* composes through the

binary operator | two processes that can concurrently, while the non-terminal symbol *sum* of the productions of *process* is used to introduce guarded choices of processes, composed with the operator +. The + operator act intuitively as an *or* operator, meaning that at a certain step a process offers a choice of different possible actions such that the execution of each of them eliminates the others. By the contrary, the | operator act intuitively as an *and* operator, meaning that processes composed by | run effectively in parallel.

Notice that we can put in parallel processes also with the constructs Id and $Id\ \langle\langle\ invTempList\ \rangle\rangle$, meaning that we are instantiating a template (see Section B.8) or an occurrence of a process previously defined. As an example, consider the following sequence of processes definition:

```
let p1 : pproc = nil ;
let p2 : pproc = nil | p1 ;
```

Process *p2* is defined as a parallel composition of the **nil** process and an instance of the *p1* process. In BlenX the definition of a process can only rely on identifiers of previously defined processes. Mechanisms of recursive definitions and mutual recursive definitions are not admitted.
The **rep** operator is used to replicate copies of the process passed as argument. Note that we use only guarded replication, i.e. the process argument of the **rep** must have a prefix *action* that forbids any other action of the process until it has been consumed. The **nil** process does nothing (it is a deadlocked process), while the **if-then** statement allows the user to control, through an *expression*, the execution of a *process*. The non-terminal symbol *seq* identifies an action, a process prefixed by an action and a process prefixed by an Id. When in a program we have a process defined using the statement *Id.process* we statically check that the Id corresponds to a previously defined sequence of prefixes.

B.4.1 Actions

The actions that a process can perform are described by the syntactic category *action*.

$$
\begin{array}{lll}
\mathit{action} & ::= & \\
 & & \mathit{Id}\ !\ (\ \mathit{Id}\ ) \\
 & | & \mathit{Id}\ !\ () \\
 & | & \mathit{Id}\ ?\ (\ \mathit{Id}\ ) \\
 & | & \mathit{Id}\ ?\ () \\
 & | & \textbf{delay}\ (\ \mathit{rate}\ ) \\
 & | & \textbf{expose}\ (\ \mathit{Id} : \mathit{rate}\ ,\ \mathit{Id}\ ) \\
 & | & \textbf{hide}\ (\ \mathit{Id}\ ) \\
 & | & \textbf{unhide}\ (\ \mathit{Id}\ ) \\
 & | & \textbf{ch}\ (\ \mathit{Id},\ \mathit{Id}\ ) \\
 & | & \textbf{expose}\ (\ \mathit{rate},\ \mathit{Id} : \mathit{rate},\ \mathit{Id}\ ) \\
 & | & \textbf{hide}\ (\ \mathit{rate},\ \mathit{Id}\ ) \\
 & | & \textbf{unhide}\ (\ \mathit{rate},\ \mathit{Id}\ ) \\
 & | & \textbf{ch}\ (\ \mathit{rate},\ \mathit{Id},\ \mathit{Id}\ )
\end{array}
$$

The first four actions are common to most process calculi. The first pair of actions represent an output/send of a value on a channel, while the second pair represent the input/reception of value or a signal on a channel. The remaining actions are peculiar of the BlenX language. The definition of *free names* for processes is obtained by stipulating that $Id?(Id').process$ is a binder for Id' in *process* and that $\textbf{expose}(Id : rate, Id).process$ and $\textbf{expose}(rate, Id : rate, Id).process$ are binders for Id in *process*. The definitions of *bound names* and of *name substitution* are extended consequently. The definition of free and bound names for boxes is obtained by specifying that the set of free names of a box $binders[process]$ is the set of free names of the *process* minus the set $sub(binders)$ of subjects of the binders. Moreover, as usual two processes *process* and $process'$ are α-equivalent if $process'$ can be obtained from *process* by renaming one or more bound names in *process*, and vice versa. As usual renaming avoids name clashes, i.e. a free name never becomes bound after the renaming. More details of this definitions can be found in [52].

species:

In BlenX species are defined as classes of boxes which are *structurally congruent*. The structural congruence for boxes, denoted with $\equiv$, is the smallest relation which satisfies the following laws:

- $process \equiv process'$, if $process$ and $process'$ are α-equivalent
  Es: `z?(t).t!().nil` $\equiv$ `z?(w).w!().nil`

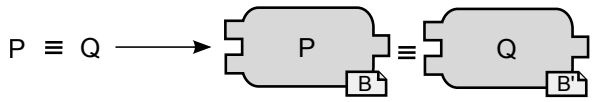
- $process \mid \mathbf{nil} \equiv process$ and $sum \mid \mathbf{nil} \equiv sum$
  Es: `P | nil` $\equiv$ `P`

- $process_1 \mid (process_2 \mid process_3) \equiv (process_1 \mid process_2) \mid process_3$ and $sum_1 \mid (sum_2 \mid sum_3) \equiv (sum_1 \mid sum_2) \mid sum_3$
  Es: `(P + Q) | (R | S)` $\equiv$ `((P + Q) | R) | S`

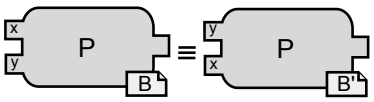
- $process_1 \mid process_2 \equiv process_2 \mid process_1$ and $sum_1 \mid sum_2 \equiv sum_2 \mid sum_1$
  Es: `P | Q` $\equiv$ `Q | P`

- $repaction.process \equiv action.(process \mid repaction.process)$
  Es: `rep x!().nil` $\equiv$ `x!().(nil | rep x!().nil))`

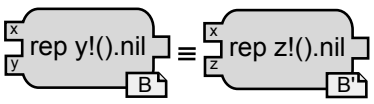
- $binders[process] \equiv binders[process']$, if $process \equiv process'$



- $binders, binders'[process] \equiv binders', binders[process]$



- $\#(Id : rate, Id_1), binders[process] \equiv \#(Id' : rate', Id_1), binders[process\{^{Id'}/_{Id}\}]$
  and
  $\#(Id, Id_1), binders[process] \equiv \#(Id', Id_1), binders[process\{^{Id'}/_{Id}\}]$ and
  $\#\mathbf{h}(Id : rate, Id_1), binders[process] \equiv \#\mathbf{h}(Id' : rate', Id_1), binders[process\{^{Id'}/_{Id}\}]$
  and
  $\#\mathbf{h}(Id, Id_1), binders[process] \equiv \#\mathbf{h}(Id', Id_1), binders[process\{^{Id'}/_{Id}\}]$
  if $Id' \notin sub(binders)$



Consider for example the program:

```
...
let b1 : bproc = #(x:1,A)
  [ ( x!().nil + z?(w).w!().nil ) | x!(z).nil ];
...
let b2 : bproc = #(y:1,A)
  [ y!(z).nil | ( z?(t).t!().nil + y!().nil ) ];
...
```

In the example we have $b1 \equiv b2$, hence the boxes belong to the same species. Notice that if we have multiple definition of boxes that represent the same species, then at run-time they are collected together and the species name is taken from the first definition (e.g. in the example the name of the corresponding species is $b1$). Hereafter, when we say that in a particular state of execution of a program the cardinality of a box species $b1$ is n we mean that in that state of execution the number of boxes structurally congruent to $b1$ is n.

Intra-communication:

consider the following piece of code:

```
let p : pproc =
  x!(m).nil + y?(z).z?().nil + y?().nil ;


let b1 : bproc = #(x:1,A),#h(m,B)
  [ p | x?(z).z!(c).nil + x?().nil + y!().nil ];
```

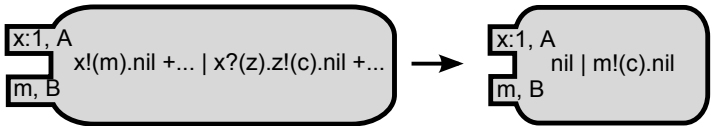
Box *b1* has a binder $\#(x : 1, A)$ and an internal process defined as a parallel composition of the *sum* process p and the *sum* process $x?(z).z!(c).nil + y!().nil$. Each *sum* composes processes guarded by input or output actions. Parallel processes that perform complementary actions on the same channel inside the same box can synchronize and eventually exchange a message, generating an *intra-communication*. In the example, several intra-communications can be performed. Indeed, each output in the first *sum* can synchronize with an input on the same channel in the other *sum*, and vice-versa. Consider the input/output pair:

```
x!(m).nil + ... | x?(z).z!(c).nil + ...
```

$x?(z)$ represents an input/reception of something that will instantiate the placeholder z over channel x, while $x!(m)$ represent an output/send of a value m over channel x. The placeholder z in the input is a binding occurrence that binds all the free occurrences of z in the scope of the prefix $x?(z)$ (in this case in $z!(c).nil$). Sometimes the channel name x is called the subject and the placeholder/value z is called the object of the prefix. The execution of the intra-communication consumes the input and output prefixes and the object m of the output flows from the process performing the output to the one performing the input:

```
nil | m!(c).nil
```

The flow of information affects the future behavior of the system because all the free occurrences bound by the input placeholder are replaced in the receiving process by the actual value sent by the output (in the example z is substituted by m). The graphical representation of the intra-communication is



If an input has no object and it is involved in a intra-communication:

```
x!(m).nil + ... | x?().nil + ...
```

then the two prefixes are consumed and no substitution is performed:

```
nil | nil
```

If an output has no object and is involved in an intra-communication:

```
... + y?(z).z?().nil + ... | ... + y!().nil + ...
```

then the two prefixes are consumed and the substitution in the process prefixed by the input is performed by using a reserved string *$emp* on which no further intra-communication is allowed.

```
$emp?().nil | nil
```

Notice that the string *$emp* cannot be generated by the regular expression defining the Id (see Section B).

If object-free outputs and inputs synchronize in an intra-communication:

```
... + y?().nil + ... | ... + y!().nil + ...
```

then the two prefixes are consumed, generating the process:

```
nil | nil
```

The stochastic nature of BlenX emerges in the above examples through the rates associated to the input/output channels. In particular, if the channel is bound to a binder, the rate is specified in the binder definition; if the binder is $\#(x:1,A)$ (or $\#h(x:1,A)$) the rate associated to an intra-communication over channel x is 1, while if the binder is $\#(x,A)$ (or $\#h(x,A)$) the associated rate is assumed to be 0 and hence no intra-communications over channel x can happen.

If the channel is not bound to a binder, then the rate has to be defined in the global $rateDec$. In particular, if $rateDec$ is:

```
<< ... , x : 2.5 , ... >>
```

the rate associated to an intra-communication over channel x is 2.5. Instead, if no specific x rate definition appears in the *rateDec* list, then the **BASERATE** definition is used. If also no **BASERATE** definition appears in the *rateDec* list, then a compile time error is generated. In the example, intra-communications over channel y need a specific definition or the **BASERATE** in the *rateDec* list.

Since to each communication channel in a box we can associate an unique rate r, then the overall propensity of performing an intra-communication on a channel x is given by the following formula:

$$r \times ((In(x) \times Out(x)) - Mix(x))$$

where $In(x)$ identifies all the enabled input on x, $Out(x)$ the enabled output on x and $Mix(x)$ all the possible combinations of input/output within the same *sum*. As an example, consider the box:

```
let b1 : bproc = #(x,A),#(m,B)
   [ x?().nil + x!().nil + x!().nil |
     x?().nil + x!().nil + x!().nil ]
```

Let the rate associated to x be 3, the overall propensity associated to an intra-communication on the channel x is calculated using the previous formula obtaining:

$$3 \times ((2 \times 4) - 4) = 12$$

where term (2×4) represents all the combinations of input/output and the last 4 represents the combinations contained in the same *sum* and hence the ones that cannot give raise to an inter-communication.

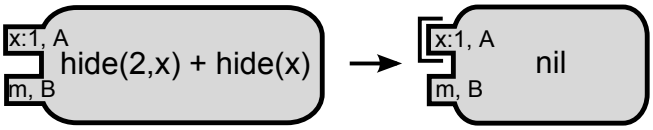
Notice that multiplying 12 by the cardinality of the species $b1$ we obtain the overall propensity that a box of that species performs an intra-communication on channel x.

hide:

consider the following box:

```
let b1 : bproc = #(x:1,A),#(m,B)
   [ hide(2,x).nil + hide(x).nil ]
```

Box $b1$ can perform two *hide* actions. The execution of both actions cause the modification of the box interface hiding the binder $\#(x : 1, A)$. The graphical representation of the actions is



The only difference between the actions is the stochastic rate association. Indeed, the first action specifies its own rate and hence is performed with a rate of value 2. For the second action, a rate has to be defined in the global *rateDec*. In particular, if *rateDec* is:

```
<< ... , HIDE : 4 , ... >>
```

the rate associated to the all hide actions is 4. Instead, if no specific **HIDE** rate definition appears in the *rateDec* list, then the **BASERATE** definition is used. If also no **BASERATE** definition appears in the *rateDec* list, then a compile-time error is generated.

To compute the overall propensity associated to *hide* actions performed by boxes of a given species, we need to calculate all the possible combinations. This combination is obtained by multiplying the number of all the enabled hide actions $hide(r, x)$ on the same binder with the same rate r and the number of all the enabled hide actions $hide(x)$ on the same binder by the corresponding base rates. The overall propensity is then obtained by multiplying this combination with the cardinality of the species.

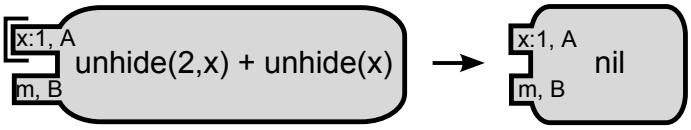
Notice that an hide action on an binder which is already hide is not enabled. A definition of an hide action on a name which is not a binder is not enabled and generates a compile-time warning.

unhide:

consider the following box:

```
let b1 : bproc = #h(x:1,A),#(m,B)
   [ unhide(2,x).nil + unhide(x).nil ]
```

Box $b1$ can perform two *unhide* actions. The execution of both actions cause the modification of the box interface unhiding the binder $\#h(x : 1, A)$. The graphical representation of the actions is



The only difference between the actions is the stochastic rate association. Indeed, the first action specifies its own rate and hence is performed with a rate of value 2. For the second action, a rate has to be defined in the global *rateDec*. In particular, if *rateDec* is:

```
<< ... , UNHIDE : 4 , ... >>
```

the rate associated to the hide action is 4. Instead, if no specific **UNHIDE** rate definition appears in the *rateDec* list, then the **BASERATE** definition is used. If also no **BASERATE** definition appears in the *rateDec* list, then a compile time error is generated.

To compute the overall propensity associated to *unhide* actions performed by boxes of a given species, we need to calculate all the possible combinations. This combination is obtained by multiplying the number of all the enabled unhide actions $unhide(r, x)$ on the same binder with the same rate r and the number of all the enabled unhide actions $unhide(x)$ on the same binder by the corresponding base rates. The overall propensity is then obtained by multiplying this combination with the cardinality of the species.

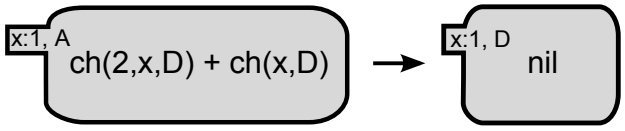
Notice that an unhide action on an binder which is already unhidden is not enabled and that a definition of an unhide action on a name which is not a binder is not enabled and generates a compile-time warning.

change:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ ch(2,x,D).nil + ch(x,D).nil ]
```

Box $b1$ can perform two *change* actions. The execution of both actions cause the modification of the box interface changing the value A of the binder $\#(x : 1, A)$ into D. The graphical representation of the actions is



The first action specifies its own rate and hence is performed with a rate of value 2. For the second action, a rate has to be defined in the global *rateDec*. In particular, if *rateDec* is:

```
<< ... , CHANGE : 4 , ... >>
```

the rate associated to the hide action is 4. Instead, if no specific **CHANGE** rate definition appears in the *rateDec* list, then the **BASERATE** definition is used. If also no **BASERATE** definition appears in the *rateDec* list, then a compile time error is generated.

To compute the overall propensity associated to *change* actions performed by boxes of a given species, we need to calculate all the possible combinations. This combination is

obtained by multiplying the number of all the enabled change actions $ch(r, x, D)$ on same values and the number of all the enabled change actions $ch(x, D)$ on same binders and with equal substituting types by the corresponding base rates. The overall propensity is then obtained by multiplying this combination with the cardinality of the species.

die:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ die(2).nil ]
```

Box $b1$ can perform a *die* action. The execution of the action eliminates the related box. The graphical representation of the action is

The action is executed with the specified rate of value 2. To compute the overall propensity associated to *die* actions we calculate the number of all the enabled die actions $die(r)$ on same rates and multiply this values by the corresponding base rates and by the cardinality of the species.

delay:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ delay(2).nil ]
```

Box $b1$ can perform a *delay* action. The execution of the action allows the box to evolve internally. The graphical representation of the action is

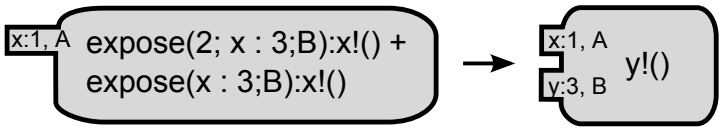
The action is executed with the specified rate of value 2. Moreover, Nil is used to identify a deadlocked box which does nothing. To compute the overall propensity associated to *delay* actions we calculate the number of all the enabled delay actions $delay(r)$ on same rates and multiply this values by the corresponding base rates and by the cardinality of the species.

expose:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ expose(2,x:3,B).x!() + expose(x:3,B).x!() ]
```

Box $b1$ can perform two *expose* actions. The execution of both actions add a new binder $\#(y : 3, B)$ to the interface, by renaming the subject into a new name to avoid clashes of names (x renamed into y with all the occurrences bound by the subject in the expose). The graphical representation of the actions is



The first action specifies its own rate and hence is performed with a rate of value 2. For the second action, a rate has to be defined in the global *rateDec*. In particular, if *rateDec* is

```
<< ... , EXPOSE : 4 , ... >>
```

the rate associated to the hide action is 4. Instead, if no specific *EXPOSE* rate definition appears in the *rateDec* list, then the *BASERATE* definition is used. If also no *BASERATE* definition appears in the *rateDec* list, then a compile-time error is generated. Expose actions are considered separately and hence the overall propensity that a box species perform an expose action is calculated multiplying the rate associated to the action by the action rates and by the cardinality of the box species performing the action.

Notice that an expose action of a binder identifier which is already present in the set binders of the box is not enabled.

if-then statement:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ if (x,unhidden) and (x,A) then x!().nil ]
```

Box $b1$ can perform the output action $x!()$ only if the conditional expression is satisfied by the actual configuration of the binders of the box containing the if-then statement. In this example if the binder with subject x is unhidden and its binder identifier is A, then

the output can be executed. The general form of the conditional expressions of if-then statements are generated by the following BNF grammar:

$$
\begin{array}{lll}
\textit{condexp} & ::= & \\
 & & \textit{atom} \\
 & | & \textit{condexp}\ \textbf{and}\ \textit{condexp} \\
 & | & \textit{condexp}\ \textbf{or}\ \textit{condexp} \\
 & | & \textbf{not}\ \textit{condexp} \\
 & | & (\ \textit{condexp}\ ) \\
 & & \\
\textit{atom} & ::= & \\
 & & (\ \textit{Id},\ \textit{Id}\ ) \\
 & | & (\ \textit{Id},\ \textbf{hidden}\ ) \\
 & | & (\ \textit{Id},\ \textbf{unhidden}\ ) \\
 & | & (\ \textit{Id},\ \textbf{bound}\ ) \\
 & | & (\ \textit{Id},\ \textit{Id},\ \textbf{hidden}\ ) \\
 & | & (\ \textit{Id},\ \textit{Id},\textbf{unhidden}\ ) \\
 & | & (\ \textit{Id},\ \textit{Id},\ \textbf{bound}\ )
\end{array}
$$

Conditional expressions are logical formulas built atoms (conditions on binder states) connected by classical binary logical operators (*and*,*or*,*not*). In the atoms the first *Id* identifies the subject of a binder, while the second *Id* (if present) identifies the binder identifier. The keywords *hidden*, *unhidden* and *bound* identify the three states in which a binder can be. As an example, the conditional expression:

```
(x,A) and ( not(y,B,hidden) or (z,bound) )
```

is satisfied only if the box has a binder with subject x of type A and has a binder with subject y which is not hidden and with type different from B or has a bound binder with subject z (see Section B.5). Notice that boxes of the form:

```
let b1 : bproc = #(x:1,A)
   [ if (y,unhidden) and (x,A) then x!().nil ]

let b1 : bproc = #(x:1,A)
   [ y?(x).if (x,unhidden) and (x,A) then x!().nil ]
```

generates compile-time warnings. Indeed, in the first case the $(y, \textit{unhidden})$ do not refer to any binder of the box, while in the second case the atom $(x, \textit{unhidden})$ is bound by the input $y?(x)$ and not by the subject of the binder. In general, at run-time atoms on binders which are not present are evaluated as false value.

inter-communication:

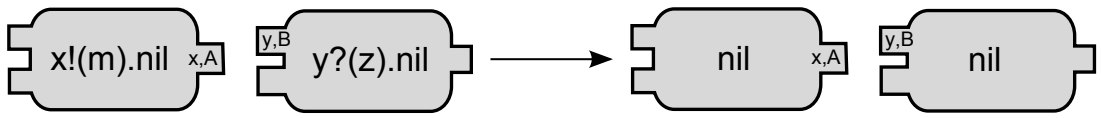
processes in different boxes can perform an *inter- communication* (distinct from the *intra-communication* described above) if one sends a value y over a link x that is bound to an

active binder of the box $\#(x : r, A)$ and a process in another box is willing to receive a value from a *compatible* binder $\#(y : s, B)$ through the action $y!(z)$. The two corresponding binders are compatible if a *compatibility* value (i.e. a stochastic rate) greater than zero is specified in the binder declaration file

```
{...,A,...,B,...}
%%
{ ... , (A,B,2.5), ... }
```

Note that intra-communications occur on perfectly symmetric input/output pairs that share the same subject, while inter-communication can occur between primitives that have different subjects provided that their binder identifiers are compatible. This new notion of communication is particularly relevant in biology where interactions occur on the basis of sensitivity or affinity which is usually not exact complementarity of molecular structures. The same substance can interact with many other in the same context, although with different levels of affinity expressed through different properties.

The graphical representation of an inter-communication is:



If the compatibility is specified by a stochastic rate, the overall propensity of the inter-communication is computed as bimolecular rate (see Section 3.2.1), considering all the possible combinations of inputs on channel x in the first box and outputs on y in the second box and multiplying this value with the product of the cardinality of the box species in the system. As an example consider the program:

```
...
let b1 : bproc = #(x:1,A)
  [ x!().nil + x!().nil | x!().nil ];
...
let b2 : bproc = #(y:3,B)
  [ y?().nil | y?().nil ];
...
let b3 : bproc = #(z:2,C)
  [ z?().nil ];
run 10 A || 20 B || 5 b3
```

Assuming boxes $b1$ and $b2$ defines two different species, the overall propensity of the inter-communication on boxes species A and B is

$$(2.5 \times (3 \times 2)) \times (10 \times 20)$$

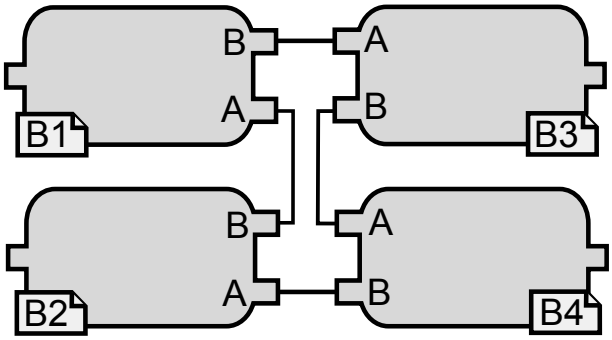


Figure B.1: Example of complex.

where 2.5 is the basal rate, (3×2) is the number of combinations of inputs and outputs and (10×20) is the product of the cardinality of the two box species.

If the compatibility is expressed by a function defined in the declaration file:

```
{...,A,...,B,...}
%%
{ ..., (A,B,f1), ... }
```

then the overall propensity of the inter-communication is computed as a *rate function* (see Section 3.2.1) and therefore it does not depend directly on the cardinality of the involved species. In the example, if the function $f1$ is as:

```
...
let f1 : function = 2 * pow(|b3|,2);
...
```

the overall propensity of the inter-communication has value 50.

Notice that in an inter-communication, values corresponding to binder subjects cannot be sent.

B.5 Complexes

A complex is a graph-like structure where boxes are nodes and dedicated communication bindings are edges. Figure B.1 report an example is reported, where $b_0 = \#(x : r_0, A_0)$ and $b_1 = \#(y : r_1, A_1)$. In BlenX, complexes are not defined as *species*, but as graph-like structures of box species. Complexes can be created automatically during the program execution or they can be instantiated also in the initial program. A complex can be

defined using the following BNF grammar:

$$
\begin{array}{lll}
\textit{complex} & ::= & \\
 & & \{\ (\ \textit{edgeList}\ )\ ;\ \textit{nodeList}\ \} \\
 & & \\
\textit{edgeList} & ::= & \\
 & & \textit{edge} \\
 & \mid & \textit{edge},\ \textit{edgeList} \\
 & & \\
\textit{edge} & ::= & \\
 & & (\ \textit{Id},\ \textit{Id},\ \textit{Id},\ \textit{Id}\ ) \\
 & & \\
\textit{nodeList} & ::= & \\
 & & \textit{node} \\
 & \mid & \textit{node nodeList} \\
 & & \\
\textit{node} & ::= & \\
 & & \textit{Id}\ :\ \textit{Id} = (\ \textit{complBinderList}\ )\ ; \\
 & \mid & \textit{Id} = \textit{Id}\ ; \\
 & & \\
\textit{complBinderList} & ::= & \\
 & & \textit{Id} \\
 & \mid & \textit{Id},\ \textit{complBinderList}
\end{array}
$$

A complex is created by specifying the list of edges (*edgeList*) and the list of nodes (*nodeList*). Each *edge* is a composition of 4 *Ids*. The first and the third identifiers represent node names, while the others represent subject names. Each *node* in the *nodeList* associates to a node name the corresponding box name and specifies the subjects of the bound binders. As an example, consider the program:

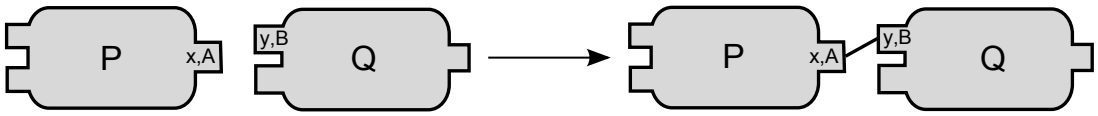
```
...
let b1 : bproc = #(x:r0,A0),#(y:r1,A1)
  [ x!().nil ];
...
let b2 : bproc = #(x:r0,A0),#(y:r1,A1)
  [ y!().nil ];
...
let C : complex =
{
  (
    (Box0,y,Box1,x),(Box1,y,Box2,x),
    (Box2,y,Box3,x),(Box3,y,Box0,x)
```

```
  );
  Box0:b1=(x,y);
  Box1:b2=(x,y);
  Box2=Box0;
  Box3=Box1;
}
...
```

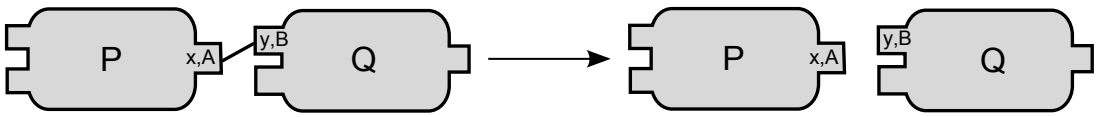
The complex C defines a complex with a structure equivalent to the one reported in Figure B.1. A complex can also be generated automatically at run-time thorough a set of primitives for complexation and decomplexation. The ability of two boxes to form and break complexes is defined in the bind declaration file by specifying for pairs of binder identifiers triples of stochastic rates:

```
{...,A,...,B,...}
%%
{ ..., (A,B,1.5,2.5,10), ... }
```

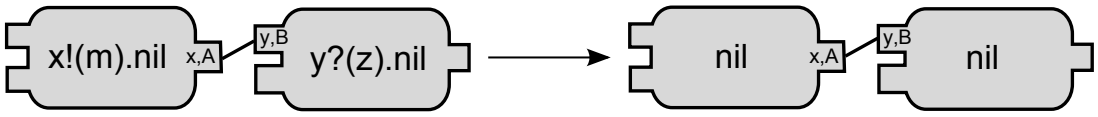
Complex and *decomplex* operations create and delete dedicated communication bindings between boxes. The biological counterpart of this construct is the binding of a ligand to a receptor, or of an enzyme to a substrate through an active domain. Given two boxes with binder with identifiers A and B respectively, the *complex* operation creates, with rate 1.5, a dedicated communication binding:



while the *decomplex* operation deletes, with rate 2.5, an already existing binding:



Finally, the *inter-complex communication* operation enables, with rate 10, a communication between complexed boxes through the complexed binders:



Notice that a binder in *bound* status is identified by $\#c(y : B)_s$ where c means that the corresponding box is part of a complex. It is important to underline that, although the bound status cannot be explicitly specified trough the syntax of the language and is used only as an internal representation, a binder in bound status is different from a hidden or unhidden binder and hence the structural congruence definition has to be extended accordingly:

- $\#\mathbf{c}(Id : rate, Id_1), binders[process] \equiv \#\mathbf{c}(Id' : rate', Id_1), binders[process\{^{Id'}/_{Id}\}]$
  if $Id' \notin sub(binders)$
- $\#\mathbf{c}(Id, Id_1), binders[process] \equiv \#\mathbf{c}(Id', Id_1), binders[process\{^{Id'}/_{Id}\}]$
  if $Id' \notin sub(binders)$

B.6 Events

Events specify statements, or *verbs*, to be executed with a specified rate and/or when some conditions are satisfied. A single *event* is the composition of a condition *cond* and an action *verb* (recall the syntax of declarations in Sect. B.3).

dec ::=
| ...
| **when** (*cond*) *verb* ;
| ...

Conditions

Events are used to express actions that are enabled by global conditions, expressed by *cond*. Conditions are used to trigger the execution of an event when some elements are present in the system, when a particular condition is met, with a given rate, or at a precise simulation time or simulation step.

$$
\begin{array}{lll}
\mathit{cond} & ::= & \\
 & & \mathit{entityList} \ : \ \mathit{EvExpr} \ : \ \mathit{rate} \\
 & | & \mathit{entityList} \ : \ \mathit{EvExpr} \ : \ \mathit{funcId} \\
 & | & \mathit{entityList} \ : \ \mathit{EvExpr} \ : \\
 & | & \mathit{entityList} \ :: \ \mathit{rate} \\
 & | & \mathit{entityList} \ :: \ \mathit{funcId} \\
 & | & : \ \mathit{EvExpr} \ : \\
 & & \\
\mathit{entityList} & ::= & \\
 & & \mathit{boxId} \\
 & | & \mathit{boxId}, \ \mathit{entityList} \\
 & & \\
\mathit{EvAtom} & ::= & \\
 & & \mid \mathit{Id} \mid = \mathit{Decimal} \\
 & | & \mid \mathit{Id} \mid < \mathit{Decimal} \\
 & | & \mid \mathit{Id} \mid > \mathit{Decimal} \\
 & | & \mid \mathit{Id} \mid \ ! = \mathit{Decimal} \\
 & | & \textbf{time} = \mathit{Real} \\
 & | & \textbf{steps} = \mathit{Decimal} \\
 & | & \mathit{stateOpList} \\
 & & \\
\mathit{EvExpr} & ::= & \\
 & & \mathit{EvAtom} \\
 & | & \mathit{EvExpr} \ \textbf{and} \ \mathit{EvExpr} \\
 & | & \mathit{EvExpr} \ \textbf{or} \ \mathit{EvExpr} \\
 & | & \textbf{not} \ \mathit{EvExpr} \\
 & | & (\ \mathit{EvExpr} \ )
\end{array}
$$

More precisely, a condition *cond* consists of three parts: *entityList*, a list of boxes present in the system; an expression used to enable or disable the event; a *rate* or rate function, used to stochastically select and include them in the set of standard interaction-enabled actions.

EvExpr can be combined through logical operators starting from atoms; furthermore, a condition can specify both an *EvExpr* and a rate (see definition of *cond*), so that we can simultaneously address rates and conditions (e.g. on structures and concentrations of species). As an example, consider the following event:

```
when(A, B : (|A| > 2 and |B| > 2) : rate(r1)) join (C);
```

The entities involved in the event are A and B, as they appear in the *entityList*; moreover, the *EvExpr* requires the cardinality of both the species identified by boxes A and B to be greater than two, so the event will fire only when there are at least two A and two B in the system. When the condition is satisfied, the event will fire with rate $r1$.

The *EvAtom*s evaluate to the boolean values *true* and *false*, and can be used to express conditions over concentrations of species identified by an Id (| Id | op $Decimal$, where $op \in <, >, =, !=$) or over simulation time or simulation steps.

A condition on *simulation time* will be satisfied as soon as the simulation clock is greater or equal to the specified time; a conditions on *simulation steps* will be satisfied as soon as the step count will exceed the number specified in the *EvAtom*. In both cases, the condition will remain *true* until the event is fired. So, events for which the only condition specified is the number of steps or the execution time are guaranteed to fire exactly once. For example, the event:

```
when(A : time = 3.0 : inf) delete;
```

will fire as soon as the simulation clock reaches 3.0, removing one A form the system.

It is important to make a remark: *Id*s that can appear in the *EvExpr* must be entities that appear in the *entityList*. The following code:

```
when(A, B : (|C| > 2) : rate(r1)) join (C);
```

will produce a compilation error. The only exception is when the *entityList* is empty (the sixth case in the BNF declaration of *cond*). In this case, the *Id*s in the expression can be chosen among all the *betaId*s or *varId*s already declared, with no restrictions.

If more complex expressions are needed (i.e. for expressing conditions on more species in the system) it is possible to use a *rate function* instead (see Sect. B.1).

Note that the number of *Id*s specified in the *entityList* depends on the event verb that is used for the current event. See the next section for more details on this point.

Events, like all the other actions that can trigger an execution in a BlenX program, can have an associated rate. It is possible to specify both rate constants (form 1, 4 in the BNF specification of *cond*) or rate functions (form 2, 5 in the BNF specification of *cond*). The rate constants are treated differently in the case of events with or without explicit *EvExpr*s. When there is no *EvExpr*, the rate is computed as a monomulecular or bimolecular rate, using the concepts introduced in Sec. 3.2.1. In the monomolecular case, the number h_μ of reactant combinations is equal to the cardinality of the species designated by the unique box in the *entityList*, in the bimolecular case the number h_μ of reactant combinations is the product of the cardinalities of the species designated by the first and second box in the *entityList*.

When a condition is present, the rate is a *constant rate* (see Sec. 3.2.1). This is to avoid the case in which a decimal value used in a comparison operation in a *EvAtom* can influence the rate of that action. Consider the two following pieces of code:

```
when (A : |A| > 2 : r) delete(2);
```

and

```
when (A : |A| > 10 : r) delete(2);
```

The second event will be triggered when there is an higher concentrations of boxes of species A, ten in this case. If we use the monomulecular way of computing the actual rate, the second event will be triggered with an higher rate than the first one, as monomolecular rates are proportional to the reactants concentration. What we intuitively expect, however, is that the two actions will take place with the same *actual rate*, hence the event rate is considered as a constant rate. Consider also the following example:

```
when (A : |A| = 0 : r) new;
```

Intuitively, this event introduces a box of species A with a given rate when there are no such entities in the system. If we compute the rate in the usual way, the event will be never executed (which is clearly different form what we expect).

For the case in which rates are specified as functions (form 2, 5 in the BNF specification of *cond*), the function is evaluated and the resulting value is used directly to compute the propensity function (see Sec. 3.2.1).

Verbs.

Events can split an entity into two entities, join two entities into a single one, inject or remove entities into/from the system. Events are feature is essential to program perturbation of the systems triggered by particular conditions emerging during simulation and to observe how the overall behaviour is affected. An example could be the knock-out of a gene at a given time.

verb ::=
- **split** (*boxId*, *boxId*)
- | **join** (*boxId*)
- | **new** (*Decimal*)
- | **delete** (*Decimal*)
- | **new**
- | **delete**
- | **update** (*varId*, *funcId*)

Verbs and conditions have some dependencies: not all verbs can apply to all conditions. The *entityList* in *cond* is used by the event to understand which species the event will modify; at the same time, the *verb* dictates which action will take place. Indeed, a *verb* specify how many entities will be present in the *entityList*:

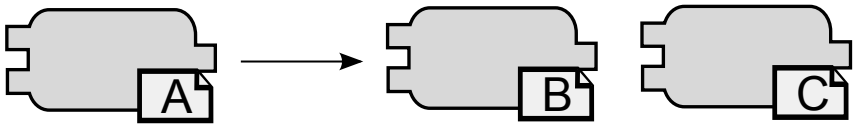
- the **split** verb requires exactly one entity to be specified in the condition list;
- the **join** verb requires exactly two entities to be specified in the condition list;

- the **new** and **delete** verbs requires exactly one entities to be specified in the condition list;
- the **update** verb requires that the condition list is empty (form 6 in the BNF specification of *cond*).

The **split** verb removes one box of the specified species from the system, and substitutes it with the two other entities specified in the ($boxId$, $boxId$) pair. In the following piece of code:

```
when(A :: r) split(B, C);
```

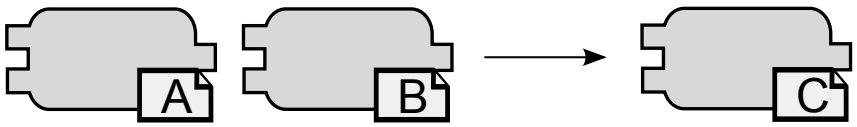
One A will be substituted by one B and one C, leading to the following behaviour:



The **join** verb removes two boxes, one for each of the species specified in the list, from the system, and introduces on box of the species specified in its ($boxId$) argument:

```
when(A, B :: r) join(C);
```

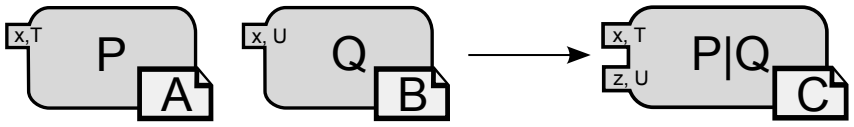
One A and one B will be joined in one C, leading to the following behaviour:



The target of the join, i.e. the box specified as argument, is optional:

```
when(A, B :: r) join;
```

If no box is specified, a new box, automatically generated form two originating boxes, will be introduced into the system:



The new box will have as the interface the union of the interfaces of boxes A and B, and as its internal process the parallel composition of the internal processes of A and B.

The **new** and **delete** verbs introduce and remove boxes. **New** will introduce into the system one copy (in its parameterless variant) or n copies (in its second variant) of the single entity present in the event list. As for the other events, the event is triggered with a certain rate and/or with a condition expression is met. The behaviour of **delete** is complementary: it will remove one or more boxes from the system when its *cond* triggers the event. Note that in the case of **delete** a box of the species specified in the entity list must be present:

```
when(A : |A| = 0 : inf) delete;
when(A : |A| = 0 : inf) new(2);
```

The first event will never fire, while the second one will fire as soon as there are no more boxes of species A in the system. Other examples of valid events are:

```
when(A : (|A| > 1 and |A| < 10)  : inf) new(100);
when(A :: r) delete;
when(A : (|A| = 2) and (steps = 3000) : inf) delete(2);
```

This set of event will produce oscillations of the concentrations of A, by introducing some boxes when the concentrations falls under a threshold and deleting them with a *decay* of rate r, until the simulation reaches 3000 steps; after that, all As are deleted from the system and no further evolution is possible.

The **update** verb is used to modify the value of a variable in the system. When the event is fired, the function *funcId* and the resulting value is assigned to the variable *varId*. Functions and variables are explained in greater detail in Sec. B.1; here it is sufficient to know that variables are global *Id*s bound to real values, and that functions are mathematical expressions on variables and cardinality of entities that evaluate to a real value.

The condition of an **update** event has no entities in its *entityList*, and no rate or rate function in its *rate* part: the event is triggered as soon as its *EvExpr* evaluates to *true*. Jointly to an **update** event it is possible to use a particular kind of condition, based on the traversal of successive states.

$$
\begin{array}{lll}
statOpList & : & \\
 & & stateOp \\
 & | & stateOp, statOpList \\
 & & \\
statOp & : & \\
 & & Id \leftarrow Real \\
 & | & Id \rightarrow Real
\end{array}
$$

The list of states to be traversed are expressed in a *stateOpList*; each *stateOp* element in the list expresses a condition on the quantity of an *Id* (i.e. cardinality of boxes for *boxId* or the *value* bound to a variable for *varId*).

StateOps are examined in sequence, one after the other. We say that a *stateOp* becomes *valid* when the condition on its *Id* is met for the first time. The '$\rightarrow$' operator recognizes when the quantity bound to *Id* becomes greater than the specified real value, while the

'←' operator recognizes when the quantity bound to *Id* becomes smaller than the specified real value.

When a *stateOp* becomes valid, the *EvExpr* passes to the evaluation of the following *stateOp* of the list. As soon as the last state in the *stateOpList* becomes valid, the *EvExpr* evaluates to *true*, so the event (update, in this case) can be fired. Once fired, the *EvExpr* restart its evaluation from the beginning of the *stateOpList*, waiting for the first *stateOp* to become valid again.

B.7 Prefixes

Prefixes are generated by the following BNF grammar:

$$
\begin{array}{lll}
dec & ::= & \\
 & & \ldots \\
 & | & \textbf{let } Id : \textbf{prefix} = actSeq \ ; \\
 & & \\
actSeq & ::= & \\
 & & action \\
 & | & action \ . \ prefix
\end{array}
$$

In other words, a *prefix* is an object bound to a sequence of actions. Prefixes are used exclusively in templates (see Sec. B.8). Templates can contain variable parts; among these parts, it is possible to specify a variable *prefix* that can be substituted with a custom sequence of actions when instantiated. An example of the usage of prefixes for easing template definitions is given in the next Section.

B.8 Templates

Templates, often referred to as *generics* or *parametric processes*, are a feature of many programming languages that allows code in an extended grammar in which code can contain variable parts that are then instantiated later by the compiler with respect to the base grammar.

In BlenX template code is *specialized* and *instantiated* at compile time using binder identifiers, code or names that are passed as template arguments. Therefore, BlenX provides a grammar for defining templates and code to instantiate and use them.

Template declaration.

It is possible to define templates for processes, boxes and sequences. The BNF for template declaration and definition is the following:

$$
\begin{array}{lll}
dec & ::= & \\
 & & \ldots \\
 & | & \textbf{template } Id : \textbf{pproc } \langle\langle\ formList\ \rangle\rangle = piProcess\ ; \\
 & | & \textbf{template } Id : \textbf{bproc } \langle\langle\ formList\ \rangle\rangle = betaProcess\ ; \\
form & ::= & \\
 & & \textbf{name } Id \\
 & | & \textbf{pproc } Id \\
 & | & \textbf{binder } Id \\
 & | & \textbf{prefix } Id \\
 & & \\
formList & ::= & \\
 & & form \\
 & | & form,\ formList
\end{array}
$$

The declaration of a template **bproc** or **pproc** follows closely the declaration of their standard counterparts, with the **let** keyword substituted by **template**, and an additional list of template formal parameters enclosed by double angular parenthesis.

The template parameter *formList* is a comma-separated list of *form*s; each *form* declares a template argument made up of a keyword anong name, pproc, binder, prefix followed by an Id. The Id will be added to the environment of the object being defined, acting as a placeholder for the object that will be used during parameter instantiation. For example, in the following code:

```
template P : pproc<<pproc P1, name N1, name N2, binder T1>> =
   x?().N1!().ch(N2, T1).P1;
```

we do not have to define the pproc $P1$, nor we have to insert the binder identifier $T1$ into the type file: this piece of code will compile without errors, as the process $P1$ and the binder identifier $T1$ are inserted into P's environment as template arguments. P will be treated by the compiler as pproc with four template arguments: a process, two names and a binder identifier. Note that the notion of "name" is pretty general: it can be any name appearing into the template, being it a channel name, an action argument or a binder name.

Template instantiation.

A declared template (pproc or bproc) is held by the compiler in its symbol-table in order to satisfy following *invocations* or *instantiations* of that template. Template instantiation

is the compile time procedure that substitute the template formal parameters with the actual parameter with which the template object will be used. For example, the following code is a possible instantiation of the previous pproc template:

```
let NilProc : pproc = nil;
let B : bbproc  = #(z, Z)
   [ P<<NilProc, y, z, Z2>> | y?().nil ];
```

The code generate by the compiler as the result of this instantiation is equivalent to the following hand-written code:

```
let NilProc : pproc = nil;
let B : bbproc  = #(z, Z)
   [ x?().y!().ch(z, Z2).NilProc | y?().nil ];
```

More precisely, a template is instantiated by using the Id of the template (pproc or bproc) and providing it with a list *invTempList* of comma-separated template invocations *invTempElem*s, whose kind has to match the kind of the template formal parameters.

invTempElem ::=
- *Id*
- | *Id* ⟨⟨ *invTempList* ⟩⟩
- | (*Id*, **unhidden**)
- | (*Id*, **hidden**)

invTempList ::=
- *invTempElem*
- | *invTempElem*, *invTempList*

bp ::=
- ...
- | *Decimal* ⟨⟨ *invTempList* ⟩⟩

Note that templates do not increase the expressive power of the language, they only make it easier to write generic and reusable code. Consider the following code:

```
template rep : pproc<<name x, pproc P>> = !x?().(P.nil);

template detach : pproc<<name x, prefix P, binder T, name y>> =
   x?().P.ch(x, UN).hide(x).ch(x, T).unhide(x).y!().nil;
```

The first template is the general pattern of a replicating process, that performs some actions and then gets back to its original state. The second template is the general pattern of an entity that waits for a signal on a binder, responds by performing some action and then forces an unbind.

Enzymes that catalyse a reaction with a substrate and then detach from it can then be written as follows:

```
let E1p : prefix = delay(rate).p!(). ... ;
let E1p : prefix =  ... ;

let E1 : bproc = #(p, TyrDomain) =
   [ rep<<y, detach<<p, E1p, TyrDomain, y>> >> ];
let E2 : bproc = #(q, XYDomain) =
   [ rep<<r, detach<<q, E2p, XYDomain, r>> >> ];
```

The programmer has only to define the prefix that codifies for the response ($E1p$ and $E2p$), without having to worry how to write code for forcing the detachment of the substrate.

Appendix C

Graphics Processing Units

Many scientific applications -medical, physical, mechanical engineering- have obtained clear benefits from visualization techniques and boosted research and technological innovation in Computer Graphics.

Recently, however, development in Computer Graphics has been driven by commercial applications (videogames, movies, etc.). The wide adoption of three-dimensional graphics in videogames, for example, pushed innovation in graphics technologies at an astonishing pace giving us a new, powerful kind of processors: GPUs. Fortunately, this fact had a positive effect on research: the continuous evolution of graphics hardware and economies of scale made powerful hardware accelerators readily available to single researchers.

In order to better understand the methods and algorithms used Chapters 4 and 5, where we will use GPUs to perform high performance computations and visualization of volumetric data respectively, this appendix introduces GPUs, their design rationale and their architecture. These concepts are applied in the aforementioned chapters to the design of algorithms for spatial visualization and for parallel processing of stochastic simulations.

C.1 GPU history

A Graphics Processing Unit (GPU) is a processor designed and built to accelerate the computation of graphics operations. Specifically GPUs are found inside graphics boards, where they are used to speed up 3D graphics *rasterization*. *Rasterization* is the task of taking an image described as a series of shapes (usually triangles) and converting it into a raster image (a bitmap, i.e. a two dimensional array of pixels -colour data-) for output on a video display.

The term GPU is often used in contrast or comparison with CPU (Central Processing Unit), the main, *general purpose* processor at the core of every computer, designed and

built to execute efficiently a variety of algorithms. In fact since the onset, a GPU was a highly specialized processor created to deal with a specific set of operations in the *rasterization pipeline*. The set of operations was limited but, thanks to their specialized implementation, these operations ran extremely fast on the GPU.

To understand how a GPU works, why the architecture of today's GPU is shaped in this way, and why the GPU can execute extremely fast only some kind of programs we need to introduce the *3D rendering pipeline*, i.e. the set of instructions that a GPU was designed to do.

This pipeline is the set of graphics operations done on triangles and points in a 3D coordinate space to transform and project them to a 2D surface (the screen) and then on the image pixels to colour and lit them (see Figure C.1). The input of the 3D graphics

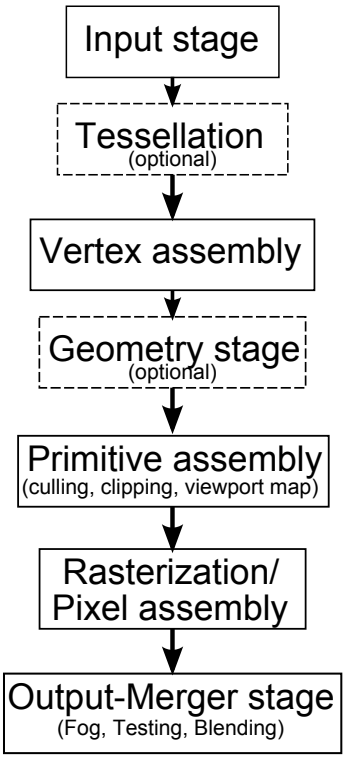


Figure C.1: The 3D rendering pipeline

pipeline is a set of *vertexes*, 3D vectors used to describe the position and orientation of shapes (usually triangles meshes) inside a three dimensional world. A first optional stage (*Tessellation*) tessellate the shapes to create a mesh made by only triangles. It optionally increases the number of triangles in order to create a more refined geometry. Then, in the *Vertex Assembly* stage, vertexes in the input mesh are processed and transformed, mapping the scene to the point of view of the camera (the "virtual" position of the user). If necessary, normal vectors used later for lighting and culling are computed. After

this phase, vertexes can undergo further combination or creation during the *Geometry stage*, before being assembled in primitives (usually triangles). During this stage (called *Primitive Assembly*), vertexes can be discarded for three reasons: 1) viewport mapping (vertexes are out of the field of view); 2) culling (vertexes are part of the back of a shape and therefore not visible); 3) clipping (vertexes belong to an portion of space explicitly set as empty by the application). Finally, primitives are projected on a 2D surface for rasterization in the *Pixel assembly* stage, where per-pixel data (such as colour) is generated using lighting, texture and depth information. The image is then composed and send to the output buffer.

The *Vertex assembly* and *Pixel assembly* stages are of particular interest. Both of them work on sets of four values: four floating point values for vertexes (representing points or vertexes in 3D homogeneous coordinates), four integer values for pixels (representing the red, blue, green colour components plus an alpha value for transparency). Also, both of them are very expensive from a computational point of view. Therefore, using SIMD (Single Instructions, Multiple Data) operations to accelerate them seemed a reasonable way of speeding up 3D graphics.

The first GPUs accelerated the Vertex assembly and Pixel assembly phases (known at the time as "Transform & Lighting"), using some standard, fixed algorithms (i.e. Gouraud Shading for lighting). Later generations started to allow some programmability, first for the Pixel assembly phase (introduced by Nvidia with *registry combiners*), then for both vertex and pixel assembly steps using *shaders*.

A *shader* is a set of software instructions used to program some stages of the rendering pipeline, substituting the standard, fixed algorithms previously used. With shaders, customized effects can be used. The shader instruction set and assembly language was very limited at the beginning; for example, loops and branches were not possible, value types were limited -floating point for vertexes, integers for pixel colours- and the number of instructions for each shader program was limited to tens of instructions. Also, pixel and vertex shading instruction set were different. Every new generation of hardware, however, the shader model was refined and more features were added: GPUs become powerful enough to be able to perform all the stages of the pipeline without the help of the CPU, and even more, thanks to the programmability enabled by shaders. Starting with the fourth version of the *shader model*, pixel and vertex shaders were unified. The latest version of the shader model added support for geometry shaders (use of the same unified shaders to program the geometry stage). Limits on program length are now only practical; there is support for loops and branching (although with a performance penalty, as we will see later), for atomic functions and many other capabilities that make them easier to use with general purpose computations (e.g. not strictly related to rendering).

C.2 GPU computing

GPU computing started as an effort from the scientific community to exploit the raw processing power of GPUs to make scientific computations. Two GPU generations ago, with the G80 series of processors and the introduction of unified shaders, graphics processors manufactures started to see the potentiality of GPUs in High Performance Computing (HPC). Today, GPUs are marked and advertised from vendors as cheaper and more performant alternatives to small traditional clusters.

Indeed, the raw power of the more recent GPUs is comparable to the computational power of a cluster with roughly a thousands of CPU cores. However, due to the nature of GPUs, this power can be exploited by only a few specialised algorithms.

C.3 GPU architecture

In general, the architecture of a GPU is tailored to 3D graphics computations. The characteristics of graphics computations (highly parallel, very high *arithmetic intensity*[1], simple stream of mathematical instructions executed on the same data types) dictated the design of GPUs: little or no cache at all, a cluster of SIMD cores, and a memory with large bandwidth. Overall, compared to CPUs, GPUs are relatively simple: CPUs are designed to run a very wide variety of programs, even purely serial programs, as quick as possible, and therefore they package very complex logic and large caches. It is surprising to see how little space on a processor die (and consequently a low number of transistors) are used for actual computation. Consider for example the floor plan of a single AMD core in Figure C.2: only the area marked as *Execution Units* and part of the *Floating Point Unit* area are dedicated to the ALUs, the Arithmetic and Logic Units used in the computations.

GPUs, on the other hand, are very specialized: most of their silicon is used to perform arithmetic computations (see Figure C.3). The small area dedicated to the scheduling of computational resources and the absence of cache, however, do not prevent the GPU from being the ideal candidate for computations that present the same characteristics of 3D graphics: high arithmetic intensity, same operation applied to all (or to a big subset of) the input data, simple or no branching.

It is important to note that even if we can consider modern GPUs as massively parallel processors, not every execution unit (shader processor) in a GPU is a single core: each shader processor contains the digital circuits to perform arithmetic computations, but not the additional logic necessary to drive it as separate unit[2].

[1] the ratio of computation to bandwidth, or more formally *arithmetic intensity = operations / words transferred*

[2] To some extent, this applies to standard CPUs as well: each core have several execution units (or ALUs);

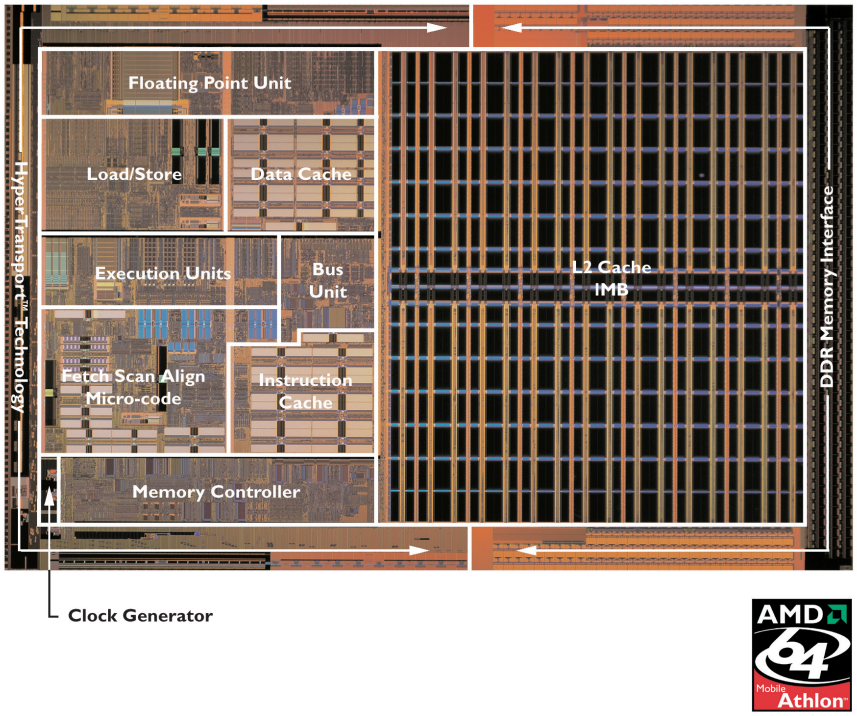


Figure C.2: Marked die plot for a single AMD core. Notice the small area devoted to actual computation ("Execution Units" and part of "Floating Point Unit"). Note the large area devoted to level 2 cache. The situations is even more extreme in multi-core processors, where often an additional level 3 cache, shared by all the cores, is present.

The architecture details vary from vendor to vendor, and sometimes even from one model to another. In this thesis we will focus on the NVIDIA GPU architecture [147], as it is the most used for GPU computing. NVIDIA was the first one to address specifically GPU computing with the introduction of CUDA (Compute Unified Device Architecture) [159].

CUDA GPUs are organized in multiprocessors, which group multiple streaming processors, the basic execution units (see Figure C.4 and Figure C.6). CUDA executes the same program on all the multiprocessors: the code for the program (kernel) is the same but both the data and the execution flow can be different and diverge. CUDA launches multiple instances of the same kernel, called threads. Threads are grouped in warps (see Table C.1) for execution on a multiprocessor.

Threads are runtime instances of the same kernel, and therefore they execute the same

up to six on mainstream processors. All the ALUs perform computations in parallel; however the instructions issued to these units, and how computation is divided and scheduled on them, is internal to the processor core and completely transparent to the user

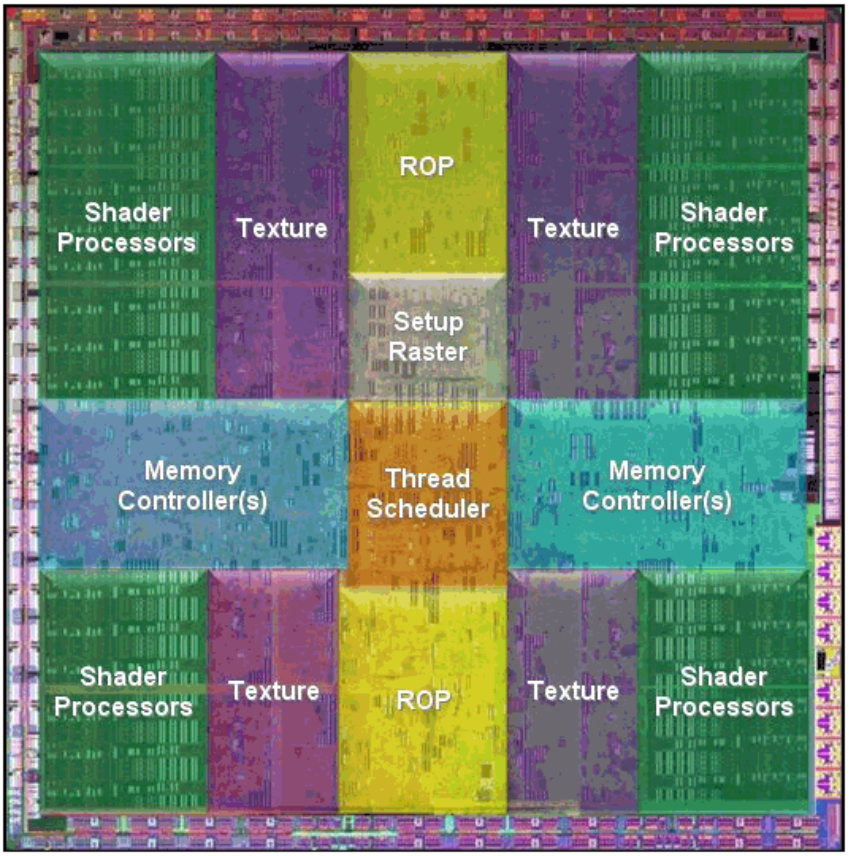


Figure C.3: Marked die of a nVidia GT200 GPU (image courtesy of Nvidia). If you don't consider die areas devoted to graphics operations (Texture units, Raster and ROPs) the computing area (Shader Processors) clearly dominates.

Device/Host	GPU/CPU
Kernel	Function called from the host, executed on device. Kernels are executed one at time, by many *threads*.
Thread	Instruction stream flowing into a single execution unit. Note that they are not like CPU threads. For example, context switch is free.
Warp	Set of threads, currently 32 threads. The Warp is the scheduling unit (one warp is scheduled on one multiprocessor)
Block	Set of threads that cooperate via shared memory.
Grid	The "structure" on which blocks of threads are launched (Only a facility for decomposing your domain, for having threads that access different parts of your data).

Table C.1: CUDA terminology

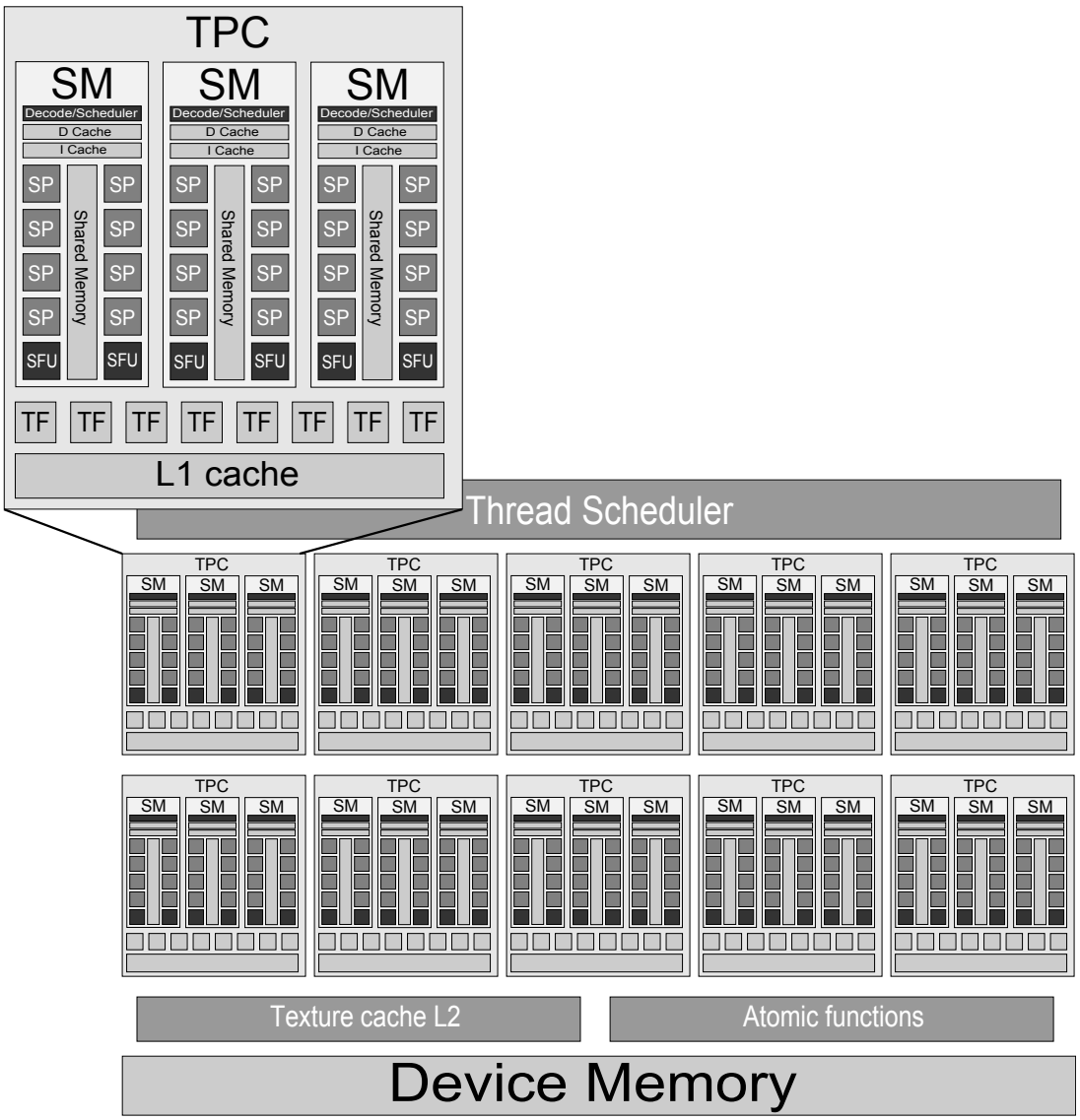


Figure C.4: The structure and computing resources of a Nvidia GT200 chip. Notice the 10 processor clusters, each containing 3 Multiprocessors

program code; furthermore, all the threads in a warp are executed by one multiprocessor in an SIMD fashion, and therefore they must execute exactly the same instruction at the same time, although on different data. If threads diverge (taking, for example, different branches of an if statement), they will be split into different warps, leading possibly to under-utilization of the multiprocessors. These restrictions help in keeping the architecture simple but powerful: thanks to the big amount of silicon allocated to arithmetic operations, the raw power of GPUs is enormous.

Applications that process large amounts of data or objects, and perform the same op-

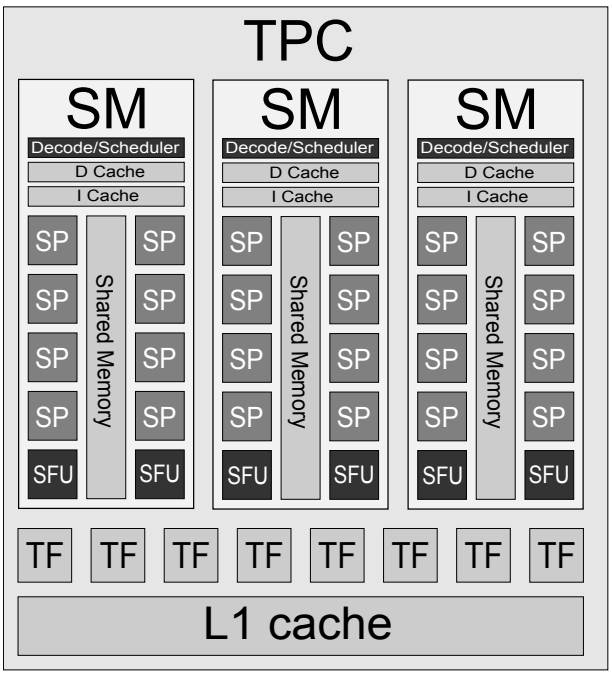


Figure C.5: The GT200 TPC, containing 3 Multiprocessors

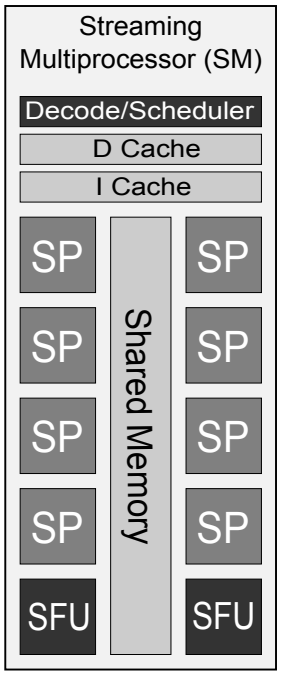


Figure C.6: A Nvidia Streaming Multiprocessor (SM), with its own Instruction Unit and 8 Streaming Precessing Unit (SPU)

erations on all of them, will fit nicely on a GPU: to keep all the streaming processors busy, and therefore to obtain good performances, tens of thousands threads need to be executed

concurrently. Therefore, applications based on the execution of disparate, short tasks will lead to the fragmentation of warps and to the under-utilization of multiprocessors. Similarly, the applications that process a small subset of data at each time will fail in keeping the streaming processors fed with enough data. Finally, applications requiring double precision floating point numbers are currently severely limited: the support for double precision was added only in the latest generation of GPUs, and in a reduced way. For example, on NVIDIA GPUs only one streaming processor for each multiprocessor is capable of operating in double precision; this leads to performances that are at best one eight of the single precision performances. Double precision is very important in some scenarios; in Monte-Carlo simulations and in numerical integration single precision is sometimes not enough. Fortunately, the next generation of GPUs will enhance significantly the support for double precision operations [72].

C.4 Programming a GPU

The first GPUs where programmed by submitting a string containing the shader program to the GPU driver through a graphics API like DirectX or OpenGL. Later, C-like higher level languages (HLSL and GLSL) were introduced, making the overall programming easier. However, these languages are still targeted to 3D graphics applications: the code had still to be submitted explicitly to the GPU via graphics API calls, data had to be mapped to graphics concepts and moved explicitly (sometimes inefficiently) back and forth from the GPU to the central memory, again using counter-intuitive graphics APIs. With the advent of GPU computing, several other languages or libraries were introduced; the latest example are Brook [7], OpenCL [8] , and CUDA.

The term CUDA usually refers to both an architecture and its associated programming model. The CUDA GPUs are programmed through an API and a set of C language extensions. CUDA embeds the GPU code inside C++ code, using the language extensions to indicate whether a function should be executed on the CPU (called "host") or on the GPU ("device"). It is therefore independent from graphics libraries.

All the details about threads, warps, multiprocessors, etc. are hidden from the end user; CUDA instead exposes the notions of blocks, grids and threads (see Table C.1) to ease the decomposition of the problem domain. As depicted in Figure C.7, threads are both the "physical" and "logical" basic unit of execution; the GPUs groups and schedules threads in warps, while CUDA offers an higher level view of grids and blocks. Grids and blocks can be used by the programmer to map the subdivisions inherent in the problem domain (in particular, spatial subdivisions) in a convenient way. Each thread is then provided with variables representing the block and grid coordinates on which it needs to

operate. Each thread can then access and process a single item or subset of the problem domain.

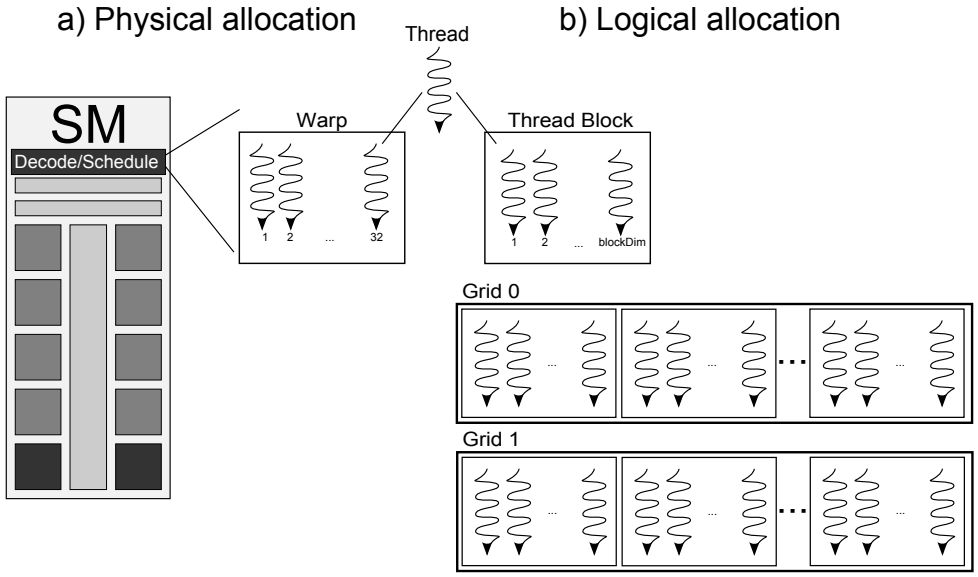


Figure C.7: The structure and computing resources of a Nvidia GT200 chip. Notice the 10 processor clusters, each containing 3 Multiprocessors

As an example, consider the simple and common scenario of porting computationally intensive loops to the GPU. In order to enable efficient execution, loops have to be transformed, strip-mining or unrolling them. After unrolling each thread executes a single, distinct iteration of the original loop. For instance, Table C.2 shows a simple algorithm that takes a vector 'a' of length 'N' and a value 'b' and increments each value of 'a' by 'b'. As expected, the sequential algorithm on the left accesses the elements of 'a' one by one. Instead, the kernel code on the right spawns 'N' parallel threads, each of them incrementing a single value of 'a'; the position in the array 'a' that the thread T has to increment is obtained by multiplying the block index of T (blockIdx.x) by the number of threads per block (blockDim.x) and finally adding the current index of T within the block (threadIdx.x).

```
//CPU code
void increment_cpu(float *a, float b,
   int N)
{
   for (int idx = 0; idx<N; idx++)
      a[idx] = a[idx] + b;
}
```

```
//GPU code
__device__
void increment_gpu(float *a, float b,
   int N)
{
   int idx = blockIdx.x * blockDim.x +
             threadIdx.x;
   if (idx < N)
      a[idx] = a[idx] + b;
}
```

Table C.2: Plain C code versus CUDA code for implementing a simple algorithm. Notice how the loop is unrolled; calling the kernel on the right will require spawning N threads, each of them incrementing a single item