Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Biological modeling and simulation as a service

Tesis de Licenciatura en Ciencias de la Computación

Laouen Mayal Louan Belloli

Director: Gabriel Wainer

Buenos Aires, 2019

BIOLOGICAL MODELING AND SIMULATION AS A SERVICE

Modelling allows us to focus on the important components of a system under study, leaving aside the non-meaningful information. To study metabolic networks, we need first to create a new model of the phenomena and then, we are able to start simulations to obtain results. A major problem in biological cells is the lack of generalization of the proposed models, each model is created to study a specific aspect of a biological cell, or to study a specific cell. Because of this, it is hard to reuse them in other works. In this thesis, we try to apply some computational concepts as modularization and model integration to improve the model integration and automation of biological cell processes. For this purpose, we propose a general model of a cell structure to use as a model framework where different models can be integrated. We also propose a stochastic genome scale model of a cellular metabolic network that is general and can be used to study the metabolism of different cells. Because a genome scale model implies having thousands identical atomic models, we introduce the concept of multi-state models that allows merging all the identical atomic models in a single one. We also propose a method for automatic modelling and simulation of biological cell processes using SBML files as the input and a web platform for remote model and simulation that can improve the collaboration between different research groups and the research time.

Keywords: DEVS, P-DEVS, Modelling, Simulation, Biological cell, Metabolic pathways, Cloud services, Metabolic network, Stochastic models, Model integration.

AGRADECIMIENTOS

No voy a decir que la meritocracia es la culpable de esta tesis porque sería hipócrita de mi parte ignorar mi punto de partida y a todas las personas que por algún motivo se cruzaron y se siguen cruzando en mi vida y que afectaron y siguen afectando radicalmente mi forma de pensar, mis oportunidades y mis capacidades. Es por esto que quiero agradecer profundamente a aquellas personas culpables de que yo esté en este momento y lugar presentando una tesis independientemente de la calidad del trabajo presentado. Desde ya el orden en el que aparecen los agradecimientos no responde a nada más que tal vez el orden cronológico en el cual entraron a mi vida las distintas personas nombradas.

A mis padres Daniel Alberto Belloli y Françoise Yvonne Poilblan por haber dado lo mejor de ellos como padres y esperando haber podido recibir lo más posible de esa genial educación que me abrió la mente a querer explorar las maravillas de la existencia humana, o al menos lo que mis capacidades me permitan explorar.

A mi hermana Maeva Ayelen Belloli por haber compartido la mejor infancia que puedo imaginar y por seguir ahí para mi cada vez que la necesito, como cuando recién llegué a la ciudad para empezar mi carrera y tuvo que soportar a un ser en transición medio año dejando todos los platos sucios, o como cuando me recibió en su pequeño estudio en París por 3 meses (ya sin dejarle los platos sucios).

A mi tía Veronica Beatriz Belloli, una de las personas más maravillosas que conocí en este mundo y a la que le debo las mejores lecturas y consejos de vida que recibí, una persona con quien puedo hablar con total libertad y que cada vez que no veía la luz al final del túnel supo darme el empujón necesario para que apareciera.

A mi hermano de vida Juan David Czernochivsky, que desde el día que lo conocí se convirtió en mi hermano y está siempre ahí para liberar mi cabeza que por momentos piensa más de lo que debería. Gracias por todos esos fines de semana en Varela y Capital que hicieron mi transcurso por la universidad una experiencia aún más copada y por enseñarme a disfrutar de los pequeños placeres de esta vida que son tan importantes como los grandes planes.

A Vanessa da Matta Heeger por ser una compañera de vida única que compartió conmigo no sólo un departamento y una economía, sino también una cultura, un idioma y una forma de pensar. Es y será siempre una persona con la que me siento profundamente identificado y que me enseño que siempre se puede avanzar y que nunca es tarde para nada. Por sobre todo, gracias por ser la persona que sos.

A Daniel Hernandez-Baquero (alias el brother), por bancarme todos los años que compartimos mini cuartos, debo admitir que debe haber sido un trabajo arduo de su parte. Gracias por desestructurar mi rígida cabeza y por todas esas noches de largas charlas. Gracias por compartir conmigo tanta sabiduría y por bancar todos mis regaños.

A mis compañeros de cursada Federico Nicolás Landini y Pablo Agustín Artuso por haberme tirado de las riendas tantos años de carrera, a ellos les debo muchísimo más que casi todos los trabajos prácticos y mi ortografía. Ellos me enseñaron a confiar en mí y salir adelante en lo que me propongo, me enseñaron que si bien muchas veces las cosas no me salen tengo las capacidades de mejorar hasta lograrlo. También me enseñaron a ser humilde y crítico conmigo mismo, lo que considero una gran enseñanza académica y de vida.

A mi camada (Roberto Alejandro Rama Vilariño, Sacha Kantor, Franco Castellacci, Gabriel Defagot, Victor Wjugow y Gonza Lera Romero) por haberme incluido en su grupo aunque gran parte del tiempo entendiera los chistes medio año después o directamente no los entendiera. Agradezco haber cursado la carrera con un grupo de personas al que admiro mucho tanto por sus capacidades académicas como por sus ideologias de vida.

A Gabriel Wainer por darme la increíble oportunidad de ir a investigar a su laboratorio en Carleton University - Ottawa - Canadá. Recuerdo decirle que yo todavía no había realizado ningún trabajo de investigación y él me respondió que estaba listo para hacer el primero. Gracias a esa oportunidad no sólo descubrí un hermoso país, sino también aprendí el significado de llevar adelante un proyecto entero desde el comienzo hasta el final.

A Cristina Ruiz-Martín por compartir una de las experiencias de vidas que más me hicieron crecer académicamente, por darme el soporte necesario en mis primeros pasos en el mundo de la investigación que resultó en mi primera publicación y hoy en día en una tesis de licenciatura. Y por hacer de Canadá y ARSlab una experiencia más linda de la que ya era.

A Diego Fernández Slezak y Andrea Goldin por estar siempre ahí mentoreándome y abriéndome las puertas de sus laboratorios. Gracias por la paciencia eterna que tuvieron y siguen teniendo. Gracias a ellos pude realizar la mitad de mi carrera ya trabajando en laboratorios junto a investigadores extraordinarios aprendiendo todos los días más y más.

A los demás integrantes del laboratorio LIAA por integrarme al grupo y tratarme como un doctorando más aunque no fuera un doctorando. Gracias a eso hoy me preparo para emprender un doctorado con más confianza de la que tendría de otra forma.

Dedicado a aquellas personas que hoy no pueden estar acá y con las que me hubiera encantado poder compartir este momento: Mi tia Virginia Belloli, mi tío Marcelo Belloli y mi abuela Laura Sarandria.

Lokah Samastah Sukhino Bhavantu.

CONTENTS

1. INTRODUCTION

In the past decades, the need to understand biological processes has increased. In medicine, a better understanding of these processes allow improving drugs and treatments. In biology and biochemistry, biological cells and their subsystems play a fundamental role and their study is essential. One of the main phenomena occurring in cells are called "metabolic pathways", a series of linked reactions that produce transformations of different metabolites (metabolites are small molecules, like hydrogen or oxygen, that are either consumed or produced by chemical reactions in the cell). These reactions conform the metabolism of the cells and their life; therefore, the study of this phenomenon has become popular in different fields related to medicine, biology, and healthcare among others.

Metabolic pathways are the result of "metabolic networks", which are reaction sets where the metabolites produced by any of the reactions can be consumed by several other reactions, defining multiple possible reaction paths. All the possible combinations of reactions, the exchanging of metabolites and the generation of reaction chains define the metabolic network; each possible reaction path within the network is a metabolic pathway.

We will show an example of a metabolic network and its pathways; Consider three reactions A, B and C where:

- $A = d \rightarrow e + f$: Consumes metabolite d and produces metabolites e and f.
- $B = e \rightarrow g$: Consumes metabolite e and produces metabolites g.
- $C = e + f \rightarrow h$: Consumes metabolites e and f and produces metabolites h.

These reactions can generate several possible reaction chains as the next ones:

- $d \rightarrow e + f \rightarrow g + f$: First reaction A reacts and produces e and f, then the metabolite e produced by A is consumed by B and converted to g, finally we end up with metabolites g and f.
- $d \rightarrow e + f \rightarrow h$: First, reaction A reacts and produces e and f, then then reaction C uses those metabolites produced by A to produce the metabolite h.

These reaction chains in which the product of the reaction A is consumed by reactions B and C are the metabolic pathways of the metabolic network which is the whole system composed by the reactions A, B and C and all its possible pathways. It is worth to notice the mentioned pathways are only possible if the metabolite d is present in the network, because if d does not exist, reaction A cannot react, and the reaction pathways will never occur.

Metabolic networks have an exponential number of possible pathways generated by all the possible combinations of which reactions consumes the product of other reactions, but only a small number of those pathways will effectively take place in the cell. Due to this reason, studying metabolic networks is not easy [1] and we need to study the metabolic

pathways that will effectively take place without the need of considering all the exponential combinations of possible pathways, where the majority of them will never occur.

When studying metabolic pathways and many other biological processes, different techniques can be used. For instance, in-situ studies conduct the research in the real place where the phenomena usually occurs, without modifying its environment to ensure that the obtained results are free of any noise that can be introduced by the observation procedure. In contrast, in-vitro studies isolate the phenomenon from its original environment to a controlled one (normally in a laboratory), this allows to control all those aspects of the environment that are not part of the phenomenon itself but still can affect the results in ways that we need to control.

A different approach is to create a model of the phenomenon that allows us to study a mathematical entity (or an informal one) instead of studying the phenomenon directly. Another method is to use simulation to get measurable results.

Observation is the most direct approach to study biological phenomena, and it remains very important as the first mechanism to understand a new phenomenon, and as the way of validating the obtained results of other methods. Likewise, observation allows to measure different properties of the phenomena that are unknown. The phenomena we are interested to study are happening in nature; for example, a biological reaction is a phenomenon where certain metabolites are bound, and new metabolites are created from them; the reaction rate is a property that indicates the rate at which the consumed metabolites are converted into the produced metabolites. The reactions and the reaction rate value of a certain reaction can only be known by observation.

Even if observation techniques as the in-situ or in-vitro can be used to study a phenomenon in the most direct way, they are expensive and time-consuming. In-situ experiments could not be possible to conduct due to the inability of studying the phenomenon without affecting its environment, the impossibility to access its environment or the lack of tools to conduct it. When in-situ experiments are not possible, in-vitro studies could be an alternative and the phenomena are taken from their natural environment to the laboratory, but those studies can also be expensive and time-consuming making them not the best option.

Considering these issues, in-situ and in-vitro experiments are usually replaced or preceded by the creation of models that allows us to describe the phenomenon without the need of observing them each time we need to know something about them. Modelling is less expensive than observation, and it is also less time-consuming, allowing us to easily conduct several experiments without significantly raising the cost or times of the research.

Modelling can be achieved only by abstraction; this allows us focusing on the meaningful components of the system under study, leaving aside the non-significant information of the process. Simplifying the phenomenon is effective not only to isolate the important part but also to control the effect of the abstracted components in the phenomenon and therefore, allowing to construct several scenarios in order explore the different properties.

Sometimes we cannot solve the models analytically; an alternative way to study the models is by running computer simulations of virtual scenarios to obtain simulated results based on the models. Simulation is the process by which we use the specified model and some specific inputs to predict the behaviour of the real phenomenon. Nowadays, with the aid of computers, we are able to automate the simulation process by using algorithms that allows us to run several computational experiments reducing the experimental times and effort. The entire process of creating phenomena models and running simulations of those models is called Modelling and Simulation (M&S) techniques.

There are different approaches to model and simulate biological phenomena; the differences between those approaches mainly remains in the abstraction level made (micro-views vs macro-views models), the formalism used to describe the properties of the phenomenon (mathematical vs computational models) and the time representation used for the models (continuous vs discrete time-based models). Next, we will explain these differences.

Macro-view models describe the behaviour of the entire system as a single piece, abstracting all sub-systems and their interactions, examples of this include models using Ordinary Differential Equations (ODEs). On the other hand, micro-view models first separate the entire system on sub-systems that can be modelled as different phenomena and after that, they are combined to obtain the final model. In micro-view models, the result is the composition of all its sub-models results.

Mathematical models are purely analytical; in these models we solve equations and we use those results. Mathematical models are so far the best approach for deductive analysis because we can apply all the mathematical language to make deductions from the model, but it can be hard to integrate them into other models or systems, and in many cases the solutions cannot be found. On the other hand, computational models are programs where its logic describes the phenomenon to be modelled, thus, we are able to make rule-based models. These models facilitate the study of systems at different levels of abstraction, improves the model integration and the study of its sub-systems. In the last decades, with the significant increase of computational power, the number of these models has also increased and now we can simulate complex systems at larger scales.

Continuous time-based models use any time representation system that allows to model values for any time amount regardless of how small could potentially be. Discrete time-based models use time representation that only allows representing some values of the continuous timeline. This leads to restrictions in the model time steps, because events should only happen in valid times.

As mentioned before, models are abstractions of real phenomena, and one of the most significant abstractions made when modelling is their environment and the interaction with other phenomena. Different phenomena can be modelled using different techniques that best fit their needs and combining those models also requires combining their modelling techniques obtaining hybrid models. For this purpose, modularization helps in avoiding problems in model integration. One way of doing this is considering each model as a black box that receives inputs and returns outputs; subsequently, combining different models only requires the models to agree in their input and output semantics but they do not

need to have any knowledge about the other models.

Considering the vast amount of different techniques and models existents in biological processes, we aim to achieve a more general solution proposing a flexible structure to easily integrate different models and approaches. We have considered the different aspects of a general structure, discussed following.

Biological cells are composed by multiple mechanisms that play different roles:

- cell signalling is the communication process that relates the cell with its environment and directs the basic actions of the cell.
- gene regulation is the process that controls the production of genes within the cell; a gene is a sequence of DNA or RNA that carries the cell information used to create new equal cells.
- metabolic pathways (the cell metabolism) is in charge of consuming and producing metabolites to generate energy for the cell.

Commonly, these processes are modelled using specific models that consider only particular instances of a single part of the cell. Examples of this can be found in [2][3]. These are rigid models that are hard to reuse as part of more complex models.

In this work, we are interested in modelling different biological cells and because of the all the differences that exist in between cell we are not able to consider a single rigid model structure for all of them. Thereby, instead of trying to do a perfect model of every biological cell, we aim to achieve a flexible structure model that helps researchers by offering a general framework that can easily be improved and adapted for different projects, allowing them to integrate different models if needed. This general structure must consider only the most general aspect off cells while allowing to add and remove external components that describe the particularities of cells so modellers are able to integrate their models of the different mechanism as the cell signalling and the gene regulatory into the general structure. Also, this general structure aims to correctly modularize the physical parts of a biological cell so each new model to integrate can be modelled using a different approach without affecting the rest.

We focus in the metabolic pathways of the biological cell using a stochastic model to study collisions and binding processes instead of modelling the entire network (which, as said, can have exponential explosion of states). In this way, the network becomes an emergent property of the simulation. The metabolic pathways are an essential part of every cell and thus we propose this system as part of the general model structure framework. However, it is not required to use our proposed stochastic model, and the structure of the simulation framework can be reused by replacing this component by a new one, or by adding more components if needed.

We propose a structure of a biological cell. This structure works as a container where to integrate different models of those mechanisms, allowing to reuse the general structure as a model integration framework.

The proposed structure focuses on the different physical compartments of a biological cell such as the periplasm, the cytoplasm and organelles (separations in the cell structure that have some level of isolation, where different processes occurring within each compartment are independent of processes occurring in other compartments). Also, these compartments are common structures in all biological cells and abstracting them is a goal in this research. Compartments naturally define a modular and hierarchical separation of the different physical aspects of the cell and we focus on that modularization to allow a better model re-utilization because each different compartment is almost independent of the rest, modellers are able to focus on each compartment separately and then combine all the compartments. Also, this modularization allows to easily replace a compartment model by a different model and models can be reused.

If models can be well defined as the integrations of multiples sub-models, collaboration is facilitated through the models sharing. This is important when dealing with complex systems where each modeler can focus on a particular mechanism of the cell. This also simplifies model validation because each sub-model can be validated independently of the rest.

Model integration can only be made if their integration is flexible enough to support their differences even when they were not initially considered, and this is not easy to achieve. On other hand, cells are very complex systems with several sub-systems and making an entire computational model could easily lead to a model impossible to validate, hard to understand, maintain and modify. As far as we know, such model does not exist to date.

We propose a model structure that takes into consideration the modular and hierarchical organization of biological cells, for this purpose we use DEVS (Discrete EVent Systems Specification) [4][5] which is a modelling formalism where modularization and hierarchal organizations are natural to define. Because of this, we are able to define a model structure that maps in a very direct way a real biological cell structure where each compartment is defined as a separated PDVES model and integrate them through links that send messages, resulting in the final model that emerges from the interaction between its sub-models as in real cells.

On the other hand, this proposed structure does not assume anything about how each sub-model is defined and it only requires their inputs and outputs to have the same semantic, as a result, the model structure becomes flexible enough and new components can be added if their message interface is consistent with the existing components' interface, and, if this is not the case, the interface can be adapted.

Proposing a general model structure also allows us to improve the modelling and simulation process. We propose to a technique that speeds and automatizes the process of developing metabolic network models developing a web platform for model generation and simulation of the biological process as a service.

As the internet has become a powerful tool not only for communication purposes but also as a service provider, cloud computing allows people of different geographical locations to work together in the same project without the need of complicated communication

channels or local computations on their workstations. Instead dedicated servers can store the shared projects and run all the necessary computations to return the required results improving the working flow. Databases are also a way to share any kind of information as for example the exchange of models, for this, first we need to define how models will be stored in those databases, one solution is to use SBML files to store them and then, implement programs that can automatically parse and generate the corresponding models from them.

The System Biology Markup Language (SBML) is an XML format for biological systems that can be used to store and exchange biological information [6]. XML was invented as a web standard for information sharing. Because of the popularity of XML today there are several XML parsers that can read SBML out of the box. Thus, it is a flexible, standardized and well-known format. SBML has become a standard in biological systems and different tools use it to read and save their models in databases [7][8]. Because of the growing popularity of SBML, plenty of computational tools for biology systems use it and several research results are stored as SBML; We can see some examples in [9][10][11][12].

Then, the entire working flow starts with the utilization of SBML files to statically describe models and as the way to save them in files that we are able to upload to the server where they will be used to generate the specified model. Since SBML is XML-format and XML was invented as a web standard for information sharing, there are plenty of libraries that facilitate working with them.

SBML files describe the models particular structure and its parameters, but they do not describe any computational rule and because of this, the model dynamic behavior is not described and we cannot automatically generate the entire models from the SBML files. Instead, we propose a model generation mechanism that has a parsing stage to consume SBML files as the input to instantiate the proposed general structure model with its metabolic pathways ready to run simulations and obtain results.

Finally, we have integrated all the mentioned parts of this work in a web platform to facilitate the access as an online service that is deployed in a remote server available for researchers with limited experience in programming, for this, we have implemented a Django [13] web platform with a client-side user interface that interacts with the model generation and simulation module. The back-end is in charge to manage uploaded SBML files, run the model generator module to generate models from those SBML files and run simulations saving the results in a MongoDB distributed database. On the other hand, the front-end user interface allows the users to visualize the uploaded SBML files, add new files, generate models from those files, run simulations and visualize the results in real time.

The micro-view level model proposed in this work is composed of a number of submodels that interact with each other. Although this approach is good to achieve flexible and reusable models, it introduces a considerable overhead in the simulation algorithm, the exchanged messages needed to communicate. We need a fast simulator able to compile and execute the model within a reasonable time.

We used Cadmium, a P-DEVS library. Cadmium has good run-time performance

because it uses meta-template programing to pre-calculate in compilation time all the static parts of the model structure that other simulators recalculate every time they need them. We modified the library using type-hiding (we use a general interface and then pre-calculate all the static structure in derived classes that implement the interfaces and are easy to compile).

The contributions of this thesis, are:

- DEVS general model structure to use as a framework where different biological models can be integrated.
- A model of entire cellular metabolic pathways from metabolic networks.
- An adaptation of the experimental Cadmium simulator to improve compilation.
- A web platform that automates the process of parsing, model generation and simulation using SBML files as the input, the web platform also implements visualization modules to show the simulation results in real time to the users.
- A really micro-view model of metabolic pathways as an internal cell mechanism integrated into the proposed general structure.

We focus in the metabolic pathways of the biological cell using a stochastic model to study collisions and binding processes instead of modelling the entire network (which, as we have said, it is computationally expensive to fully explore). In using stochastic processes, the metabolic pathways become an emergent property of the simulation. Metabolic pathways are an essential part of every cell and thus we propose this system as part of the general model structure framework. However, it is not required to use our proposed stochastic model, and the structure of the simulation framework can be reused by replacing these components by a new one, or by adding more components if needed.

This work aims to introduce a general structure that not only improves model integration but also propose a direction that could help to model an entire biological cell which as far as it has not yet been achieved. Using a hierarchical structure, we are able to model each cell mechanism as a separated and simpler component. Then, we incrementally integrate those components and eventually, we can obtain a computational model of an entire biological cell that is easy to understand, maintain, modify and validate. This process also facilitates backtracking when some sub-model is not working. Different approaches can be used in each sub-model, leading to different combinations of integrated models. For example, Cell-DEVS [14] could be used to define the spatial model of the bulk-solutions while the rest of the components remain as DEVS models.

The following sections are organized as follows: chapter 2 introduces the project background, including related works and the used DEVS formalism. Chapter 3 presents the proposed theoretical model and explain some important concepts and problems related to the phenomena under study, and we show how we have addressed these problems. Section 4 introduces the model architecture. Section 5.1 discussed the web platform. Section 5.2 introduces the parsing and model generation processes that allow automating the model

generation from SBML files. Section 6 shows the work we did to improve Cadmium compilation time. Finally, section 7 introduces some validation results.

2. BACKGROUND

In this section, we will review existing works in the area of metabolic pathways, in particular computational models. First, we will introduce some examples of different models of metabolic pathways; next, we will review research on automatic modelling and model integration; finally, we will review how the improvements made in computational biology have been applied so far in the field of M&S of biological cell systems, including an explanation about the DEVS formalism.

2.1 Related work

As we have already explained in section 1, metabolic pathways are a series of linked reactions that produce transformations of different metabolites. These reactions conform the metabolism of cells. Each reaction consumes a set of metabolites called the reactant and produces a set of metabolites called the product. When a reaction can also go in the opposite direction (consuming the product and producing the reactant), it is said that it is a reversible reaction.

In the Introduction, we have also explained the history of metabolic pathways research, from the expensive in-situ/vitro studies until current computational M&S approaches. We have also discussed the differences between macro-view models, which describe the behaviour of the entire system as a single piece; and of micro-view models, where the entire system is first divided into subsystems.

The implementation of computational models is relatively new in the field, and therefore, some important computational concepts, such as modularization and reuse have not yet been applied extensively to the modelling of biological processes.

M&S of biological cells is an important field and several models were made to study different aspects of them. For example: In [3] a micro-view DEVS model of the cellular metabolism by mitochondria is proposed; the mitochondrion is an organelle commonly found in eukaryotic cells and play an important role in different diseases as for example diabetes and deafness. In this work, the authors center their attention in the particular metabolism cycles of glycolysis and Krebs cycle occurring within the mitochondria and their model is helpful for that purpose, but being able to reuse the model to study the mitochondria metabolism effects in different biological cells could be helpful for other studies. In [15] a macro-view mathematical model is proposed to specifically study the glycolysis metabolic pathway independently of the mitochondrion. This model is shown to be accurate according to experimental results, but it could be generalized if we abstract the specific aspects of the reactions (product, substrate, etc) to allow modelling other reactions as well. By generalizing the model, we are able to reuse it to model other metabolic pathways.

The glycolysis pathway modelled in [3] and [15] is shown in Figure 2.1, its main role is to converts glucose into pyruvate, and it releases energy during the process. As shown

in Figure 2.1, the reactions involved in the glycolysis are catalyzed by enzymes (the arrow labels), and most of them are reversible (double arrow).

As seen in the previous example, biological processes can be modelled at different levels depending on the requirements of the model. In [16][17] the authors explain how combining macro and micro levels can bring advantages; the abstraction at different levels allows the models to describe each part in the correct level. In [16] an extension to DEVS called Multi-Level-DEVS (ml-DEVS) is proposed to support multi-level models explicitly. In ml-DEVS, the information at macro-levels can be accessed from micro-level components and vice versa. In [17] a discrete event multi-level model is used to study metabolite channelling, a system commonly modelled with differential equations at macro-view. Using a multi-level model, the authors address the problem that some parts of the modeled system are not continuous, while differential equation requires the system to be continuous. An example of not continuous systems are the metabolites in a compartment, which is a discrete amount.

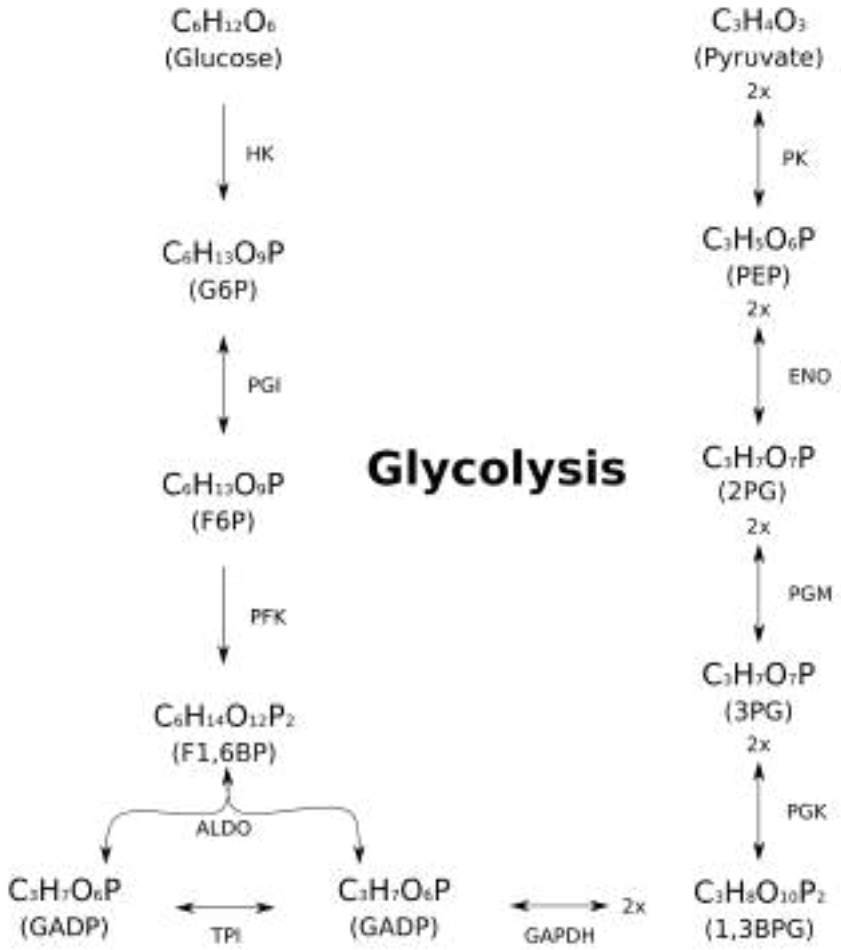


Fig. 2.1: Metabolic pathway of Glycolysis.

Using a visual representation of the models and simulation results allows us to understand the phenomena better, consequently several tools provide M&S visualization. Visualization of multi-level models is presented in [18] allowing researchers making detailed inspections by folding and unfolding the model substructures.

Biological processes are complex systems that easily leads to large models, and tools helping to automate some parts of the M&S process of these phenomena allows researchers to avoid dealing with the complexity of the tasks. We can automate different aspects of the process, for example: In [19] a methodology for detecting signal pathways from a source of data automatically allowed modellers to find signal pathways on large reaction networks that are hard to find by hand. Signal pathways are metabolic pathways that meet specific requirements, then, the authors proposed a weighted graph model of the metabolic network and an integer linear programming (ILP) optimization algorithm; the idea behind the IPL model is to find out a minimum-weight subnetwork (i.e. subgraph) of specific size which accomplishes the signal pathway requirements using the minimum possible number of biochemical reactions. Another work in automatic pathways detection is presented in [20], where an algorithm was introduced to study metabolic networks where some pathways are already identified, and one want to know other alternative pathways in the network.

A very important part of M&S that can be automated is the visual representation of both the model and the simulation results. Humans are very good to find patterns and results visually. For researchers without any background in computer programming, understanding a programming model is hard without visual representation. In [21] a framework for mathematical automatic modelling of signal-transduction through visualization was presented.

The amount of biological data saved in System Biology Markup Language (SBML) databases across the internet increases each day and as mentioned in [22], the implementation of systems automating the interaction with this data is important to manage a large amount of information to obtain relevant results that can be well replicated. Because of this, some recent works focused their attention on the automation for biological systems M&S using SBML as the data source. For instance, in [23], the SBML-DEVS platform introduced a framework for modelling reaction kinetics of biological systems using SBML to save and read the model's data. For this purpose, the authors used the LibSBML library [24] to translate the chemical reactions into differential equations and generate macro-view models that can easily be share automatically. Likewise, in [25] the authors presented a database system designed specifically to store biochemical pathways in SBML format. The system allows to reconstruct and analyze pathways from the stored data automatically.

On the other hand, because biological processes are complex systems that can be hard to address at once, it is useful to divide the entire system into several subsystems. Then, model each subsystem, and finally, integrate them. An example of this was shown in [26], where the authors created 28 independent models of the different subsystems of the bacterium Mycoplasma genitalium and integrated them into a single multi-level whole-cell model. By making 28 independent models, they were able to use the right tool to model

each subsystem and validate them individually. The use of flexible frameworks to integrate different formalisms and plug-ins help to find more accurate tools for each model and submodel, as done by JAMES II [27].

The integration of models can be easy or not, depending on the model's compatibility. In [28] the authors introduced CytoSolve, a method for integrating multiple pathway models implemented in independent source code. CytoSolve runs all the model's simulations in parallel, and coordinates their inputs and outputs generates a final single simulation result from the model interactions.

A general review of different computational tools for metabolic processes is shown in [29].

Not only the computational tools used for M&S need to be prepared for reusability and integration, but we also need to generate models prepared for reusability and integration. In [30] the idea of model reusability was discussed. In [31] the authors presented a discussion about some computational concepts as modularity and standardization applied for model re-usability. Finally, an emphasis on hierarchical modelling and composition in system biology was presented in [32].

Since spatial modelling in biological cells play an important role and different approaches are used for this purpose [33][34]. Model integration and reusability is a powerful tool to test different models of the spatial aspects of a biological process without modifying the entire model.

None of the mentioned works has proposed a model of a general biological cell structure, instead, they focused on different subsystems of cells [3][15][17] or in a particular cell [26]. If we aim to achieve a whole-cell computational model, we need a place where to integrate all the models of the cell subsystems. Modelling the structural aspects of a cell gives us a framework where to integrate the cell subsystem models. In order to allow re-usability of the model structure, we need a general structure abstracted from particular aspects of some cells that can be easily adapted to each particular case.

On the other hand, the continuous advances in computational science allowed improving the simulation processes of models at different levels. For example, the increase in microprocessor's speed allows us to run each day more complex simulations. Likewise, advances in networks and communication allow more fluent exchanging of model's information stored in databases. Recently, XML-based technologies have improved the exchange of remote messages on the Internet [35], and as we have already mentioned in [36], XML has also improved the way that models are saved and built. Having an automatic way to save and build models and a better information exchange facilitates the task of remotely share models.

A consequence of these advances was the definition of the System Biology Markup Language (SBML), which is now the standard for biology system representations using XML. It allows saving and exchanging models, and it is also useful for modelling visualization and validation [6]. The main advantage of SBML is the standardization of biology data

storage making easier to produce automated tools able to read SBML data from different sources.

Finally, in recent years we have seen some research on web platforms for biological structure automatic visualization, including, for example, MicrobesFlux [37], a web platform for drafting metabolic models from the KEGG database. Nonetheless, in most cases, M&S is normally executed locally in the laboratory, and as mentioned in [6] simulation result replication becomes hard for the rest of the community without the authors' setup.

2.2 P-DEVS formalism

Discrete EVents System specification (DEVS) [4][5] is a hierarchical and modular formalism for modelling Discrete Events Systems (DES). The hierarchical and modular structure of DEVS allows defining multiple submodels that are coupled together in a bigger model by connecting their inputs and outputs throw messages. In the same way, the resultant model can also be used as a submodel in a new coupled model with other models, defining a multi-level hierarchical structure.

In DEVS, there are two kinds of models: atomic models defining the system behaviour, and coupled models defining the system structure. Atomic models define the behaviour throw transition functions that move the atomic model from one state to another by following programmed rules.

States transitions are triggered when:

- External transition: The model receives external events from other another model.
- Internal transition: The lifetime of the current state is consumed and the model transition without any external event.

Coupled models provide modularity to the structure and their behaviour is the result of its submodels. DEVS models do not change their behaviour depending on the coupled model where they are used, the behaviour of any DEVS model is independent of other models, and it never knows what is occurring in the outside. This modularity is helpful for modelling biological systems, it allows using different modelling levels for each the submodel.

In the DEVS formalism, model's input receives messages from other models, and whenever two messages arrive at the same time, rules are used for selecting the order in which the messages will be processed by the model through a select function. Thus, the model makes multiple transitions in the same simulation time. Depending on the model, different messages processing orders could generate different final states after processing all the messages affecting the simulation result.

P-DEVS (Parallel Discrete Event Specification) is a modelling formalism that extends the well-known DEVS formalism. The extension is made on the fact that P-DEVS can handle simultaneous events occurring in the model and all the simultaneous events will be processed in the same transition function. Also, it adds a new transition (the confluence)

to customize the behaviour of collisions between internal and external events in the same model.

2.2.1 Atomic models

An atomic P-DEVS model is defined as an 8-tuple $< X, Y, S, ta, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda >$ where:

- X is the set of inputs event.
- Y is the set of outputs event.
- S is the set of sequential states.
- ta: $S \rightarrow R$ is the time advance function which is used to determine the lifetime of a state.
- δ_{ext}: $QxX^b \rightarrow S$ is the external transition function which defines how an input event changes a state of the system, where: $Q = \{(s, te) | s \in S, 0 \leq te \leq ta(s)\}$ and $X^b \subseteq X$.
- δ_{int}: $S \rightarrow S$ is the internal transition function which defines how the state of the system changes when an internal event occurs.
- δ_{conf}: is the confluence transition function which defines how input events change a state of the system if they arrive at the same time as an internal event is scheduled.
- λ: $S \rightarrow Y^b$ is the output function which defines the output of the system before an internal transition is triggered, where $Y^b \subseteq Y$.

The atomic model defines the behaviour of its system in the transition functions, a system starts with an initial state and remains in that state until an external event occurs or when the time advance is reached. Each time the system transitions from one state to a new one, it uses the transition functions corresponding to the transition event: External events will trigger the external transition, completed time advances will trigger the internal transition, and if both events occur at the same time, the confluence transition will be triggered. Once the transition function calculates a new state, the model transitions from the current state to the new state.

Always, before the internal or confluence functions are triggered, the lambda function is used to calculate the output to send to the outside. These messages will then be received by other models at the simulation time at which the transitions where triggered.

2.2.2 Coupled models

The P-DEVS coupled model is defined as a 6-tuple $< X, Y, EOC, EIC, IC, Z >$ Where::

- X is the set of inputs event.
- Y is the set of outputs event.

- EOC Is the External Output Coupling that defines links from sub-models output ports to the model output ports.
- EIC is the External Input Coupling that defines links from the model input ports to the sub-models input ports.
- IC is the Internal Coupling that defines links from sub-models output ports to other sub-models input ports.
- Z is the set of output to input translation functions, each function translates output events from a model to input events of the receiver model. These functions are commonly ignored, many simulators do not allow to specify this function and the input events are received with the same format as they were sent.

Coupled models are defined by multiple P-DEVS sub-models that are connected throw message links and connected between their ports, those links are specified in the EOC, EIC and IC components. The P-DEVS sub-models can either be, coupled or atomic models and the hierarchical structure of a P-DEVS model is defined by using other coupled models as sub-models and therefore, achieving different levels in the hierarchy and the hierarchical structure is constructed then, by the composition of several coupled models.

Ports are used to avoid broad-casting messages every time a model generates output. By using ports, messages can be directed to the corresponding receivers.

It is important to notice that for a coupled model, all its sub-models are black boxes, a coupled model does not discriminate between atomic and coupled sub-models and the global model hierarchy is unknown for the model and its sub-models. The only things that must be consistent are the inputs and outputs messages that the different components share between them.

2.2.3 Why we choose P-DEVS

In biological processes collisions are very common, metabolites collide with proteins and depending in the binding process mechanism the binding order is so fast that it is useful to abstract the model from the order in which metabolites were bound to the catalyzer protein.

P-DEVS handle message collisions throw message bags, so models receive multiple messages at the same time in a set of messages that are considered unordered. This property is very useful to model stochastic metabolic pathways where multiple reactions happen at any time and where collisions are an important part of the phenomenon.

DEVS coupled models have a static structure where sub-models are defined once and cannot be removed or added during simulation time. This property of the formalism could be a limitation in a biological system where the physical structures represented by models are destroyed and generated over time, for example, when compounds are bound to the catalyzers, they are destroyed and recombined into new compounds.

In this work we propose to use coupled components to model de cell structures as compartments and membranes, these structures are static and do not change over time and thus, there is no problem to use DEVS. All the components that play an active role and have an associated behaviour, are modelled in atomic components, but because we need to be able to create and destroy them, and to improve computational power, we do not use separated atomic models for individual components. Instead, we use multi-state models, a modelling concept we propose in this work and will be better explained later. In multi-state models, atomic models represent sets of equal components where states are collapsed to improve computational performance, and because the number of equal components is part of the model state, it can be updated in simulation time by a model transition function.

3. THEORETICAL MODEL

In this section we will introduce the theoretical concepts of the proposed model; first we explain how we separate the common and particular aspects of biological cells to achieve generic model classes that can be used to obtain instances for different biological cells. Second, we introduce the structure of a real biological cell and the proposed mapping between that structure and the general model structure, finally, we explain how we handle the high number of model components and links generated due the micro-view level modeling by using multi-state models and a routing system.

In biological cells, active components with common behaviours can be abstracted in order to create general model classes. A model class is a more general model that instead of modeling a single component of a biological cell, it describes those properties of the component that are common and parametrizes the particular properties that are not common. For example, Formula 3.1 describes the common behaviour of all non-reversible reactions of biological cells and Formula 3.2 shows a particular instance of the general model shown in Formula 3.1 (i.e. a model of a particular non-reversible reaction of a biological cell).

$$\begin{aligned} &\sum_{i=0}^{n} a_i s_i \rightarrow \sum_{j=0}^{m} b_j p_j \\ &a_i \in substrate \quad stoichiometry \\ &s_i \in substrate \\ &b_i \in product \quad stoichiometry \\ &p_i \in product \end{aligned} \tag{3.1}$$

$$\begin{aligned} &8Fe + S_8 \rightarrow 8FeS \\ &n = 2, m = 1 \\ &a_1 = 8, a_2 = 1 \\ &s_i = Fe, s_2 = S_8 \\ &b_1 = 8 \\ &p_1 = FeS \end{aligned} \tag{3.2}$$

The general equation shown in Formula 3.1 tells that every reaction is a process that takes an initial set of metabolites $[a_1, \ldots, a_n]$ with certain amounts for each metabolite in the set $[s_1, \ldots, s_n]$ and transforms it into a final set of metabolites $[b_1, \ldots, b_n]$ with their amounts $[p_1, \ldots, p_m]$. The formula 3.2 is a particular instance where the reaction transforms the initial set of metabolites composed by 8 metabolites of Fe and a single metabolite of S_8 into a final set of metabolites composed by 8 metabolites of FeS.

Because in Cadmium the atomic models are declared as C++ classes, this approach is easily achieved by just creating a parametrized abstract atomic model class, then, a model generator uses this abstract class to generate model instances. We are able to use traditional OOP techniques to define the general atomic model classes that the model generator module uses to generate the final model (explained later in section 5.2). These general model classes are saved in C++ header files that will be included and compiled in the final model executable object.

The defined general model classes are parametrized with the particular values of each instance to obtain the final DEVS models to use. These parametrized values are individual properties that are not common to all models of that class, and they must be differently

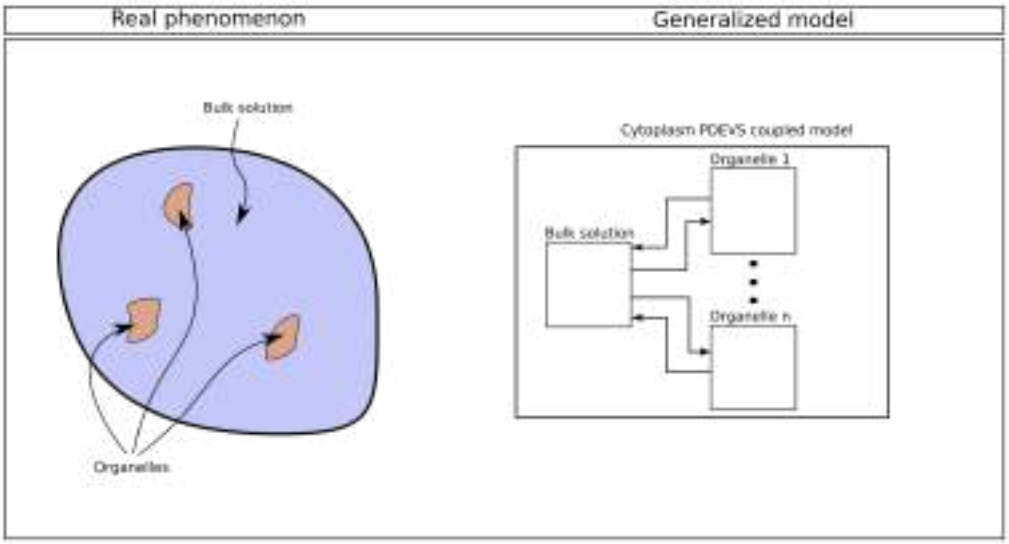


Fig. 3.1: Example of the model general structure for a very simple cytoplasm with organelles.

defined for each instance of the general model class to obtain a model of a particular component of a biological cell. Thus, to construct a final DEVS model of a particular biological cell component, we need first to collect the correct values from that component to parametrize the general model class and obtain the instance that models that biological cell component.

This idea also applies to the biological cell structures. Compartments and organelles remain the same until a cell division is reached. We can generalize the cell's common structures and parametrize the non-general aspects. An example of this is the cytoplasm: all cells have a cytoplasm, but each cell could have zero or more organelles with the same general structure, thus, the cytoplasm structure can be generalized as shown in Figure 3.1, where the number of organelle components is variable.

The model structures are modelled using DEVS coupled models (explained in 2.2.2). In Cadmium, coupled models are instances of the cadmium::coupled class, and the constructor takes as parameters of the DEVS coupled model: the list of sub-models, ports, EIC, EOC and IC links. The model generator defines the general structure model classes as methods (i.e. C++ functions) that consumes parameter values and produces lists of sub-models, ports, EIC, EOC and IC to construct the coupled model. In order to modify the general structure classes, we must modify those methods.

To obtain more general models, we separate the biological cell structure model from the model of the active components. Then, we propose a model of a general cell structure that parametrizes the values of different cells and where the biological cell active components can be modelled independently from the rest, allowing to easily modify or replace them. Then, the general biological cell model can be instantiated using information of a real cell.

Now that we have introduced the idea that a biological cell general structure can be abstracted, we will show the proposed model of the general structure and how does this structure map a real biological cell.

All biological cells have a periplasm membrane that wraps the cell. The periplasm interacts with the extracellular space exchanging metabolites. The periplasm is a complex membrane composed of two sub membranes, the periplasm outer membrane in charge of interacting with the extracellular space and the periplasm inner membrane in charge of interacting with the cytoplasm bulk solution. Also, the cytoplasm has a transmembrane where reactions that consume metabolites from the extracellular and cytoplasm at the same time occurs. Even though the periplasm is a membrane, reactions can occur within it. The periplasm wraps the cell cytoplasm bulk solution, which is a jelly-like fluid that contains metabolites, enzymes and organelles. Each organelle has an outer membrane that separates the organelle from the rest of the cell and exchanges metabolites with the cytoplasm. Also, organelles have an inner space with metabolites and enzymes. In terms of modelling, a cell can be considered as a hierarchical structure with components and sub-components where the periplasm is at the highest level and the metabolites at the lower level.

Each of these biological organelles can be modeled using DEVS; we can build a DEVS hierarchical structure using almost the same structure observed in the biological cell and the mapping between the biological cell organization and the structure of the DEVS model remains simple and clear.

We designed the model hierarchy similar to the explained real cell structure that has a hierarchical nature. But, there is a significant difference between the model structure and the real cell structure, instead of defining a single high level model to represent the extracellular space with only one sub-model representing the periplasm and so on, we decided to model the extracellular, periplasm, cytoplasm and organelles as sub-models that are all located at the highest level of the model hierarchy. This design decision allows a better integration of new sub-models that could interact at the same time with components that would be at different levels, making the model more flexible to changes. Then, the general model structure is the coupled model shown in Figure 3.2 where the biological cell hierarchy is defined by the communication links between the sub-models that represent the extracellular, periplasm, cytoplasm and organelles. As we can see in Figure 3.2, the extracellular space sub-model only communicates with the periplasm sub-model, and the periplasm sub-model communicates with both the extracellular and the cytoplasm sub-models. Finally, the cytoplasm communicates with the periplasm and with the organelles sub-models, this communication line defines the same hierarchy present in a real biological cell.

In the proposed structure, we model each metabolite as an individual an independent entity obtaining a micro view model that represent the biological cells at the genome scale. The implemented model structure represents the physical separation of the cell compartments where reactions and other processes occur. These compartments not only store subcomponents, but they handle the interaction with other compartments and the metabolites movement through spaces and bulk solutions. On the other hand, the metabo-

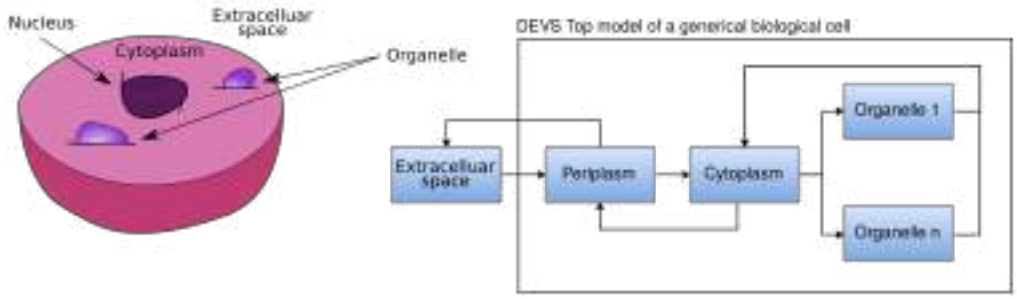


Fig. 3.2: Mapping between a general biological cell and the general model structure. The DEVS models are simplified and a better diagram is presented in figure 4.1.

lites are not static entities that are always placed in the same compartment interacting with the same elements. Metabolites are consumed and created from reactions, and they can travel from one compartment to another. On the other hand, metabolites do not introduce behaviour by themselves; instead, they are used by the reactions as the substrate and product. Because of this reason, we have defined the metabolites as weak and dynamic entities that only exist in the atomic model states and within the messages. Both messages and component states are not static and, therefore, metabolites can be constructed and destroyed as needed. Consequently, we are able to model the dynamics of a biological cell in a micro-view.

Also, as mentioned in [1] and in chapter 1, whenever a metabolic network is modelled we need to deal with the exponential explosion of possible combination reactions within the network. We cannot consider every possible pathway; instead, we need to calculate those pathways that will effectively occur. For this purpose, we have modelled the spaces as stochastic processes that determine when collisions between proteins and metabolites happen and send the information to the component in charge of modelling the reaction. The metabolic network is then not explicitly modelled. For this purpose, we defined atomic models (called space models) to model the dynamics of how the metabolites and enzymes collide within a compartment. Space atomic models store in their states the number of metabolites and enzymes existent in the compartment and use stochastic processes to determine when metabolites and enzymes collide. When collisions occur, the space model sends the information to the corresponding component in charge of handling the reaction, and once the reaction is over, the reaction product is sent back to the corresponding space atomic models to update their number of metabolites and enzymes.

Using a hierarchical and modular structure that represents the physical compartments of a biological cell allows us to easily improve or even change different components of the model without modifying the rest. This is important because even if we propose a specific model for the metabolic network that model spaces as atomic models, other modellers could change these components to use a different approach that better fits their needs. For example, they could model the space using Cell-DEVS, which can model spatial aspects. Having a hierarchical and modular structure also allows us to add new models and to connect them to the existing components in order to consider new elements of the biological cell that were not yet considered, eventually leading to a whole-cell complex model.

A major problem of working with a micro-view model as the one we propose in this work is that biological cells have thousands of hundreds of enzymes that are modelled as separated atomic models. These components are located in the most-bottom level of the model hierarchy called enzyme sets. The periplasm has four enzyme sets (the outer, inner and trans membranes and the inner space). The cytoplasm has one enzyme set (the inner space) and each organelle has two enzyme sets (a membrane and an inner space). At the same time, each enzyme set has up to thousands of different kinds of enzymes and there are up to hundreds enzymes of each kind, leading to an enormous number of enzyme atomic models. This could result in high overhead in the simulation processing system due to the number of atomic models and links that requires to be correctly coordinated by the simulator. The simulator coordinator system is in charge of transitioning its sub-models and sending messages from one model to another through the links when the virtual time advances. Thus, the simulation time could be very high.

Likewise, the model generation process must generate all the enzyme model instances. If there are multiple equal enzymes, the generator must generate multiple exact instances of the same model, and this consumes processing time and memory. On the other hand, the resulting model is a C++ program that must be compiled, and as we will explain in section 6, compiling large models also has a considerable overhead.

3.1 Multi-state models

To deal with the problem of having a large number of atomic models, we introduce the concept of multi-state models to combine all the atomic models representing the same enzymes in a single atomic model. The state of the new atomic model is the combination of the state of each enzyme in the set. Also, to improve the simulation process memory, when we combine all the enzyme states, we can merge the states of those enzymes that have the exact same value until a transition is executed. Thus, instead of having several identical states we have a single one. By doing so, we keep the number of components low enough to run a simulation while still modelling at the genome-scale.

A model state is defined as a set of variables where each variable is a simple or complex object; a transition is a modification of one or several fields of the state. The state of a multi-state model is a set of pairs (sub-states, ta) that represent the state of all the models that share the same behaviour associated with their time advance. When the time advance is calculated, we use the minimum time advance of all sub-states. When the internal transition function is triggered, only the states with the minimum time advance are processed, and the rest is updated using the elapsed time. Because the grouped models are equals, each sub-state in the array has the same fields and the transition rules are the same for all of them. Thus, we can define a multi-state model for any atomic model as follows:

If we have a DEV atomic model $M =< X_A, Y_A, S_A, ta_A, \delta_{ext_A}, \delta_{int_A}, \delta_{conf_A}, \lambda_A >$. Then, we define its multi-state model as $M_{mult} =< X, Y, S, ta, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda >$ where:

- $X = X_A$.
- $Y = Y_A$.
- $S = \{\{(s,t)|s \in S_A, t \in R\}\}$.
- $ta(s) : min\{s'.t|s' \in s\}$.
  The time advance is the minimum time advance between all the state sub-states.
- $\delta_{ext}(te, x^b, s) : \{(\delta_{ext_A}(te, x^b, s'), ta_A(\delta_{ext_A}(te, x^b, s')))|s' \in s\}$.
  All the model sub-states will be forced to an external transition by the original DEV formalism definition. Thus, the time advance must be calculated for all the new sub-states using the composition of the time advance of the current sub-state external function.
- $\delta_{int}(s) : \{(\delta_{int_A}(s'), ta_A(\delta_{int_A}(s')))|s' \in s \; if \; ta_A(s') = ta(s)\} \cup \{(s'.s, s'.t - ta(s))|s' \in s \quad if \quad ta_A(s') > ta(s)\}$.
  All the model sub-states with the minimum time advance are the sub-state that must transition in the triggered internal time, for these sub-states, the new time advance must be calculated using the composition of the time advance and the current sub-state internal function. For the rest of the sub-states, their time advance must be updated according to the elapsed time.
- $\delta_{conf} : \{\delta_{conf_A}(x^b, s')|s' \in s \; if \; ta_A(s') = ta(s)\} \cup \{\delta_{ext_A}(ta(s), x^b, s')|s' \in s \; if \; ta_A(s') > ta(s)\}$.
  Following the definition of DEV, the confluence transition must be triggered for all the sub-states that have an internal transition scheduled at the same virtual time at which the external events occur. For the remaining sub-states, the external transition must be triggered with the current global state time advance as the elapsed time (i.e., the time advance of those sub-states that are handled by the confluence function).
- $\lambda : \{\lambda_A(s')|s' \in s \quad if \quad ta_A(s') = ta(s)\}$.

Similarly, we can allow the original transition functions of the sub-states model to create multiple states appending those new states to the sub-states array. This can be interpreted as a non-deterministic DEV model in which states can transition to a state set instead of transitioning to a single state. If the set is empty, the sub-state is eliminated. This mechanism allows to create and destroy atomic models dynamically.

Because now multiple atomic models are combined into a single multi-state model, the newly single model will receive all the messages directed to the original atomic models. Therefore, each time a message arrives at the multi-state model, it is necessary to determine to which original model the incoming message goes, in order to trigger the external transition only for the sub-state corresponding to that original model. For example, in Figure 3.3.a (normal model), we have that the model A sends messages to the model C1, and the model B sends messages to the model Cn. Then, when we combine models C1, ..., Cn into a single multi-state model (as defined earlier) we need to link the model A and B to the new multi-state model. In this case, when the model A sends a message to (C1, ..., Cn), the external transition function should be triggered only for the sub-state

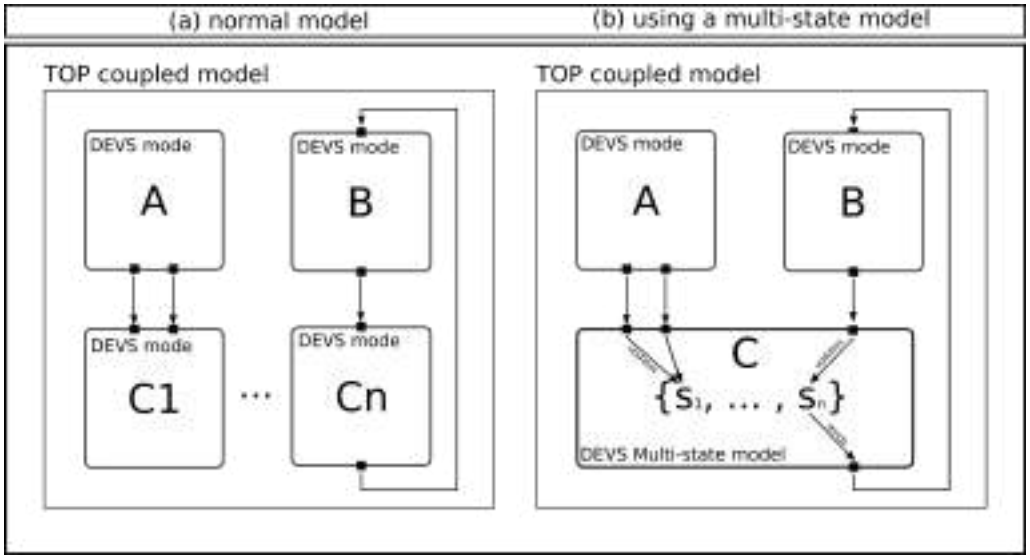


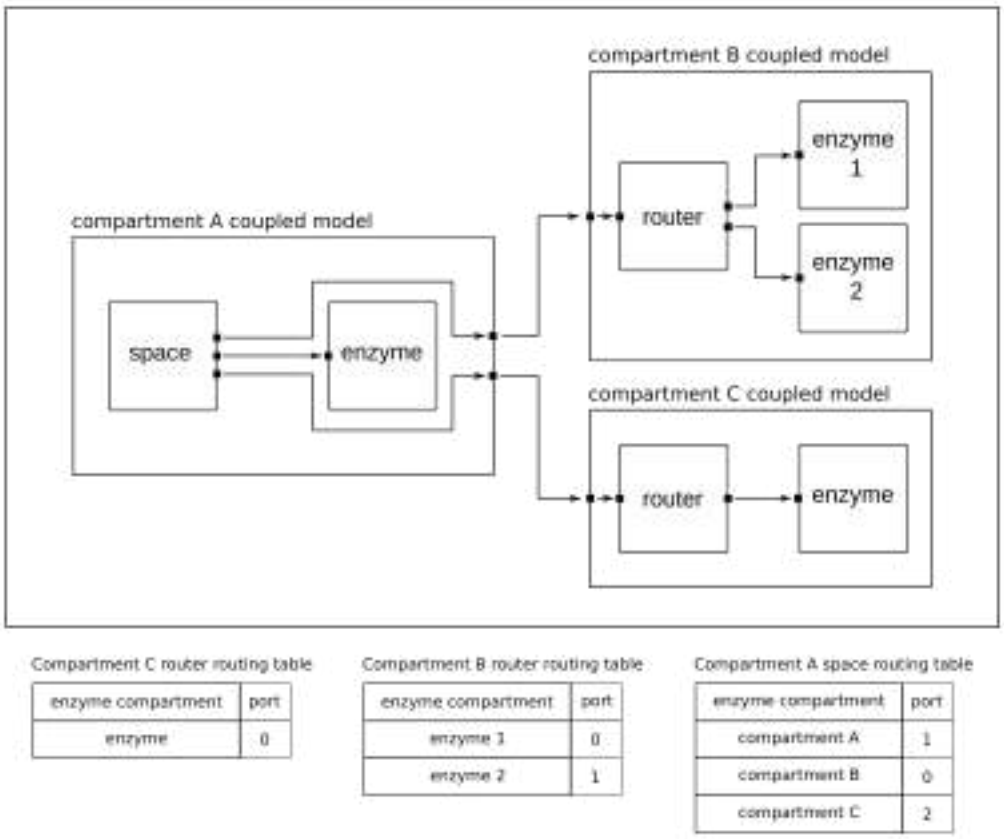
Fig. 3.3: Example of port mapping in a multi-state atomic model.

corresponding to the model C1 (sub-state S1), because the model A is connected to the model C1 but not to the remaining combined models (C2, ..., Cn).

A simple solution to this problem is to add all the existing ports in the original models to the multi-state model. Each port will then be mapped to the sub-state corresponding to the original model of that port. In this way, all the messages arriving at a particular port will trigger an external transition in the sub-state associated with that port, without affecting the other sub-states. For example, in Figure 3.3.b (using a multi-state model), the model A is connected to the multi-state model (C1, ..., Cn) using the same links that linked the model A with the model C1 in the original model (Figure 3.3.a). Those ports are also mapped to the sub-state S1 (the sub-state of the original model C1). Then, each time the model A sends a message to the multi-state model, those messages will trigger a transition of the sub-state S1 without modifying the rest of the sub-state. This is the correct behavior since in the original model in Figure 3.a, model A only sends messages to the model C1. Thus, only the sub-state of C1 should be modified when A sends a message to the multi-state model (C1, ..., Cn).

Similarly, we can map the output messages generated by the model sub-states to different output ports. In that way, output messages of a sub-state are sent to the right destination. Figure 3.3 illustrates the mapping between the multi-state model input and output ports and the sub-states.

By using multi-states models we significantly decrease the atomic model's amount by grouping equal enzymes in a single model; nevertheless, as we already mentioned, there are thousands of hundreds of enzymes. Consequently, we have a considerable number of links to connect each enzyme to other components. Because enzymes are located in enzyme sets within compartments, we are able to reduce the number of existent links by unifying all the links that communicate with the same components, and then we redirect the messages



Compartment C router routing table

enzyme compartment	port
enzyme	0

Compartment B router routing table

enzyme compartment	port
enzyme 1	0
enzyme 2	1

Compartment A space routing table

enzyme compartment	port
compartment A	1
compartment B	0
compartment C	2

Fig. 3.4: A model routing system where a compartment space sends metabolites to enzymes from different compartments.

within each component.

3.2 Events routing system

We have used a routing system with similar concepts to network protocols, where routing tables are used to determine the next component where to send messages. Each space model needs to send the metabolites that are ready to react to the enzyme atomic model in charge of handling that reaction. For this purpose, it uses a routing table that indicates which port is connected to each enzyme set and the metabolites are sent to the enzyme set where the enzyme belongs. Once the messages are received by the enzyme set, a router is used with a new routing table to redirect the metabolites to the corresponding enzyme multi-state model. Figure 3.4 shows an example of a simple model architecture for a space that sends metabolites to multiple enzyme sets in different compartments and the corresponding routing tables associated with the model.

In Figure 3.4, space only knows to which port it must send the metabolites depending on the enzyme compartment. Once the enzyme reaches the compartment, a router redirects the metabolites to the corresponding enzyme within the compartment.

By using this routing mechanism, we keep the model top-levels links simple and only

the lowest levels include several links from the routers to the enzyme models. This improves the simulation performance because only the bottom level coordinators deal with most of the links.

This routing system is a modelling mechanism not related to the modelled phenomenon, and it should be transparent to the simulation results without adding any additional delay in the simulating virtual time. Router models wait for incoming messages, and when that happens, an instantaneous internal transition is scheduled to output the routed messages using zero time units.

4. THE MODEL ARCHITECTURE

Now that we have introduced the theoretical concepts of the model's architecture we will explain the implementation of the architecture in a top-down fashion. First, we will explain the coupled models' structure, and then we will explain the atomic models' behaviour.

4.1 Coupled models

The architecture of the top model represents an abstraction of a real biological cell interacting with the extracellular space. It is therefore composed as shown in figure 3.2: an extra cellular space model, a periplasm model, which includes three sub-models representing the outer, inner and trans membranes of the periplasm, a cytoplasm model and multiple organelle models.

Because in biological cells metabolites can travel from the extracellular space to the cytoplasm (passing through a trans-membrane reaction, or moving to the periplasm bulk and then to the cytoplasm through the periplasm outer and inner membranes), we represented the extracellular space as a compartment linked with the periplasm outer and trans-membranes to send and receive metabolites from it. Likewise, metabolites can travel from the cytoplasm bulk to any organelle bulk by first passing through the organelle's membrane. Consequently, we have a link between the cytoplasm and each organelle.

Commonly, organelles have only one membrane, but this could not be the case and we consider a variable number of membranes. Each membrane can communicate with the organelle's bulk solution and with other models outside the organelle. By generalizing the organelle membranes, we can consider the periplasm just as any other organelle with three membranes (inner, outer and trans). Then, the cytoplasm and extracellular space models have links to the corresponding organelle membranes to which it can interact.

In biological cells, metabolites travel through the compartments in transport enzymes, but they are also catalyzed by non-transport enzymes, allowing the cell to produce energy and to transform metabolites. For this reason, each compartment model has an inner enzyme set where those reactions occur.

Figure 4.1 shows the coupled TOP model structure of the cell. In this structure, each sub-model represents a physically separated compartment, and the links between two sub-models represent that those compartments exchange metabolites.

On one hand, as a compartment has one or multiple membranes, each sub-model has one or multiple input ports to receive metabolites through its different membranes. For example, the Periplasm model has three membranes and therefore it has three input ports (one for each membrane). Then, other models are able to send metabolites to the different Periplasm model membranes by sending them to the correct port. The Periplasm outer membrane is connected to the port named "O", the inner membrane is connected to the port named "I" and the trans membrane is connected to the port named "T". Then,

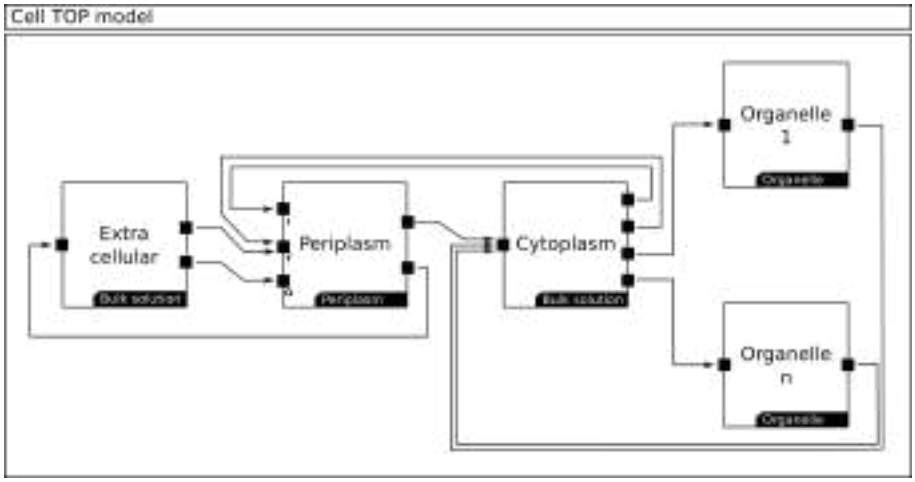


Fig. 4.1: The DEVS coupled structure: The periplasm ports communicates with its outer (O), inner (I) and trans (T) membranes. The black labels in the sub-models indicate their coupled model general class.

as in real biological cells, the extracellular space only interacts with the Periplasm outer and trans membranes, the extracellular space model is only connected with the periplasm model input ports "O" (outer membrane) and "T" (trans membrane). Likewise, in real biological cells, the Cytoplasm communicates with the periplasm through the inner and trans membranes and then, the Cytoplasm model has links connected to the Periplasm model input ports "I" (inner membrane) and "T" (trans membrane).

On the other hand, each model has one output port for each external membrane (membranes of other models) to which it sends metabolites. For example, the Cytoplasm model sends metabolites to two different Periplasm model membranes and to all the organelle membranes. Therefore, it uses two different output ports to connect to the Periplasm input ports (one for each membrane) and one different output port for each organelle. In this way, the Cytoplasm can send metabolites to one of the Periplasm model membranes by sending the metabolite through the output port connected to the correct Periplasm model input port connected to that membrane.

Finally, all the coupled top model sub-models are coupled models that will be explained later on in this chapter.

The Nucleus is an important sub-system of biological cells, but not all cells have one, and by modelling the Nucleus we would restrict which cells can fit in the proposed structure. Furthermore, the Nucleus could be modelled by either adding a new component as mentioned before, or by considering the Nucleus as a new organelle (as we did with the periplasm).

4.1.1 The periplasm coupled model

In the biological cells, the Periplasm transports metabolites from the extracellular space to the cytoplasm through different membranes. It can also catalyze reactions in its inner

space (a bulk solution) to transform metabolites as part of the cell metabolism. Because of this, we represented the periplasm as a coupled model composed of three membrane enzyme sets models (representing the outer, inner and trans membranes), and a compartment with a bulk solution model where we handle the reactions occurring within the Periplasm.

The Periplasm membrane enzyme sets are models that connect the different compartment to exchange metabolites through transport reactions. Transport reactions are reactions whose main purpose is to transport metabolites between compartments. For this purpose, transport reactions consume and produce metabolites from different compartments; a metabolite in a compartment can be consumed by a transport enzyme that will produce the same metabolite in a different compartment. Likewise, transport enzymes can modify metabolites in the transportation process. Consequently, the metabolites that are consumed are not always the same as the metabolites produced.

The Periplasm membranes are the inner, outer and transmembrane models. The inner membrane model connects the Periplasm model with the Cytoplasm model and the reactions in the inner membrane consume and produce metabolites from the Cytoplasm and the Periplasm models. The outer membrane model connects the Periplasm model with the Extracellular space model and therefore, reactions in the outer membrane model consume and produce metabolites from the Extracellular and Periplasm models. Finally, the transmembrane model connects all three models at the same time (Periplasm, Extracellular and Cytoplasm) and therefore it consumes and produce metabolites from the

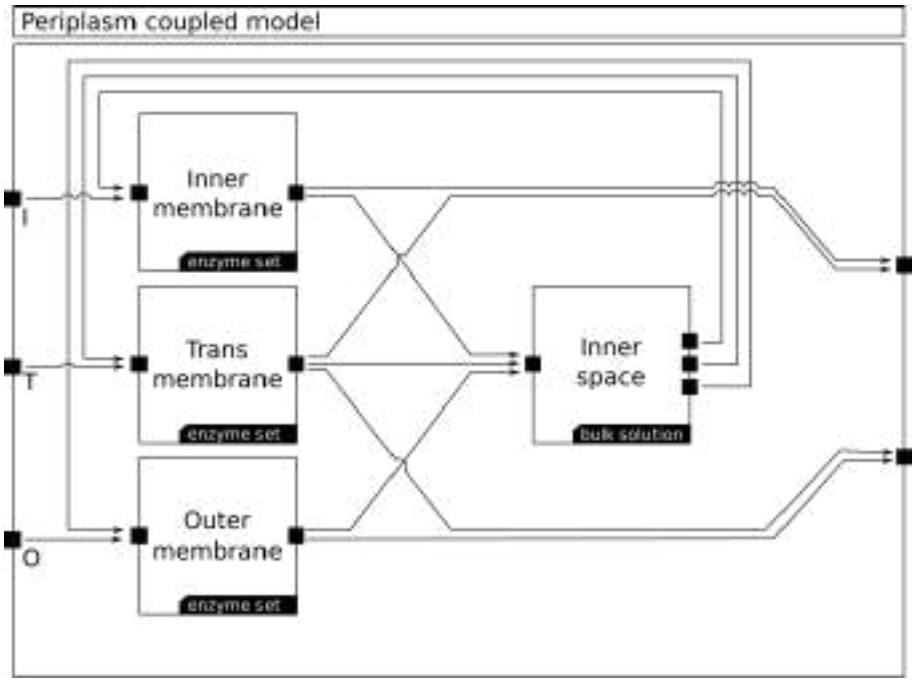


Fig. 4.2: The Periplasm coupled; Black labels in the sub-models indicate they are instances of those coupled model classes.

three models. Figure 4.2 shows the Periplasm coupled model structure where the links represent the connections we have just explained.

The metabolite flow within the Periplasm model is as follows:

On one hand, reactions occurring within the periplasm that are not transport reactions are modelled in the inner space coupled model. This model is an instance of the general bulk solution coupled model explained in section 4.1.2.

On the other hand, when the inner space coupled model determines that metabolites have collided with enzymes located in the Periplasm membranes, the inner space sends the metabolites to the membranes. The membranes also receive metabolites from other compartments (by the Periplasm input ports) and generate the metabolite exchange process between other compartments and the periplasm model.

4.1.2 The general bulk solution coupled model

The Extracellular, Cytoplasm, Periplasm and organelles inner space are different instances of the general bulk solution coupled model class. A bulk solution represents a space where reactions are catalyzed and where metabolites are exchanged with the compartment membranes. On one hand, the Extracellular and Cytoplasm bulk solution interacts with the cell membrane, which is the Periplasm membrane model. On the other hand, the Periplasm and organelles inner spaces interact with their own compartment membranes.

The general bulk solution model represents either a compartment without membranes (as it is the case of the Extracellular space and the Cytoplasm) or a sub-model of a compartment with membranes that has its own inner space (as it is the case of the Periplasm and Organelles). In both cases, the bulk solution model only exchanges metabolites with membrane models, and for each related membrane, it uses a different port. Figure 4.3 shows the general bulk solution coupled model.

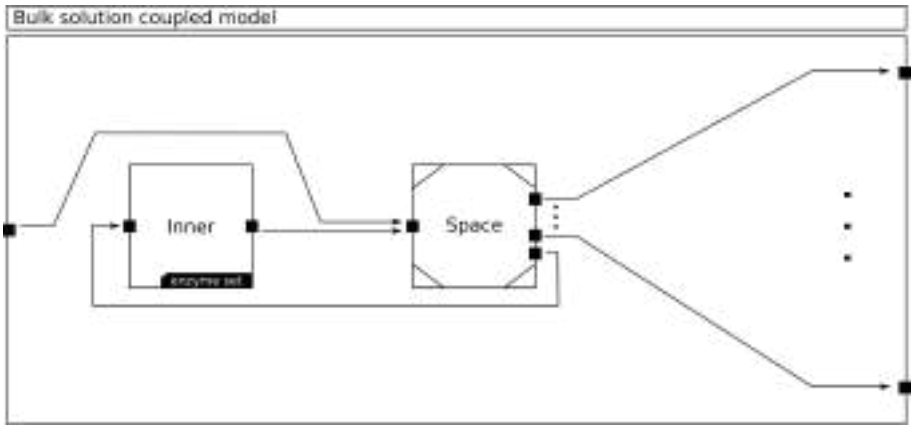


Fig. 4.3: The general bulk solution coupled model. Black labels in the sub-model indicate the coupled model class, the white triangle in the sub-model corner indicates that it is an atomic model.

Because bulk solutions directly communicate with external membranes and can be part of compartments models, they do not have their own membranes. Instead, bulk solution coupled models have one input port that directs all the incoming metabolites to the bulk solution space model and multiple output ports (each one connected with a different membrane).

The metabolite flow within a bulk solution model is as follows: the space atomic model represents the spatial distribution and movement of metabolites and enzymes inside a compartment. The space also models the collisions between metabolites and enzymes using a stochastic process explained in section 4.2.2. Each time the space model determines that metabolites have collided with enzymes, the space model sends the metabolites to the corresponding enzyme model that represents the process of metabolite binding and reacting, which is explained later in section 4.2.1.

The enzyme atomic models are located in enzyme set coupled models that can either represent a compartment membrane or a compartment internal metabolism (i.e. the enzymes catalyzing non-transport reactions within a compartment). The compartment's internal metabolisms are represented by the inner coupled model of the bulk solution model. Thus, every time the space model determines that a metabolite has collided with an enzyme of the compartment metabolism, the space sends the metabolite to the inner coupled model. The space not only has information about the enzymes in the inner model, but it also has information of the enzymes of related compartments (i.e. enzymes that are connected to the bulk solution through the output ports). The space calculates collisions for all the enzymes and if a metabolite has collided with an enzyme outside the bulk solution, the space sends the metabolite through the corresponding output port.

4.1.3 The organelle coupled model

A biological organelle is a cell element specialized in handling one or multiple functions of the cell. In this work, we have modelled a general organelle as a coupled model with multiple membranes that communicates with the organelle inner space (a bulk solution model) and with other compartments. The organelle structure can be seen as a bulk solution wrapped by membranes that communicate the bulk solution with other compartments. Figure 4.4 shows the coupled model of a general organelle.

It is worth to recall that in the structural framework proposed in this work, every component can be modified, replaced, added or even removed. If a cell has an organelle that does not fit the proposed structure here, or if the modeller wants to use a macro-view model for one or more organelles, they are able to add their model to the structure and to communicate that model as a new organelle model.

As shown in Figure 4.4, the general organelle model has an arbitrary number of membranes "N" and an arbitrary number of output ports "M". Each membrane is only connected with those output ports that communicate the organelle with compartments that interact with that specific membrane. For example, in the case of the Periplasm (that is a particular case of an Organelle), The outer and inner membranes are connected with only two output ports while the transmembrane is connected with all the three output ports.

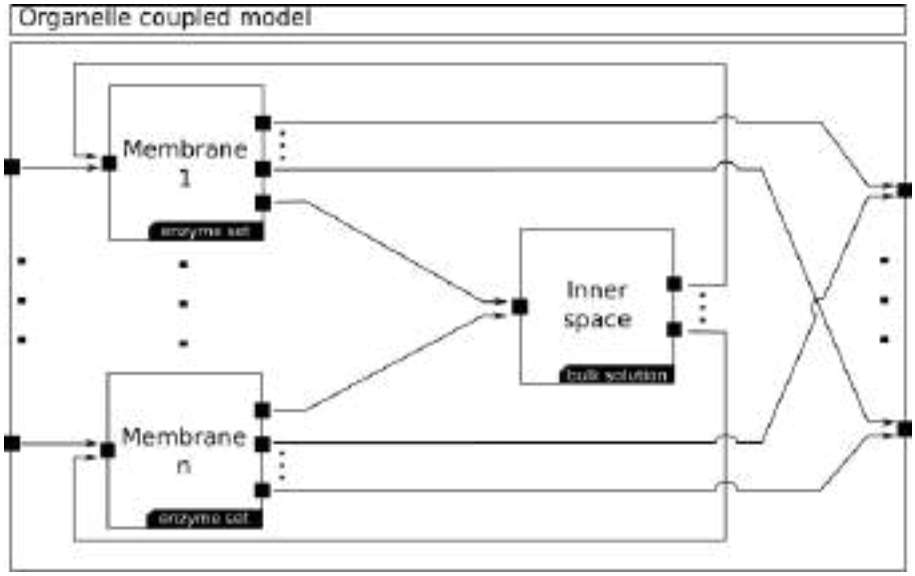


Fig. 4.4: The general organelle coupled model with multiple membranes, the periplasm model is a particular instance of this structure.

Likewise, an organelle has one input port for each membrane.

In a biological cell, organelles are placed in the Cytoplasm; consequently, Organelle models send and receive metabolites from the Cytoplasm model. In the proposed structure, an organelle can be considered as a very simple cell model with its own metabolic network. Then, the metabolites flow is encapsulated and neither the Cytoplasm knows what happens with them inside the organelles nor the organelles know what happens with metabolites in the Cytoplasm.

4.1.4 The enzyme set coupled model

The final coupled structure is the enzyme set, a simple model where a set of enzyme atomic models are grouped under the same compartment. The grouping of the enzymes allows the space models to direct the metabolite messages using the routing system explained in section 3.2 by sending the messages to the corresponding enzyme set, instead of sending the message to each enzyme. This reduces the space models outgoing links. Figure 4.5 shows the coupled model structure of the reaction set.

Because all the atomic enzyme models are linked to the same output ports, they use the same routing table to route their messages to the space models outside the enzyme set.

4.2 Atomic models

Now that we have already explained the model structure (coupled models), we will explain the atomic models.

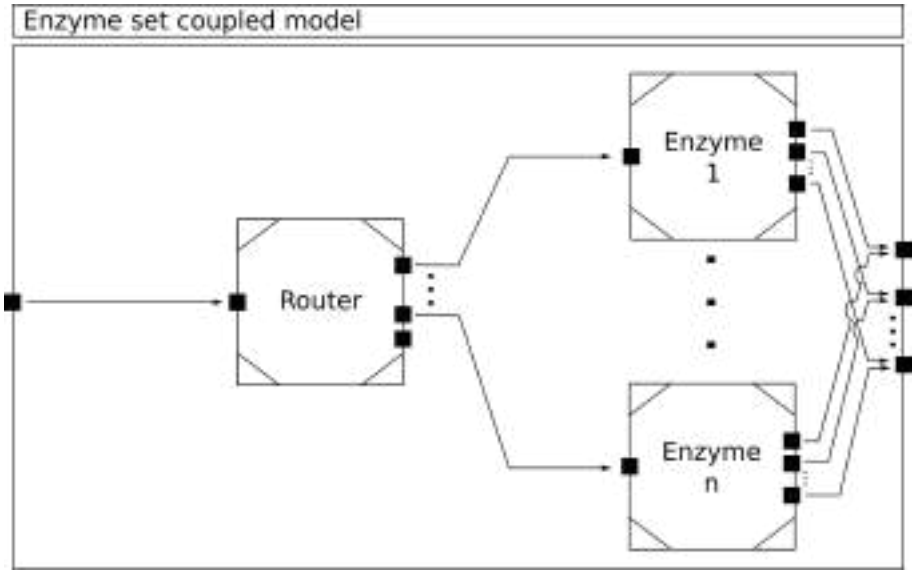


Fig. 4.5: An enzyme set with the router and the enzyme atomic models.

4.2.1 The general enzyme model

As we have already mentioned, an enzyme can handle multiple reactions. Then, considering that:

1. Each reaction consumes and produces different metabolites and have different rates and bounding constants.

2. Reactions are not physical elements of a biological cell, but they are the final behavior of an enzyme when it is catalyzing a reaction.

3. We used model components to only represent physical elements of a biological cell.

We decided not to model each reaction as a different atomic model (and the enzyme as a coupled model with reactions inside). Instead, enzymes are defined as atomic models that, within their internal logic, they handle the multiple reactions they can catalyze. Each time the enzyme atomic model receives metabolites messages, it will determine which reaction logic should trigger for those metabolites. On the other hand, two reactions cannot occur at the same time in a single enzyme, therefore, the enzyme is a model that can behave like one and only one of its reactions at a time.

Also, as we have already mentioned, there are several identical enzymes within a compartment, because of this, in order to avoid unnecessary simulation overhead, we have implemented the enzyme atomic model as a multi-state model.

To make it easier to understand the enzyme atomic model, we will first explain the model as a single DEVS atomic model and we will then expand it to a multi-state model. Also, To better understand the enzyme atomic model, we will first separate the general attributes of an enzyme model (those that must be instantiated when generating a final

model and remain constant during the entire simulation) from the enzyme model dynamic state (those that change at runtime).

The enzyme attributes and states are as follows:

- **Attributes:**
  - **Enzyme reaction:** (defined for each handled reaction)
    - Rate: A constant that indicates how long the reaction takes.
    - Substrate stoichiometry: A set of metabolites associated with amounts that indicate the left side of the reaction stoichiometry formula.
    - Product stoichiometry: A set of metabolites associated with the right side of the reaction stoichiometry formula.
    - $K_{off_{STP}}$: The K_{off} constant for rejecting Substrate To Product (STP) metabolites.
    - $K_{off_{PTS}}$: The K_{off} constant for rejecting Product To Substrate (PTS) metabolites.
    - Reject rate: A constant that indicates the rate at which metabolites are rejected for that reaction.
  - **Enzyme:** (not related with the enzyme reactions above)
    - Routing table: a table that indicates where we send must each rejected or produced metabolite.
- **Enzyme reaction state**: (defined for each reaction)
  - Bound substrate: A set of pairs (compartment, Boolean) indicating whether the substrate coming from a compartment is bound or not.
  - Bound product: A set of pairs (compartment, Boolean) indicating whether the product coming from a compartment is bound or not.
  - Status: one of Binding, Rejecting, Reacting

The product and substrate stoichiometry attributes are instances of the Formula 3.1 that indicates the equation of the reactions substrate and product. The enzyme model uses the reactions stoichiometries to determine which reaction must be triggered based on which stoichiometry consumes the received metabolites. Once the enzyme determines the reaction that must be handled, the enzyme will use the attribute and state of that reaction until the reaction finishes.

We use a simple binding process in which we assume that once the first metabolite is bound, the rate at which the remaining metabolites bind increases more than the time step explained in section 4.2.2. Thus, we can consider that all metabolites from the same compartment arrive at the same time, reaching a binary state of all bound or none bound. Consequently, bound substrate and product are Booleans.

The general enzyme model state flow is as next: The enzyme model starts in the status "Binding" with the bound substrate and product of all the reactions set as false for all the related compartments (compartments that send metabolites to the enzyme). Once a

metabolite arrives, the status will be updated to determine one of the next values: "Rejecting", if incoming metabolites are rejected, "Reacting" if all the related compartments already sent their metabolites and incoming metabolites are not rejected, or "Binding" if not all the related compartments have already sent their metabolites. The enzyme model can only accept incoming metabolites in the "binding" status and if its status is "reacting" or "rejecting" then, it will remain in that state for the time specified by the reacting or rejecting rate attribute, once the rate time is over, the reaction sends the product or rejected metabolites back to the corresponding compartments and comes back to the "Binding" status.

The enzyme rejects the incoming metabolites with a probability based on the K_{off} disassociation constant following the rule of Formula 4.1. Once the reaction starts, the reaction rate constant indicates how long the reaction will take until the produced metabolites are ready.

$$\boxed{P(X > K_{off}) \quad where \quad X \sim U(0,1)} \tag{4.1}$$

Also, the enzyme rejects the incoming metabolites if there is a reactions conflict. This occurs when the enzyme is in "binding" status, but some compartments already sent their metabolites. Then, the incoming metabolites must belong to the same reaction of the already bound metabolites. If the incoming metabolites and the already bound metabolites are from different reaction stoichiometries, then the metabolites are rejected.

An enzyme can in some cases handle reactions in both directions (substrate to product and product to substrate). When this is the case, each direction is considered a different reaction and therefore they cannot be handled at the same time. Thus, once metabolites from a compartment are bound as substrate, no product can be accepted until the reaction is over and vice versa.

Figure 4.6 Show the DEVSGraph diagram of the enzyme state transitions. The circle in the continuous lines (external transitions) shows the probability that the model will use that path if all conditions are satisfied. Note: The loop transition from "Binding" to "Binding" and the transition from "Binding" to "Reacting" are disjoin conditions, i.e., they cannot be satisfied at the same time.

Once we define each enzyme, we compose multiple equal enzyme models in a multi-state model. First, we will introduce some important and simple definitions:

- Enzyme equality: Two enzymes are equal if the values of their attributes are equal.
- State equality: Two enzymes have equal states if they are equal enzymes and their states are equals.

The multi-state enzyme is the application of the definitions explained in 3.1 to the single enzyme model we have just discussed, but we also merged the equal states because we do not care to which enzyme the metabolites were sent to. Instead, we only need to know how many enzymes are in each state at any given virtual time (Binding product,

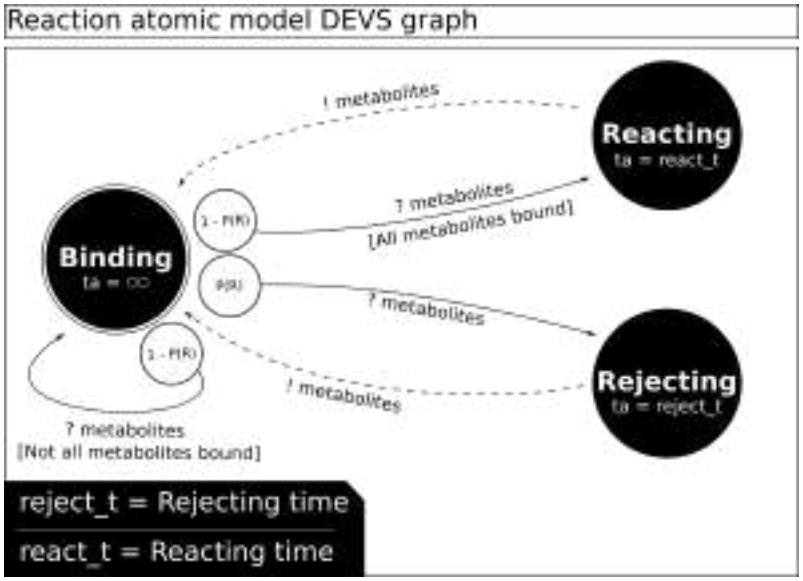


Fig. 4.6: The reaction atomic model DEVSGraph.

Binding substrate, Reacting and Rejecting).

Taking under consideration that we do not need to identify each enzyme, we can define the sub-states array elements as pairs (State, Amount) where the Amount indicates the number of enzymes that are currently in the specified State. Using this system, whenever metabolites arrive from the spaces, the external transition calculates the metabolite amount that can be accepted depending on the available number of states that can accept those metabolites and for the acceptable metabolites, it calculates how much will be rejected by applying the rejecting Formula 4.1 once for every acceptable metabolite. This mechanism could be optimized to calculate the rejected amount in a single step for all the acceptable metabolites with a more general formula obtaining a multi-level view model, but for this work, we propose a micro-view model and we keep the simple Formula 4.1 to calculate whether metabolites are rejected or accepted.

Finally, the enzyme atomic model confluence function is very simple. We first call the internal function and then we call the external function with an elapsed time equal to zero. This is a modelling decision that has no other motivation than obtaining more probability to bind new metabolites by first calling the internal function where we free all the reacting and rejecting reactions.

Two equal enzymes within the same multi-state model will send and receive metabolites to and from the same compartments. Because of this, every time there are metabolites to send from the enzyme multi-state model to one or multiple compartments, instead of sending one message per metabolite we group metabolites by their destination compartments. In this way, all the metabolites directed to the same compartment are sent in a single message, reducing the simulation message routing overhead. Then, once all the metabolites are correctly grouped, a routing table is used to determine by which port

the metabolites must be sent to reach the corresponding compartment using the routing system we have explained.

4.2.2 The general Space model

The space model represents the collisions between metabolites and proteins that are floating in the compartment bulk-solution. This model uses the following state values:

- Floating: all the metabolites and enzymes are in the bulk solution.
- Accepting metabolites: there are new metabolites in the compartment, incrementing the total number of metabolites in the bulk solution.
- Colliding: represents the process used to calculate the collision that happened in an interval of time in the compartment.
- Sending Metabolites: represents the process of sending the metabolites that have collided with enzymes to the corresponding enzyme models.

The model starts in the "Floating" state. Using fixed-length intervals, it calculates which metabolites have collided. At the end of each interval, the model sends each colliding metabolite to the corresponding enzyme. For this purpose, at the end of each interval, the model triggers a sequence of two instantaneous internal transitions. In the first transition, the model goes to the "Colliding" state, where it calculates all the collisions that occurred since the last internal transition using the Algorithm 1. Then, in the second transition, it switches to "Sending Metabolites" and it sends the metabolites that have collided to the corresponding enzymes. Finally, the model returns to the "Floating" state where it will remain for the duration of the interval.

Each time an external event occurs, the model switches to "Accepting metabolites" and it increases the number of metabolites in the compartment.

Figure 4.7 Show the DEVSGraph diagram of the space atomic model. The time advance of the "Floating" state is only calculated when transitioning from the "Sending metabolites" state. For the remaining transitions to the "Floating" state, the time advance is the interval time minus the elapsed time from the last "Sending metabolites" state. This is because we want fixed time steps to calculate collisions.

To understand the logic of the space atomic model, we will first focus on the real biological binding mechanism that we have modelled. There are multiple binding mechanisms; one of the most common (which we used here) is as follows. Enzymes and metabolites float in the bulk solution, and when a metabolite is near an enzyme, they can collide. When a metabolite collides with an enzyme there is a chance that the metabolite will bind with the enzyme. The probability of this event is not independent of the current enzyme state. Instead, every time an enzyme binds to a new metabolite, the chances to bind to new ones increase exponentially.

Algorithm 1 Calculates the enzyme and metabolites collisions

procedure CALCULATE_COLLISIONS($state, K_{on_{PTS}}, K_{on_{STP}}$)
  $collisions \leftarrow \{\}$
  for $enzyme \in state.enzymes$ **do**
    $p_{kons} \leftarrow []$
    for $reaction \in enzyme.handled\_reactions$ **do**
      $P_{kons} \leftarrow P_{kons} \cdot [< reaction, STP, P_{partial\_binding}(reaction, K_{on_{STP}}) >]$
      if reaction is reversible **then**
        $P_{kons} \leftarrow P_{kons} \cdot [< reaction, PTS, P_{partial\_binding}(reaction, K_{on_{PTS}}) >]$

    if $\sum_{p \in p_{kons}} p > 1$ **then**
      $p_{kons} \leftarrow [\frac{p_{kon}}{\sum_{p \in p_{kons}} p} \mid p \in p_{kons}]$
    $X \leftarrow U(0, 1)$ ▷ Uniform random value
    for $i \in [0, \ldots, |P_{kons}|]$ **do**
      if $\sum_{j=0}^{j=i-1} P_{kons_j} < X < \sum_{j=0}^{j=i} P_{kons_j}$ **then**
        $collisions \leftarrow collisions \cup \{p_{kons_i}\}$
  return $collisions$

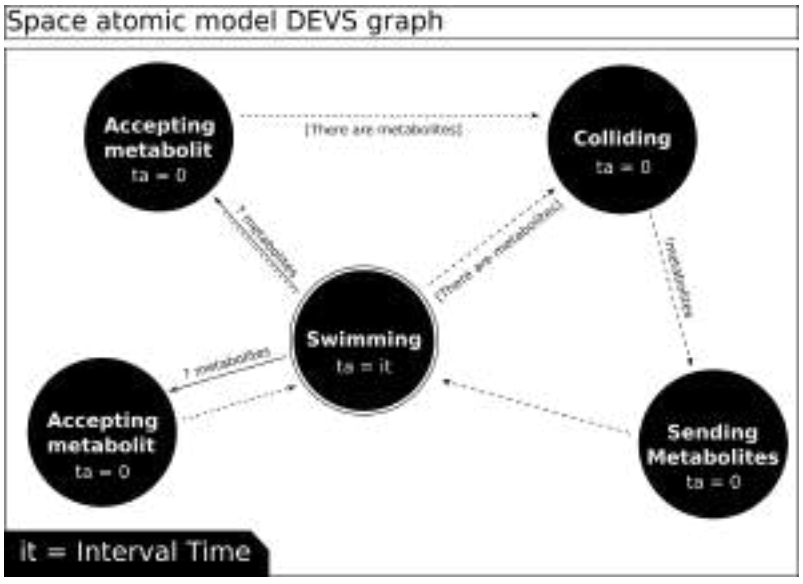


Fig. 4.7: The Space DEVSGraph. The double line (continuous and dotted) represents a confluence function.

To model the binding mechanism that we have just explained, we consider that once an enzyme binds to the first metabolite, the probability to bound the remaining needed metabolites is close to one (because it increases exponentially). Because of this, once an enzyme binds the first metabolite, we can assume that it will bound all the remaining metabolites it needs to react. Thus, we use a binary probability function where the enzyme binds all the metabolites it needs to react to, or it does not bind any metabolite at all.

Some enzymes consume metabolites from different compartments. In this case, the binding process is independent for all the compartments and each space model will calculate the binding process without knowing the state of other compartments.

To calculate the binding probability, we use a spatial stochastic formula that considers the space volume to calculate the metabolite concentrations as shown in formula 4.2, where L is the Avogadro constant used to deal with the fact that concentrations are calculated in moles unit and in our micro-view model we have the metabolites in metabolite amount units.

$$\boxed{[A_i] = \frac{|A_i|}{L * compartment\_volume}} \tag{4.2}$$

Multiple enzymes can bind metabolites of the same type. Then, if an enzyme binds a metabolite, the concentrations of that metabolite type decreases. To model this, we use a uniform random distribution to determine in which order the enzymes will bind the metabolites. Each time an enzyme binds some metabolites, the free metabolites amount is updated, and the concentrations decrease for the remaining free enzymes.

Likewise, an enzyme can handle multiple reactions depending on which metabolites the enzyme bounds. To model this, we use a two-step process where we first use the formula 4.3 to calculate the partial probability of the enzyme to bind the metabolites of each reaction that it handles. Then, we use the formula 4.4 to integrate all the partial probabilities and determine which metabolites will bind the enzyme. As formula 4.4 uses the interval [0, 1], we normalize the single probabilities to fit in the interval when the sum of all the single probabilities is greater than 1.

$$\boxed{\begin{array}{l} P_{partial\_binding}(reaction, k_{on}) = e^{\overline{\prod_{i=0}^{n} [A_i] * k_{on}}^{-1}} \\ A_i \in \text{reaction stoichiometry.} \\ n : \text{number of elements in the stoichimetry.} \\ K_{on} : \text{association constant.} \end{array}} \tag{4.3}$$

$$P_{bind}(r_i) = P(\sum_{j=0}^{j=i-1} P_{partial\_binding}(r_j, K_{on_j}) \leq X < \sum_{j=0}^{j=i} P_{partial\_binding}(r_j, k_{on_j}))$$
$$X \sim U[0,1] \tag{4.4}$$
$r_j : j \in [0,n]$ the j-th reaction handled by the enzyme in some arbitrary order.
$n :$ number of handled reactions.

The formula 4.4 splits the interval [0, 1] into $n+1$ disjoint subintervals. Each interval represents a different reaction handled by the enzyme. The duration of each interval is determined by Formula 4.3, according to the concentration in the compartment of the metabolites involved in the reaction. Once the interval [0, 1] is divided, we use a uniform random variable to choose one of the sub-intervals. Finally, the enzyme binds the metabolites of the reaction represented by the chosen interval. The last interval (the number $n+1$) represents a case where the enzyme does not bind any metabolite. The length of the last interval (the interval number $n+1$) is not obtained from Formula 4.3; instead, it is the remaining space once the first n intervals are calculated.

When external events occur, new metabolites arrive at the space model and the number of metabolites must be incremented. Then, the space state is updated to consider the incoming metabolites and they will be available in the next scheduled internal function when collisions are calculated. Each time that metabolite is sent to an enzyme, the space atomic model decrements the number of enzymes of that type until the enzyme returns the rejected or produced metabolites.

4.2.3 The router model

The router is a very simple model that redirects incoming messages through the corresponding ports depending on the message enzyme ID field. The model uses a routing table which is a key value map of enzyme IDs and port numbers.

This model is used for performance purposes (as explained in section 3.2) and it does not model any part of a biological cell. Thus, this model must not have any impact on the simulation results. Because of this, the incoming messages are routed in zero virtual time.

The model behaviour is as follows: the model remains passivated until one or more messages arrive. Then, the external transition saves the messages in the model state and schedules an internal transition in zero time. After the external transition ends, the output function sends the saved messages through the corresponding ports. Finally, the internal function clears the messages and comes back to the passive state. Figure 4.8 Show the DEVSGraph diagram of the router atomic model.

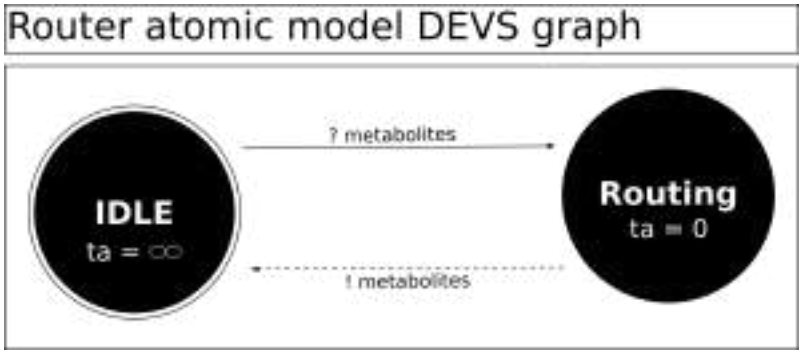


Fig. 4.8: The Router atomic model DEVSGraph.

5. METABOLIC PATHWAYS AUTOMATIC M&S AS A SERVICE

As we have already mentioned, the concrete goals of this thesis are to:

1. Propose a general structure that serves as a modelling framework for biological cell models.
2. Implement a web platform for automatic modelling of biological cells metabolic pathways that consumes SBML file to generate the model.

Until now, we have discussed the first goal: in chapter 3 we introduced the theoretical model and in chapter 4 we introduced the model architecture (coupled and atomic models). Here we will discuss the second goal by introducing a Web architecture and a user interface flow. Next, we will explain the automatic model instantiation process using SBML files of particular biological cells. Finally, we will introduce different modules that we have built for this project but are independent and generic to be reused in other projects as well.

5.1 Web platform architecture for automated M&S from SBML in DEVS

In this thesis, we defined a platform that allows a remote access to the automatic M&S application system. For this purpose, we used a classic client-server architecture where clients send commands to the server, the server then executes the commands and returns results back to the clients. The proposed platform was implemented in Django, a Python framework for back-end development [13] that handles all the automatable parts of a web platform. For example, Django handles HTTP requests from clients and redirects their messages to a given controller function; then, it sends the function results back to the clients. Django also handles all the interaction with database engines, automating SQL query generation processes. Django ALSO allows running multiple applications in a single server [38]. Django integrates the different applications using the Django URL routing system [39]. Using these features, we can build a complex platform based on multiple simpler applications running in the same server.

Django uses a design paradigm very similar to the well-known Model View Controller (MVC), which is a design concept mainly used in client-server web applications because it introduces a perfect decoupling between the server data state that must be persisted, the client user interface and the controllers that communicate the client interface with the data models. In an MVC design, modules are separated as follows:

- The **model (M)**: a logical representation of the data. The model is not the data itself, but an abstraction that allows to work and manage data without the need of handling an underlying database. Because of this, persisting and managing data becomes independent from the database engines being used.
- The **view (V)**: a user interface (UI) layer that defines how the client sees the server data. The View also allows the client to send commands and new data to the server in order to modify the current server data state.

- The **controller (C)**: handles the information exchange between the model and the view. The controller manages all the logic that modifies the current server data state using the model layer and sends the correct interface to the client using the view layer. Likewise, the controller manages the commands and data sent by the client.

Developers must specify each one of these items in a Django application and plug the applications in the Django server.

In this thesis we developed a set of Django applications called MPath-MS (Metabolic Pathways Modelling & Simulation), which provide a web platform for automatic modelling and simulation of biological cells metabolic pathways. The applications developed are:

- PMGMP (Parsing and Model Generation of Metabolic Pathways): it manages the SBML files and the DEVS model generation process from the SBML files.
- DEVSDiagrammer: used to visualize the generated DEVS model structure that allows exploring the different model levels.
- Sim-RT-Runner: it provides the client with a way to run simulations of the generated DEVS models and to visualize the results in real time.

All these applications are plugged into a Django server that redirects the incoming HTTP requests to the corresponding application to integrate them in a single server.

Figure 5.1 shows the UML diagram of the general architecture of MPath-MS. The inter-process communications are represented with arrows. The inter-module communications (different modules in the same process) are represented with diamond arrows, indicating a module is a collaborator of another module.

As shown in Figure 5.1, the client communicates with the Django HTTP server, which uses the implemented applications to handle the request and send a response to the client. Even though the applications are Django applications, they interact with the server operating system to compile the generated models, store them in the server file system, execute new simulation processes and manage them as independent processes running outside the Django platform.

Each application communicates with the operating system to execute different tasks:

- The PMGMP application generates and compiles Cadmium models (C++ programs) using the uploaded SBML files and stores them in the file system. Likewise, each time a new model is generated, the PMGMP application generates a JSON model diagram also stored in the file system.
- The Sim-RT-Runner creates new process in the server to run the compiled models stored in the file system and start simulations that will keep running until it stops them. In another hand, simulation results are directly stored by the simulation process in a separated MongoDB database, then, the Sim-RT-Runner reads the

MongoDB database outside the Django model system to obtain the simulation results in real time and show them to the client.

- The DEVSdiagrammer application uses the JSON model diagrams generated by the PMGMP application to show to the client a visual and interactive diagram of the generated models.

The platform has a Django front-end (View) to allow the clients communicating with the server. The clients use the front-end to achieve the following tasks:

1. Upload and manage SBML files.
2. Generate models using the uploaded files.
3. Visualize model structures.
4. Run simulations.
5. Visualize simulation results in real-time and download them in CSV format.

The client interaction flow is the following: the client starts by uploading SBML files that will be stored in the server. Once the SBML files are uploaded, the client can display the list of available SBML files and generate the desired DEVS models. At any moment, the client displays the list of generated models and starts simulations. Finally, once a simulation started, the client is redirected to the result visualization module, where a polling system is used to update the graphics of the simulation results in real-time. Simulations only stop when the client manually stops them.

The platform uses the model generator explained in section 5.2 to generate and compile SBML files. Currently, the user cannot make any modifications to the model generator, but the platform is stored in a GitHub project that can be downloaded, modified, and redeployed in a new server. Future work could consider adding the possibility of customizing the model generator from the client user interface by adding a new Django application in charge of handling the modifications.

To better understand the entire model generation and simulation flow, we use the example shown in Figure 5.2. The example shows a common use case, where an SBML file is first uploaded to the server, then the model is generated, and finally, the simulation starts. Once the simulation is started, the client is redirected to the real-time visualization module.

In Figure 5.2 we can see the entire client-server interaction flow, the client starts the communication by uploading an SBML file. The PMGMP application stores the SBML file and redirects the client to the “SBML file list” view. Next, the client sends to the PMBMP application a request to generate the model. For this purpose, the application consumes the already uploaded SBML file from the file system, generates the model and stores the generated model and JSON diagram in the file system. Once the model is generated and compiled, the PMGMP application sends a confirmation message back to the client. The client then goes to the “Generated model list” view and sends a request

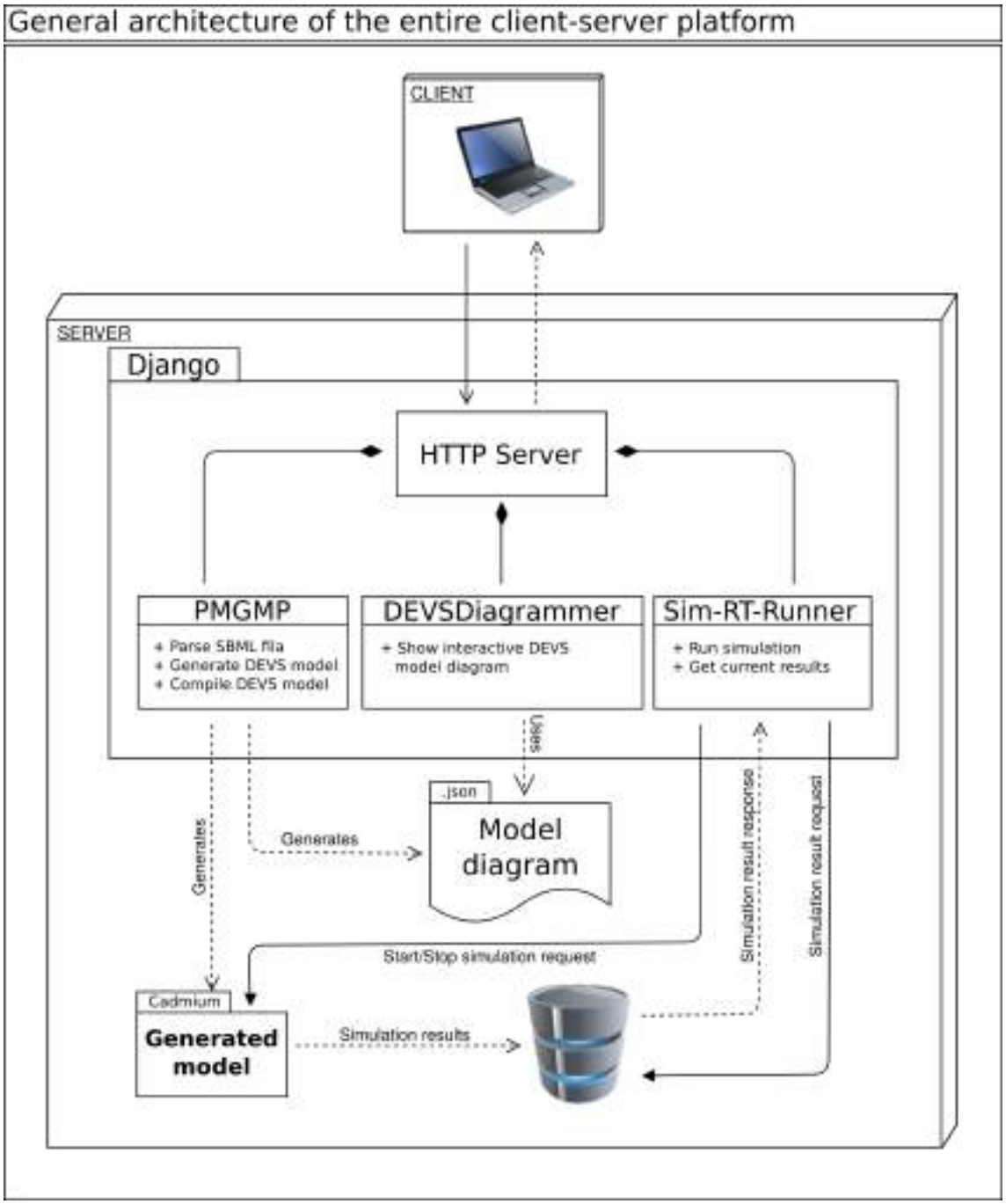


Fig. 5.1: The general web platform architecture diagram.

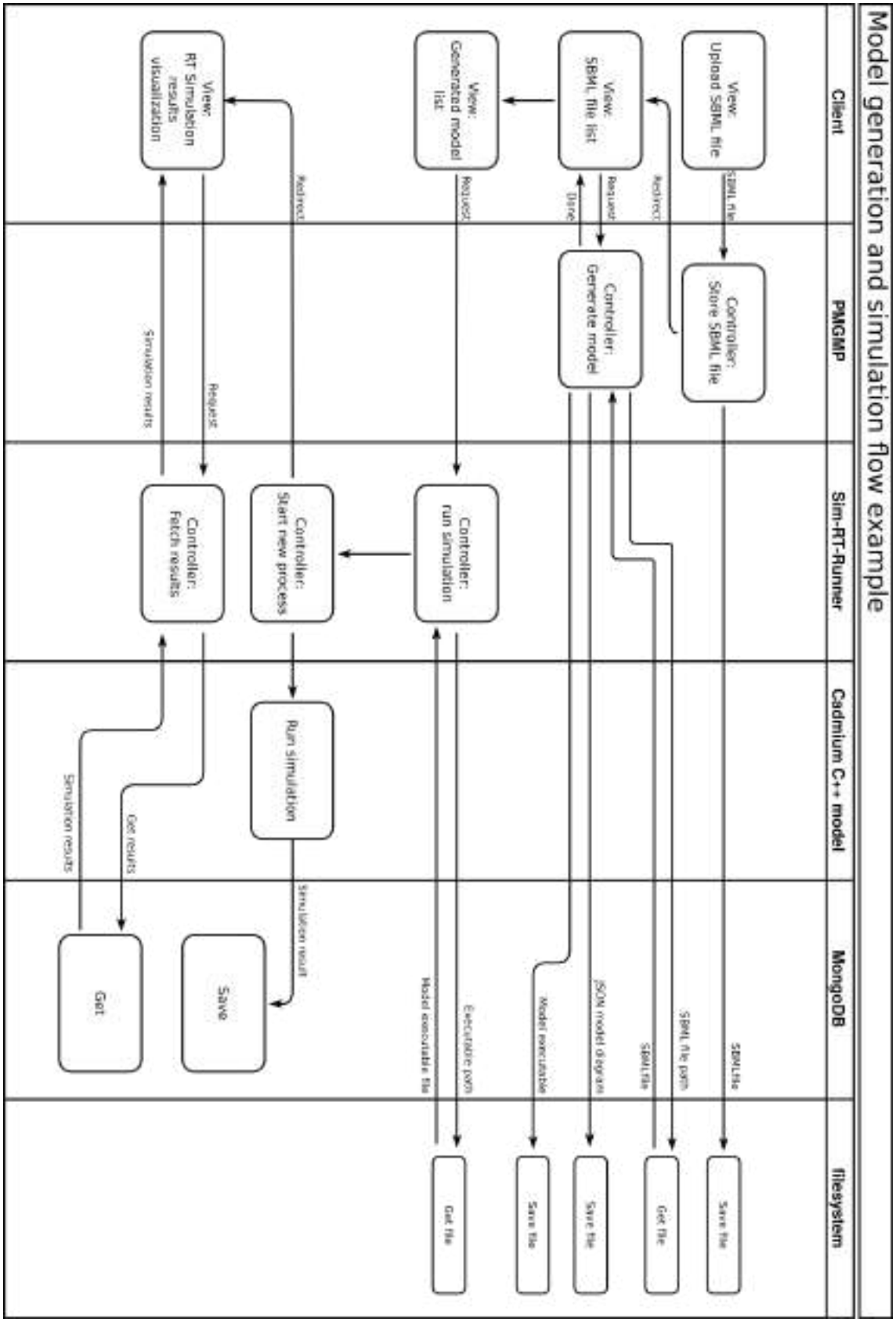


Fig. 5.2: Example of the flow of an entire model generation and simulation.

to start a simulation to the Sim-RT-Runner application. The Sim-RT-Runner gets and executes the model from the file system. Once the simulation is running, the simulation process starts storing the simulation results in the MongoDB database and the Sim-RT-Runner redirects the client to the "RT Simulation visualization" view, where the client start fetching the simulation results by sending multiple requests to the Sim-RT-Runner application. Each time the client sends a request, the Sim-RT-Runner application reads the MongoDB database to get the results stored by the simulation process and return the result to the client.

In this example, the client never displays the generated model DEVS structure. Because of this, the DEVSDiagrammer application is not part of the example process. But, whenever the client wants to check the generated model DEVS diagram, it goes to the "Generated model list" view and sends a request to the DEVSDiagrammer application to display the model diagram.

As we have already mentioned, the C++ simulation program (model) is an independent process running outside the Django platform. Then, in order to obtain the simulation results, the model stores the simulation results in a MongoDB database, so the Django platform can get the simulation results by querying the database outside the Django model system.

The platform User Interface (UI) is separated into different sections, each of which allows the user to manage a different stage of the automatic M&S process:

The SBML files manager section shown in Figure 5.3 displays the list of all the uploaded SBML files, and for each one, it allows to remove it (the red trash icon), download it (the green cloud) or to generate the corresponding DEVS model (the circular black arrows). If the model was already generated, a green tick is shown instead of the circular arrows to indicate the model ready. We can also add new SBML files using a form.

Once generated and compiled, a manager allows to select a model and run a new simulation. When the client starts a new simulation, the client is redirected to the real-time

Fig. 5.3: SBML file manager.

simulation visualization section shown in Figure 5.4. This module allows showing different graphics of the simulation results that are updated in real-time and. At any moment, the client can download the results in CSV format. The manager also allows the user to explore running and finished simulation results.

Finally, the DEVSDiagrammer (Figure 5.5) allows the user to display an interactive diagram of the DEVS models. We will explain the DEVSDiagrammer in more detail in section 5.4.

As simulations run as independent processes outside the platform, we use MeMoRe (Metric MongoDB Recorder) to persist the simulation results in a MongoDB database, which remains accessible by any other application connected to the database. For this purpose, the generated Cadmium model (compiled C++ program) takes as parameter a simulation ID. Each time the simulation generates a new output, it uses the simulation ID to persist the data with that ID in the MongoDB database. Then, each time the platform wants to retrieve simulation results, it uses the simulation ID to filter the results of the corresponding simulation.

Once the simulation is running, the server sends the ID to the client, and it redirects the client to the visualization section to start visualizing the results. The visualization uses the ID to poll the server for new simulation results, and it updates the chart. Each time a client requests new results to the server, it sends in the request body, the simulation ID and the last virtual time at which it already has the results, avoiding the server to resend already fetched results.

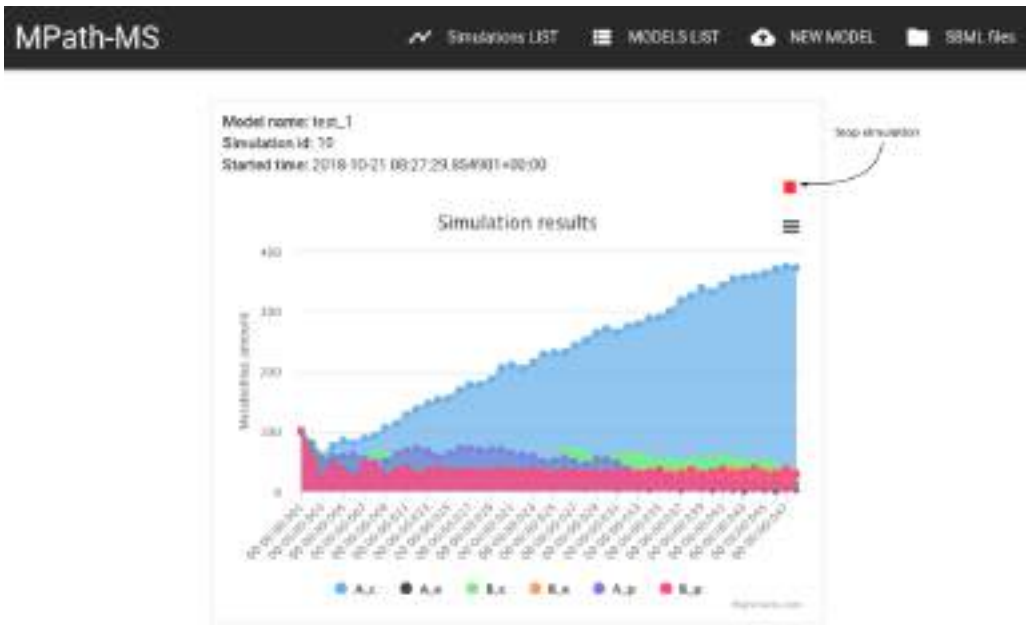


Fig. 5.4: The real-time visualization of a running simulation.

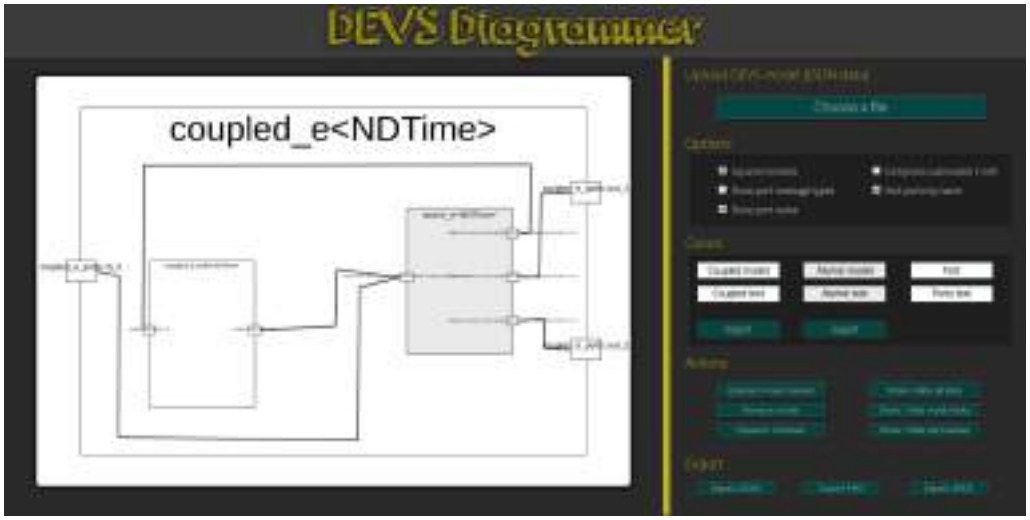


Fig. 5.5: The DEVSDiagrammer section user interface.

5.2 The instantiation procedure

As we have already mentioned, our platform uses SBML files to generate Cadmium DEVS models. Each Cadmium DEVS model is a C++ program that uses the Cadmium library, and implements atomic and coupled models within the program code. As we can see, to run a simulation, a C++ program must be compiled and executed. Here, we show how to do this in an automated fashion. First, we will introduce The System Biology Markup Language (SBML). Next, we will explain the information we use from the SBML files. Finally, we will explain the model C++ code generation process.

5.2.1 The System Biology Markup Language (SBML)

Because SBML main purpose is to store information of different biological phenomena, the specification language must be flexible. Therefore, SBML specification language is generic, and even though each SBML file must use the SBML structure, it can store any information needed. Because of this, an SBML model does not ensure that a specific information will exist in a file. Also, most biological models are macro-view models defined in a mathematical notation, and most existing SBML models do not include information about micro-view aspects of the cell (as for example, the Kon and Koff constants). In this work, we assume that the provided SBML file to generate the DEVS model will include all the information required. If this is not the case, we need first to complete the missing information.

The SBML general structure defines a simple hierarchy contained under the *Model* tag. In the first level, there are different lists of elements that can be present in biological processes (for example, compartments, species and reactions). In the second level, each list stores several single elements. For example, *ListOfReactions* contains multiple instances of *Reaction* elements. Every single element contains a sub-structure to store its information.

Additionally, an element can have a *note* attribute where any valid XHTML structure can be stored and where all the information that does not fit into the SBML specification structure is stored. Also, a list of parameters can be specified. The parameter list lets us adding custom values to the model that are not specified in common macro-view models.

Figure 5.6 shows a general structure of an SBML v2.5 file. Black diamond arrows indicate element relations with their cardinality. The $*$ symbol in the cardinality means it can be zero, one or multiple elements. The lists of the SBML structure that have the information we use to automatically generate our micro-view model are those in the gray boxes. Element names are surrounded by <> to indicate they are XML tag elements.

The SBML structure of a Reaction element is shown in Figure 5.7. Black diamond arrows indicate element relations with their cardinality, while white arrows indicate element inheritance. The $*$ symbol in the cardinality means it can be zero, one or multiple elements. All the elements within the gray boxes are necessary to generate the DEVS models, the remaining elements are simply ignored when using the SBML file. Element names are surrounded by <> to indicate they are XML tag elements.

In Figure 5.8 we can see the Species SBML attributes. SBML species elements represent the metabolite species and have no internal structure: They are just elements with attributes. Therefore, in Figure 5.8, there is a single XML element called *Species* with multiple attributes.

5.2.2 Obtaining information from the SBML files (the SBMLParser)

Now we will explain the SBML information we use to generate the models. There are informations that are explicitly defined in the SBML structure, and there are informations that must be derived from the relations of the SBML structure:

Explicitly defined information

On one hand, we need to obtain the model structure information (coupled models). For this, we use the ListOfCompartments shown in Figure 5.6, in which the different compartments are defined. The compartment types (Extracellular, Periplasm, Cytoplasm and Organelle) are declared in the ListOfCompartmentTypes. Thus, using the compartment list with their corresponding types, we can obtain all the information needed for the DEVS model structure shown in Figure 4.1. On the other hand, reactions properties are declared in Reaction elements within ListOfReaction.

The reaction information needed for the model proposed is declared in the Reaction structure shown in figure 5.7 as follows:

- The reaction id is obtained from the *id* attribute, which is a string id (Sid). It allows us to identify the reaction within the model.

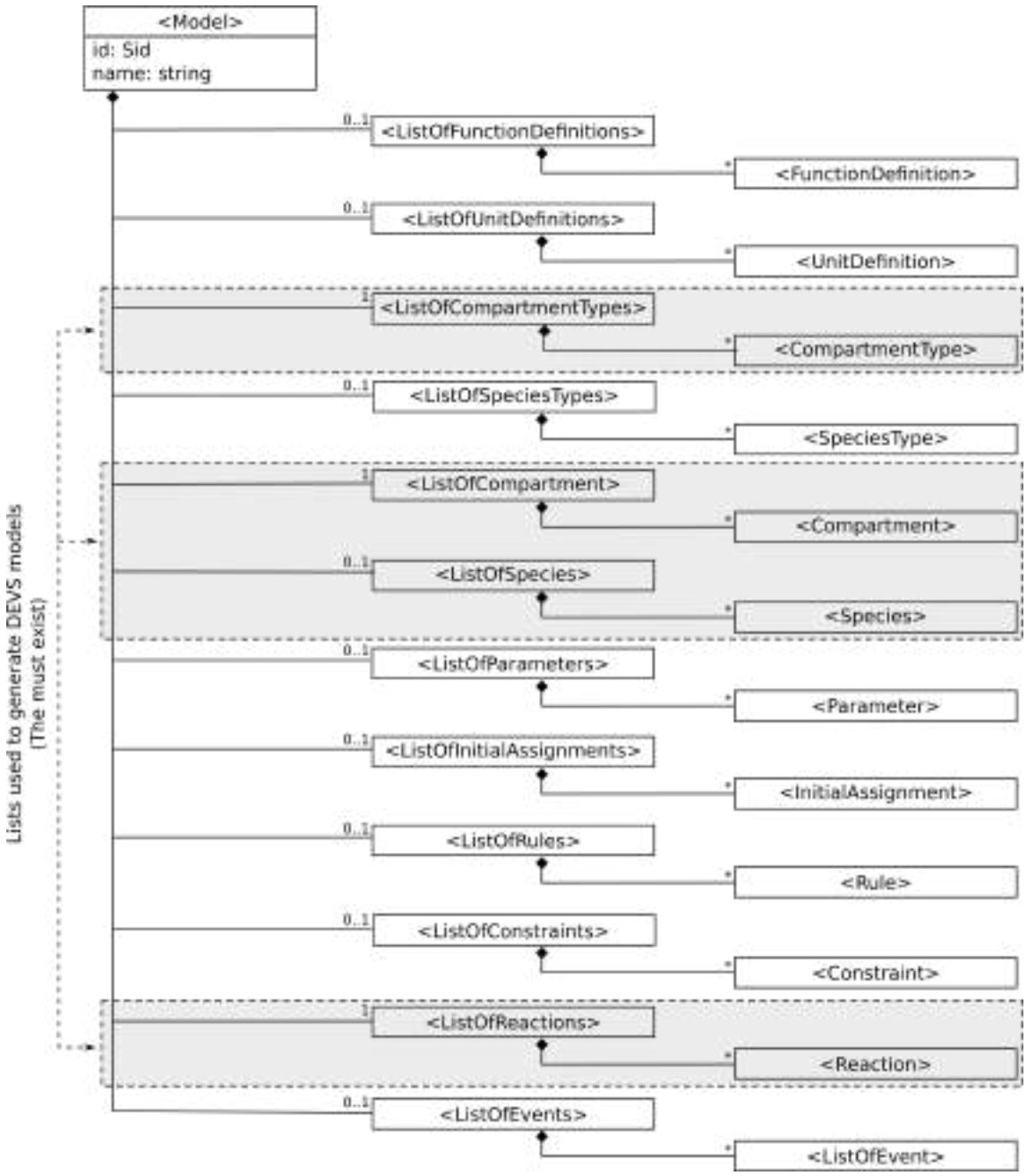


Fig. 5.6: General structure of an SBML v2.5 file and all available lists and elements [40].

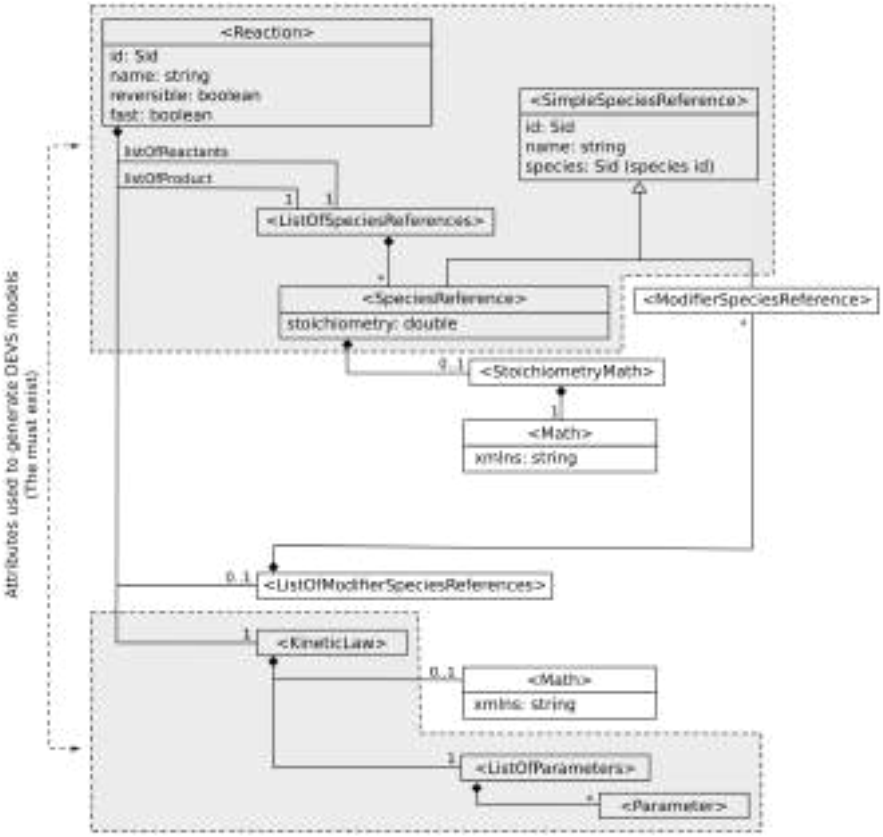


Fig. 5.7: The SBML Reaction element structure [40].

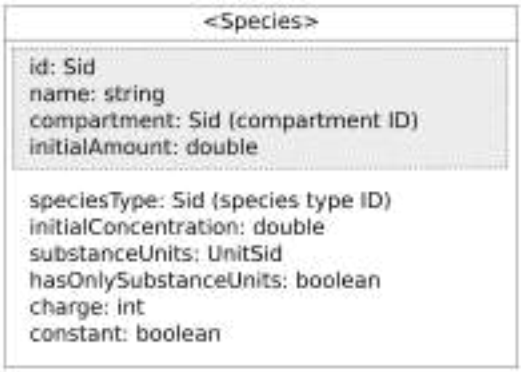


Fig. 5.8: The SBML Species element structure [40].

- Whether the reaction is reversible or not is defined in the *reversible* attribute (true means that the reaction is reversible).
- The reaction stoichiometry is defined in the *listOfReactants* and *listOfProducts* children elements, which are lists of species belonging to the stoichiometry. Each species within these lists has a *stoichiometry* attribute, which is a number indicating the amount of that species contained in the stoichiometry.
- The remaining properties are defined in the *ListOfParameters* contained in the KineticLaw child element. These elements are:
  - The reaction and reject rate indicating the time that takes to react and reject metabolites.
  - The Kon association constant.
  - The Koff disassociation constant.

As we can see in Figure 5.7, in the SBML specification there are more elements available than those we use. Not used elements (those outside the gray boxes) are simply ignored.

Information obtained from the SBML structure

As mentioned earlier, SBML is most commonly used for mathematical macro-view models where the physical structure of biological cells is not considered. Therefore, this information is not present in the SBML model. Because of this, we need to obtain that information from the structure present in the SBML model.

In order to generate the proposed model from SBML files, we need to know which enzymes catalyze each reaction, and in which compartment each of the enzymes is. The SBML specification structure does not considers enzymes. Enzymes are specified in the reactions using logical expressions to denote the enzymes' name. For example, the logical expression in Formula 5.1 indicates the reaction is handled by the enzymes $b0854$ and $b1126 - b1125$. Notice that a reaction can be handled by different enzymes and a single enzyme can handle different reactions at different times.

$$(b0854 \; or \; (b1126 \; and \; b1125) \tag{5.1}$$

To know to which compartments each enzyme belongs, we could use the compartment to which the reactions handled by that enzyme correspond. Nevertheless, in SBML all the reactions are listed in the ListOfReaction without being separated by compartments. Because of this, we need to deduce the reaction compartments from the reaction stoichiometry as following: on one hand, species are compartment-specific elements, if a metabolite can be found in multiple compartments, the metabolite must be declared multiple time using different species IDs for each compartment (normally by adding the compartment name as suffix). On the other hand, the stoichiometry of the reactions determines the species a reaction consume and produce. Then, we can deduce the reaction compartment from the reaction stoichiometry species by examining where its product and substrate belong to. For example, if a reaction stoichiometry has species from the extracellular, periplasm and cytoplasm compartments, then, according to section 4.1, the reaction must

belong to the periplasm trans-membrane. If the reaction consumes species from a unique compartment, then the reaction belongs to that compartment. If the reaction consumes species from two or more compartments, then, one of those compartments must have a membrane that communicates them, and the reaction belongs to that membrane. The species information used from the SBML *Species* element are the ones whitin the gray boxes in the Figure 5.8:

- *id*: An identification attribute that allows us to differentiate each metabolite (species) during the simulation.
- *name*: Allows us to show the real biological name of each metabolite when plotting the simulation results.
- *compartment*: Contains the compartment ID where the species belongs.
- *initialAmount*: Specifies the total number of metabolites of that species existing in the simulation at the starting time.

The remaining attributes (outside the gray box) are not used to generate the DEVS models, and they are ignored. As we can notice in Figure 5.8, some of these attributes (as for example the "initialConcentration" and "SubstanceUnits") specify information needed by macro-view models.

The number of enzymes available at the simulation departure point is not mentioned in the SBML structure and must be given as parameter by the modeller each time it runs the simulation. Removing that information from the model allows to easily experiment with different scenarios. The initial amount of each species (metabolite) can be specified in the SBML model, but to allow to modify the scenario, we let the modellers to redefine those values in the simulation parameters explained later in this section.

In order to generate the DEVS model, we must first get all the information from the SBML file, process it and store it to be later accessed when generating the model. For this purpose, we have defined the data structure shown in Figure 5.9.

Figure 5.9 shows an UML diagram of the data structure used to store the processed information. The white arrow indicates that the product and reactant elements have both the same structure (metabolite by compartment). As in the parsed data structure, a reaction is handled by multiple enzymes and an enzyme can handle multiple reactions.

The structure shown in Figure 5.9 allows us to filter reactions and enzymes by the compartment they belong to using the location as the grouping attribute. As we have already mentioned, the reaction location is calculated from the reaction stoichiometry using the species associated compartment. Likewise, the reaction parameters used to create the enzyme atomic model instances are placed in the parameter structure that is related to each reaction.

Space models need the list of related enzymes (I.e. those enzymes handling reactions that consumes and/or produce species from it). These related enzymes can belong to

any compartment, for example, enzymes in the periplasm outer membrane consume and produce metabolite from and to the extra cellular space, then, the extra cellular space is related to enzymes of the periplasm compartment. For this purpose, in the parsed data structure (Figure 5.9) we have separated the reactions stoichiometry by its species compartments. The result of the separated stoichiometry is stored in the *product by compartment* and *reactant by compartment* structures. Using the separated stoichiometry, we can filter all the reactions related with a compartment and then, we can use the obtained reactions to get the enzymes related to that compartment.

To separate the stoichiometries by compartments we use dictionaries with the compartment id as keys and chunks of stoichiometry as values, in other words, the reaction stoichiometry is divided by the species compartments within the stoichiometry formula. This separation not only allows us to determine the related compartment of a reaction, but it is also needed by the enzyme atomic model explained in section 4.2.1 to know where to send the produced metabolite.

The SBML file parser works as follows:

- An XML parser is used to store the SBML file data in a Document Object Model (DOM) structure easy to travel.
- The parser obtains the compartments information from the *ListOfCompartments* and *ListOfCompartmentsType* to generate the compartments, reaction sets and locations elements.
- Reactions are parsed from the *ListOfReactions*, and for each reaction, the parser gets the reaction information from the structure shown in Figure 5.7 as follows: It obtain the reaction stoichiometry from *ListOfReactant* and *ListOfProduct*. It check if the reaction is reversible or not checking the *reversible* attribute. Then, all the kinetic information is obtained from the *ListOfParameter* that is in the reaction *KineticLaw* element, this information includes: $K_{on_{STP}}$, $K_{on_{TPS}}$, $K_{off_{STP}}$, $K_{off_{STP}}$, *Reaction rate* and *Rejection rate*.
- Once the reaction information is ready, the product and reactant are separated by compartments. Finally, the reaction location is deduced from the reaction related compartments.

As we have already mentioned in section 5.2.2, enzymes are not declared in the SBML structure, but as logical expressions in the *Note* element of each reaction (as shown in Formula 5.1). The result of evaluating the logical expression is a list of enzyme names that handle the reaction. Then, when a reaction is parsed, its enzyme logical expression is evaluated, and the names obtained are used to create the enzyme components that will model the reaction. Because an enzyme can handle multiple reactions, during the process of creation of new reaction enzymes, we can find that some of the models already exist. In this case, we do not duplicate the model; instead, the reaction is added to the list of reactions that the existing enzyme model handles. Recall from section 4.2.1, that an enzyme is no more than a list of reactions that can occur one at a time. This is why we

only need the enzyme name to create an enzyme model.

For large models with many reactions and enzymes, the parsing stage can take a long time (several minutes). In order to avoid parsing again the same SBML file in future usages, the parsed structure can be serialized as a JSON object and be later loaded to reuse them (using a serialized ".pickle" file).

5.2.3 The DEVS model generation processes

Once the SBML file is parsed, we are ready to generate the model. Next, we will explain the model structure generation process that uses the explained data structure to access the pre-processed SBML information.

Let us consider the structure used to generate the data of the compartment models. The model generation process first generates the data for the models using this structure and then, it generates the model C++ code and the parameters XML file with the model starting state.

The compartment model data is represented by the ModelStructure class and it is composed of the following data:

- **Id**: The compartment model id.
- **Enzyme_sets**: sets of enzymes within the same membrane or the compartment inner enzyme set.
- **Routing_table**: A table that uses the enzyme location to determine for each enzyme, which port must be used to reach the enzyme using the routing system we have already explained.
- **Membrane_EIC**: The external Input Coupling mapping between the membranes and the model input ports. If no membrane is present, then, this is empty, and the input ports are all directed to the compartment space atomic model.
- **Space_parameters**: The space atomic model parameters:
  - **cid**: compartment id.
  - **Interval_time**: the space fixed-length time step at which the model time advances. This is a parameter that is not specified in the SBML file and must be provided by the modeller.
  - **Metabolites**: The compartment species list.
  - **Enzymes**: The compartment related enzymes, which are all the enzymes where the compartment send and/or receive metabolites from.

The routing table is generated from the related enzyme locations. Similarly, Membrane_EIC is generated by assigning links from each membrane to a new input port. The

EIC mapping is then used to link the compartment membranes with other compartments.

The remaining data is obtained from the parsed data structure shown in Figure 5.9. The atomic model space parameters are obtained from the compartment parameter elements, and the compartment enzyme sets are generated from the enzyme set elements.

Now that we have defined the data structure of the compartment models, we will explain the DEVS model generation, which is a top-down process that starts creating the top cell coupled model and continues constructing the sub-components recursively until the bottom elements are built. Once all sub-components are ready, the cell top model is finished.

The cell top model is composed by the Extracellular, Periplasm, Cytoplasm and organelles compartment models. These compartments must be generated before creating the cell top model. For each compartment (Extracellular, Periplasm, Cytoplasm and Organelles), first its ModelStructure is created and next, the compartment model is generated as follows: The compartment atomic models are created using the *enzyme_set* and *space_parameter* attributes. Once the compartment atomic models are ready, the EIC, IC and EOC links are generated as shown in Figures 4.2, 4.3, 4.4, 4.5. Finally, once the compartment coupled models are ready, the cell top model EIC, EOC and IC links are generated using the structure shown in figure 4.1 and the entire model is ready.

Figure 5.10 shows a diagram of the top-down recursive strategy to generate the models as a rooted tree graph where the root is the top model generation process and each level defines the process recursive hierarchy. Each line link in Figure 5.10 represents a recursive call to a function that generates a sub-model that is a dependency of the parent model.

Cadmium models are C++ programs, and the model generator must generate C++ code ready to be compiled and executed. To write the final C++ code, the model generator uses the modelCodeWriter module, which uses parametrized C++ templates to build .cpp files. A parametrized C++ template is just a Python string template [41] that can be formatted using the *format()* method to replace some parametrized part of the string with custom strings. For example, the template *"int {a} = {b} + {c}"* is a simple C++ template where a, b and c are the parameters that will be replaced with the strings passed as the parameter to the *format()* method (bracket symbols are used to define the template parameters within the string). Then, using this template, we can write any C++ code that adds two operands and store the result in an integer variable. An example of the code generated with this template is as follows:

- *"int {a} = {b} + {c}".format(a="var1", b="var2", c="var3")* → *"int var1 = var2 + var3".*
- *"int {a} = {b} + {c}".format(a="var1", b="2", c="3")* → *"int var2 = 2 + 3".*
- *"int {a} = {b} + {c}".format(a="var1", b="(2 + 3)", c="var1")* → *"int var1 = (1 + 2) + var1".*

The decision of implementing the model generator in Python as a code writer module, instead of generating the model directly in the same C++ program that runs the model,

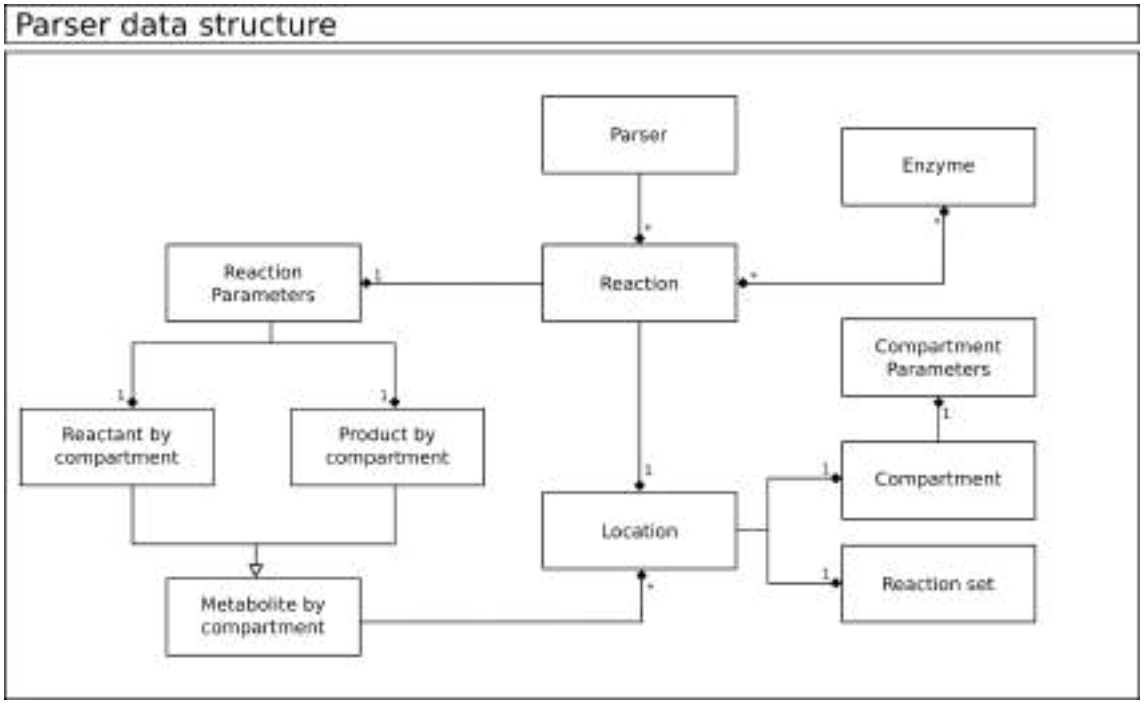


Fig. 5.9: UML diagram of the parser data structure to store the processed data obtained from the SBML file.

has two main purposes. On one hand, Python is a high-level language with more libraries ready to parse XML files, and it allows us to build code easier to maintain. On the other hand, if we generate the model in the same C++ code that runs the model, then, each time we execute a simulation, the model needs to be generated, which is very inefficient.

We have defined different C++ templates. There are templates to define atomic and coupled models, ports and links. The model generator generates the model and uses the ModelCodeWriter to write the models code in a correct order in order to satisfy the dependencies. When the model is generated, we obtain the top_model_definitions.hpp and top.cpp files with the correct C++ code for the top models and all its sub-components.

In Cadmium, atomic models are defined as classes built by the modeller. These classes must implement the internal, external, confluence, time advance and output functions that Cadmium will call at simulation runtime. We defined general atomic model classes as explained in section 3. These .cpp are used by ModelCodeWriter which includes them in the generated *top.cpp*. The template code to generate an atomic model contains the definition of a new C++ class derived from the corresponding general atomic class. The atomic model name is defined as the new class name. Then, the new atomic model class inherits the internal, external, confluence, time advance and output functions from the general atomic model class, and it becomes an instance of that general model class as we have explained in section 3.

In Cadmium, coupled models are instances of the *cadmium::dynamic::modeling::coupled*

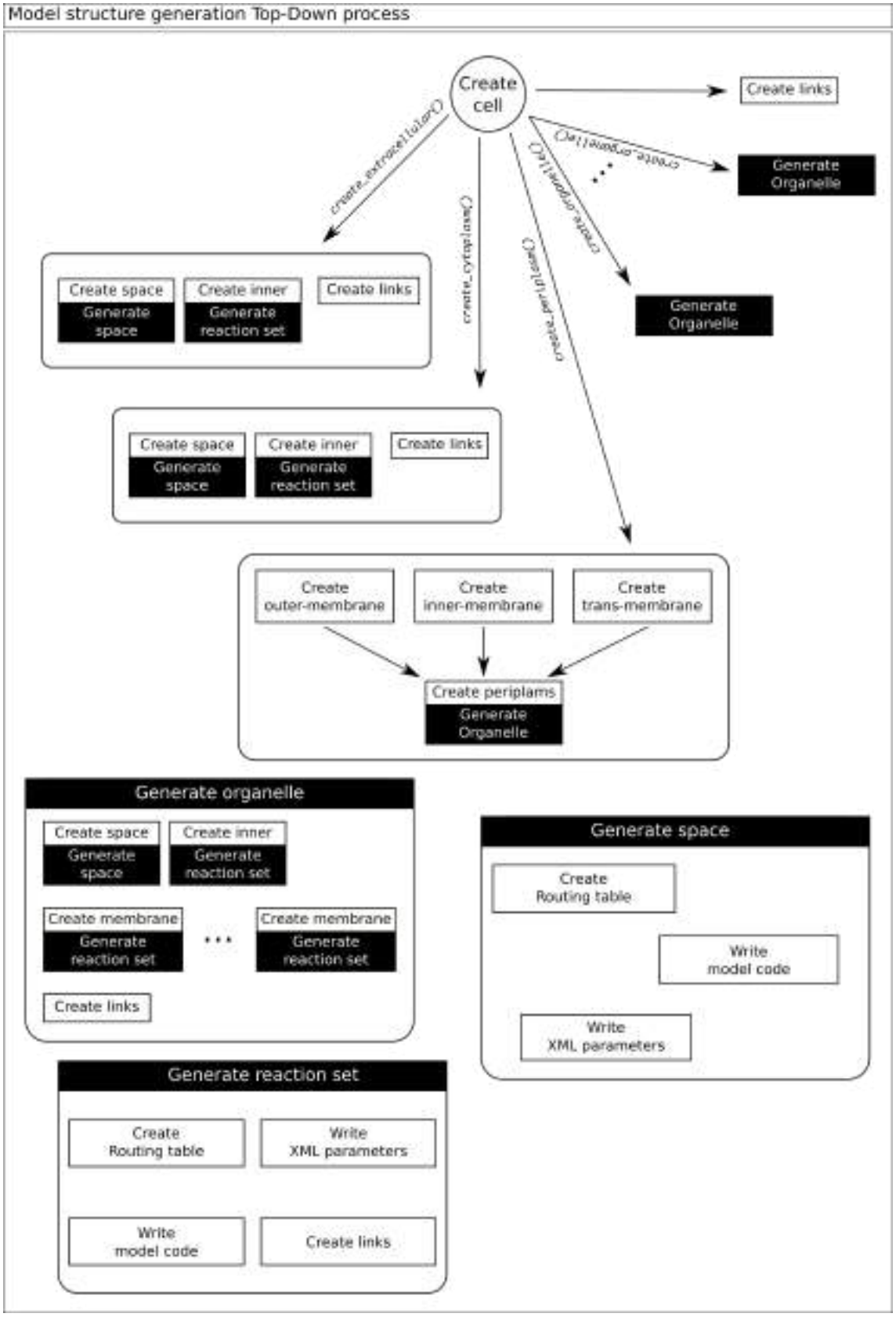


Fig. 5.10: The Top-down model generation process structure.

class. The code of any C++ Cadmium coupled model is almost the same. The only parts that change are the parameters used to construct the coupled model using this function, i.e.:

- The sub-model names.
- The message type of each port.
- The model input and output ports.
- The model links.

The C++ template defines a new coupled model instance with this function using custom parameters, and it assigns the coupled model to a variable with the model name. It is important that the code of a coupled model is executed after all its ports and sub-models are created. If this is not the case, the code will never compile due to undefined variables. Then, to generate a coupled model we first use C++ templates to generate the model ports and links code which are instances of the Cadmium ports and links classes.

The code of each model is generated just after the model generator has generated the model using the recursive strategy we already explained (Figure 5.10). Then, every coupled model code will be written after all its sub-models code is written and the dependencies are satisfied.

To better understand the recursive model code writing process, we will use the following example: We have a coupled model A with two sub-models B and C, where B is atomic and C is a coupled model composed of two atomic models D and E. Then, the model generator will first try to generate the model A, but using the recursive strategy, it first needs to generate the models B and C. For this purpose, it will use the atomic model C++ template to write the code of the model B, and after that, it will generate the model C. Because the model C is a coupled model, it will first recursively write the code of the atomic models D and E using the same template used to write the code of the model B and then it will write the code of the coupled model C that uses the models D and E (already written). Finally, it will write the code of the model A that uses the code of the models B and C (already written). To write the model A it will use the same C++ templates used to write the coupled model C.

When the ModelCodeWriter is initialized, the first thing that happens is that the top_model_definitions.hpp file is created to write all the sub-model and port definition codes and, next, the top.cpp file is created to write the final TOP model code. After these files are ready, all the dependencies' include statements are written. The dependencies are the Cadmium libraries and the general atomic model classes. Finally, the created *top_model_definitions.hpp* file is included in the model *top.cpp* file to make the definitions written by the ModelCodeWriter accessible from the TOP model code. The initial code of the model file (*top.cpp*) is shown in Listing 5.1.

Listing 5.1: top.cpp

```
/** include general dependencies **/
#include <typeinfo>

/** include Cadmium library **/
#include <cadmium/modeling/dynamic_coupled.hpp>
#include <cadmium/modeling/dynamic_model_translator.hpp>

/** General atomic model classes **/
#include <pmgbp/atomics/reaction.hpp>
#include <pmgbp/atomics/router.hpp>
#include <pmgbp/atomics/space.hpp>
#include <pmgbp/model_generator/reaction_set.hpp>

/** include the created .hpp file where the ports code will be written **/
#include "top_model_definitions.hpp"
```

Once the initialization is done, the ModelCodeWriter is ready to start defining models and ports and the model generator module will use it to create the models using the recursive strategy we have already explained.

The custom parameters of each atomic model obtained from the SBML file are written in an XML file. Each time the model generator generates a model, it writes the model code using the ModelCodeWriter and it writes the model custom parameters obtained from the SBML model in the created XML. The main purpose of using this system is that now, the simulation starting point values (as the initial amount of each metabolite) are set in a separated XML file that is consumed in run-time. This allows the modeller to modify the initial condition of the simulation to obtain different scenarios without the need for regenerating and compiling the model. The XML parameter file is written in a more specific and readier to consume format, so the models will easily access it.

As we have already explained, each DEVS atomic model is an instance of a general atomic model class explained in section 3. Cadmium provides a function to generate a new atomic model instance from a C++ class, which takes a variable number of parameters that will be forwarded to the class constructor. To create a new atomic model, we use this function and we pass as parameter the model id and the path to an XML file where all the atomic model parameters are specified. The general atomic model classes' constructors use the XML to read the instance parameters from there.

XMLParameterGenerator defines all the model parameters in the XML file. This module is similar to ModelCodeWriter, and both are used when a model is generated. Once the model is generated, its parameters are set in the XML file. This module uses a python XML library to generate all the parameters in the program main memory, and after the model is generated it flushes the generated XML structure into a file used by the atomic model constructors.

There is a single XML file where all the atomic models' parameters are defined. Using a single file for all the atomic models reduces reading time, a slow task that involves accessing the hard drive. Then, the general atomic model classes' constructors use the model id to find its parameters within the XML file. Finally, we define the XML file path in a variable

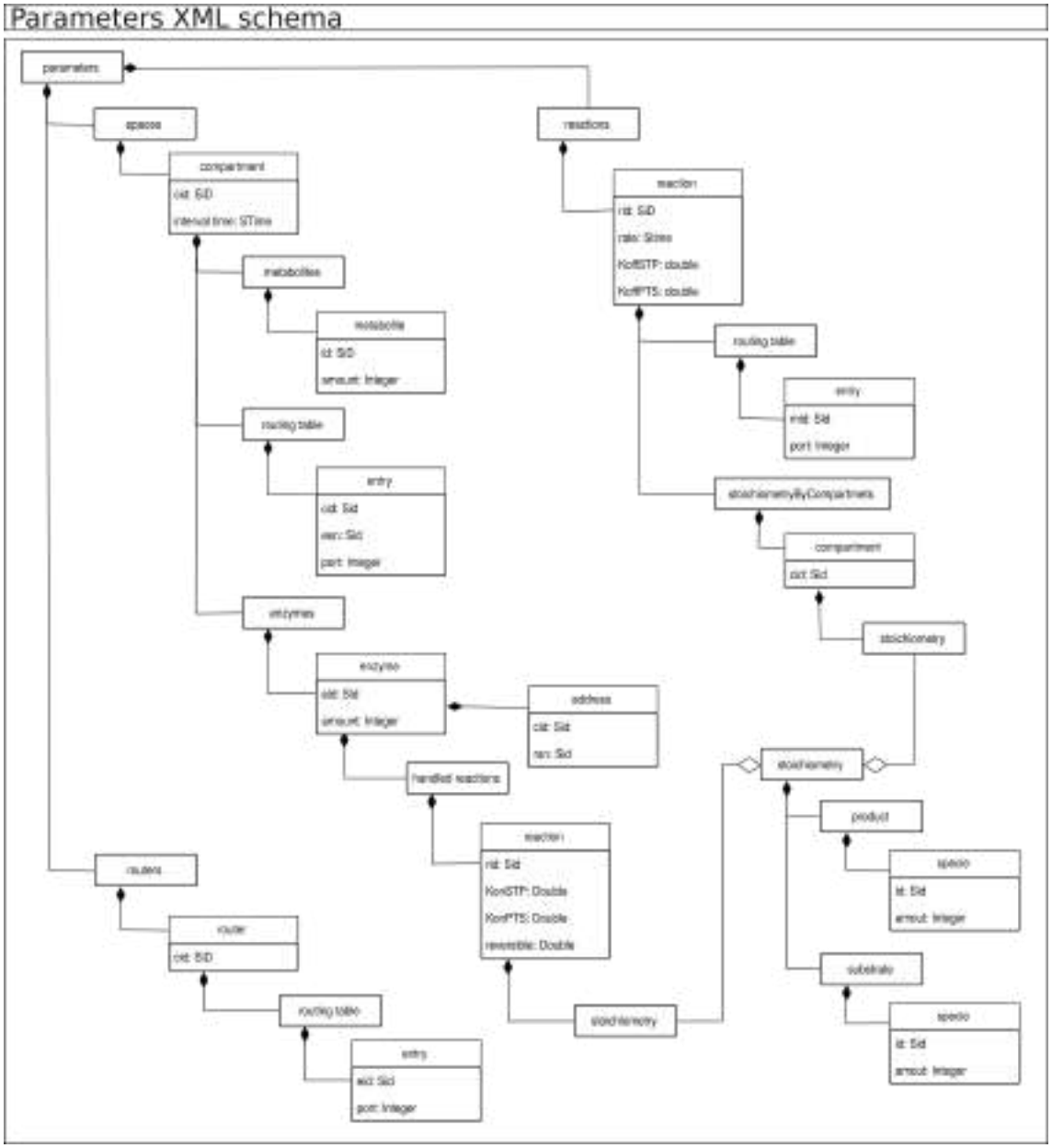


Fig. 5.11: The UML diagram of the XML parameters file.

that is parametrized in the final executable program to allow executing the simulation with different XML files. This allows us to define different simulation scenarios in different files.

Figure 5.11 shows the general XML parameter UML diagram where all the mentioned parameters in section 4.2 are specified. The root element is the *Parameters*. This element contains 3 main lists: *spaces*, *reactions* and *routers*.

Space atomic models' parameters: All the parameters of a space atomic model are located in a *compartment* element within the *spaces* list element.

- The *cid* attribute has the model ID used by each model to find its corresponding parameters.
- The *interval time* attribute defines the fixed-length interval time steps used by space atomic models. This attribute is a STime, a string to define time with the format "hours:minutes:seconds:milliseconds".
- The *metabolites* is a list containing the information of the metabolites that belong to the compartment. Each metabolite information is specified in a *metabolite* element within this list and has the following attributes:
  - *id*: Identifies the metabolite across the models. When a metabolite is sent from one model to another, the id is sent to identify it.
  - *amount*: Is an integer attribute that defines the initial amount of metabolite available in the compartment.
- The *routing table* is a routing table that specifies for each pair of compartment ID and enzyme set ID, which output port must be used to send metabolites to an enzyme. Each *entry* element has the following attributes:
  - *cid*: The id of the compartment where the enzyme set is located.
  - *esn*: The id of the enzyme set.
  - *port*: The number of the port that must be used to reach the enzyme set.
- The *enzyme* contains the information of the enzymes that are related to the compartment. Each enzyme information is specified in an *enzyme* element within this list and has the following attributes and children:
  - *eid*: Identifies the enzyme across the models.
  - *amount*: Is an integer attribute that defines the initial amount of free enzymes (I.e. an enzyme without any metabolite bound) in the compartment.
  - *Address*: It is a child element containing the compartment and enzyme set IDs where the enzyme belongs.
  - *handled reactions*: It is a child element organized as a list with the information of all the reactions handled by the enzyme. For each reaction, it includes a *reaction* element with the following attributes and children:
    - *rid*: The id of the reaction.

  - *KonSTP*: The Kon constant for the Substrate to Product direction.
  - *KonPTS*: The Kon constant for the Product to Substrate direction.
  - *reversible*: A boolean indicating if the reaction is reversible or not.
  - *Stoichiometry*: A child element that has two lists of species (*Product* and *Substrate*) that specifies each one of the species id and amount that belongs to the reaction stoichiometry.

Enzyme atomic models' parameters: An enzyme is basically a list of handled reactions. Thus, to instantiate an enzyme atomic model we need all the parameter of the enzyme handled reactions. Each reaction information is contained in a *reaction* element whithin the list *reactions* and has the following attributes and children elements:

- *id*: The reaction id.
- *rate*: The time that takes the reaction to convert the consumed metabolites to the produced metabolites.
- *KoffSTP*: The Koff constant for the Substrate to Product direction.
- *KoffPTS*: The koff constant for the Product to Substrate direction.
- *Routing table*: is a list containing routing table entries that indicates which port must be used to send each metabolite to its corresponding space atomic model. Each entry has a mid (metabolite id) and a port number.
- *ListOfStoichiometryByCompartment*: is a list of stoichiometry divided by compartments as we have already explained. As we can see in Figure 5.11, this list has a child for each compartment, and each compartment has a single child where the reaction stoichiometry related to that compartment is specified.

Router atomic models' parameters: All the parameters of a router atomic model are contained in a *router* element within the *routers* list element. A router has an *id* attribute to identify the model and a *routing table* element. Each child of the *routing table* element is an *entry* elements that specify for each enzyme id (the *eid* attribute) the port where the message must be routed to reach the enzyme (the *port* attribute).

As we can see, the model generator module uses three collaborators: SBMLParser, ModelCodelWriter and XMLParameterGenerator. This module first initializes the model structure using the parser to fetch and interpret the model information from the SBML file and then generates the model and writes the model C++ code and the parameter XML file using ModelCodeWriter and XMLParameterGenerator.

The complete model generator process works as follows. The model generator initializes the SBMLParser, ModelCodeWriter and XMLParameterGenerator. Then, it initializes ModelStructures using the top-down strategy in Figure 5.10. Each time an atomic model is defined, its parameters are added to the XML structure. Once the entire model is built, ModelCodeWriter and XMLParameterGenerator close their files and the model is ready to be compiled.

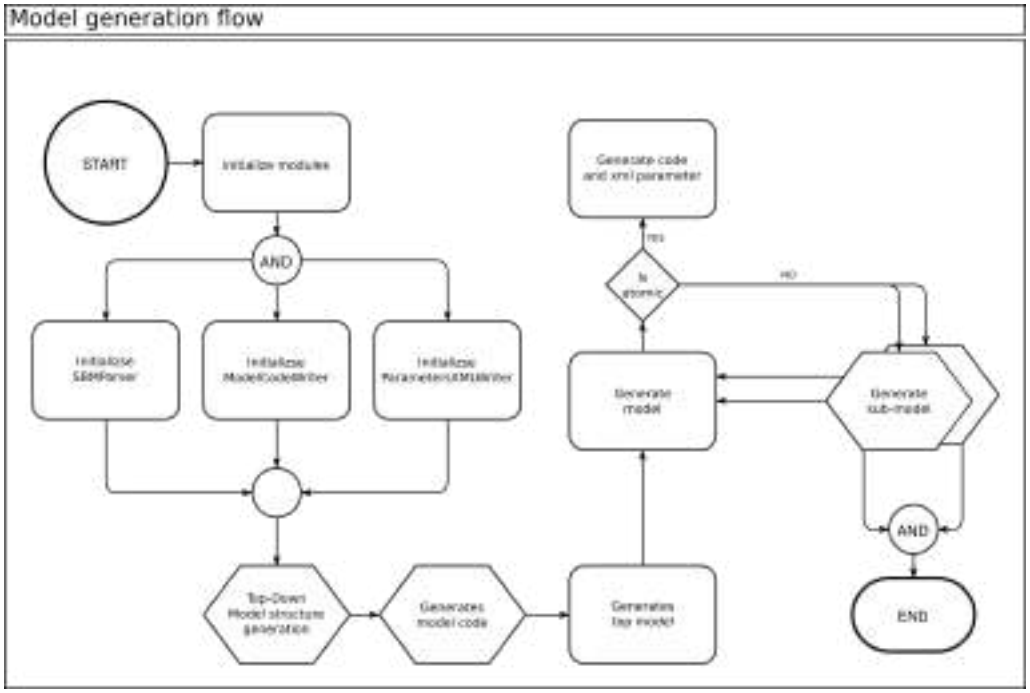


Fig. 5.12: The recursive model generation flow.

Figure 5.12 shows the model generation flow that we have just described. For each model, if it is an atomic model, the code is written, and the parameters are added to the parameters XML structure and the model is ready. If the model is a coupled model, all its sub-models must be generated first.

5.3 The NDTime C++ class for DES purposes

DEVS is a DES (Discrete Event Simulation) formalism based on continuous time, meaning that for every virtual time t of the simulation there is a finite set of events occurring simultaneously, however, the time representation remains continuous and events can occur at any moment within the continuous timeline. The formalism defines the time within the positive real numbers set, thus, a virtual time t could be any positive real number.

However, models can use subsets of the real numbers to represent their virtual time. For example, a DEVS model could use a clock representation of the time where seconds is the minimum unit of time to schedule transitions. If this is the case, using integers to represent the seconds from a determined initial time is more than valid to represent the time. Even some physic models that do not use irrational numbers are able to use a clock-like time representation if very small units of time are allowed. For this purpose, we have implemented Natural Deep Time (NDTime), a simple C++ class for time representation in DES models.

Clock-like time can easily be represented with integers using timestamps. Nevertheless,

to avoid fast overflow during simulations, we have separated time in hours, minutes, seconds, milliseconds, microseconds, nanoseconds, picoseconds and femtoseconds. For each unit of time, we use a different variable. The hours are represented with a long unsigned integer. The minutes and seconds are sexagesimal fractions, and they are represented with integers between 0 and 59. The remaining units represent one-thousandth of the next greater unit and are represented with integers between 0 and 999.

DEVS simulators virtual times must allow additions and substraction. Likewise, because of some simulation checks, negative time should be available. NDTime uses a sign Boolean to determine whether the represented time is positive or negative.

5.4 The DEVSDiagrammer to export and visualize Cadmium models structure using JSON

Cadmium is a header only simulator easy to integrate into any C++ project, but it has no user interface. To allow modellers to diagram their coupling model architectures, we have implemented DEVSDiagrammer, a C++ module to export Cadmium model structures into JSON files and a JavaScript interface to visualizes the exported JSON.

The DEVSDiagrammer is divided into two modules: the first is a C++ header only module that can be integrated alongside with Cadmium, it consumes the model, and produces the model structure in JSON format. The second module is the web DEVSDiagrammer, which consumes the JSON structure and displays a coupled model.

The C++ module recursively searches the coupled model components; for each component, it plots an empty structure with ports if it is an atomic model, or it constructs a JSON structure if it is a coupled model. Once all the components' structures are ready, the DEVSDiagrammer iterates the EIC, IC and EOC model structure to export the model links and ports.

The web platform uses CreateJS [42] to plot a visual diagram of the model. This platform displays the model in a white canvas as a big box that contains smaller boxes for the model sub-models. Links are drawn as lines that connect the models' ports. Ports are drawn as lite squares on the model and components edges.

DEVSDiagrammer has multiple functionalities (shown in Figure 5.5) that allow the user to handle the model in order to obtain a clean visualization and to export the obtained diagram as images. Likewise, the user customizations can be downloaded as new JSON structures. The users can use these customized structures to visualize the model in the future.

Next, all the functionalities are listed and briefly explained:

- **Options:**
  - **Squared models**: If checked, all sub-models are displayed as squared; if not checked, sub-models can be rectangles.
  - **Show port message types**: Whether to display or not the port message's type name. Printing the message type is useful to debug links, but can add significant noise to the diagram; hiding allows focusing on other aspects of the model.
  - **Show port name**: this is similar to Show port message type, but with the port names. Port names are the port type names.
  - **Compress submodels in left**: Sub-models are automatically positioned within the model to use the space according to their links. For this purpose, a toposort algorithm is used over the model links graph to minimize the number of links connecting models from left to right. The "compress sub-models in left" feature is an algorithm that runs over the toposort to place each model at the most left valid position without breaking the topological sorting. Note: besides this automated model positioning system, users can drag and drop models in custom positions within the canvas.
  - **Sort ports by name**: Used to sort model ports in alphabetical order.
- **Colours:**
  - Set coupled, atomic and ports background and text colours.
  - Export colours for later use.
  - Import already exported colours.
- **Actions:**
  - **Expand in new canvas**: All the selected models and sub-models (by clicking the model) are drawn in a new canvas as a full model. This is a good option to expand a sub-component and work with its internal structure.
  - **Remove model**: Remove selected models from the canvas.
  - **Expand / Contract**: Toggles the select model and sub-models to display or hide their internal structure, this can be used to hide all the sub-models' internal structures and focus on the model's top level.
  - **Show / Hide all links**: Toggles the model links to display or hide them.
  - **Show / Hide model links**: Toggles all the links related to the selected models. First, the models to toggle must be selected. When a model has multiple links, this option is great to debug the couplings by parts.
  - **Show /Hide port names**: This toggles the ports names to be displayed or hidden.

- **Export**:
  - **Export as JSON**: Export model structure as JSON with the customizations.
  - **Export as PNG**: Export model as a PNG image.
  - **Export as JPEG**: Export model as JPEG image.

Likewise, links can be handled by holding some point in the link and dragging that point over the canvas. This allows links to have custom paths from the source port to the sink port.

5.5 MeMoRe (Metric MongoDB Recorder) to store simulation logs MongoDB

One of the goals of this thesis is to deploy the proposed platform in a remote server using Modeling and Simulation as a Service (MSaS) [43]. Multiple users could use the platform at the same time, and the overhead could make the server unresponsive. Deploying the platform in a cluster capable of balancing the simulation between multiple cores can improve this situation. On the other hand, simulation results must be available at the platform at any time. Saving simulation results in a single database for all the simulations running in the cluster can be useful. To deal with these issues we have used MongoDB, a distributed document driven database [44] that store information in JSON documents. As MongoDB is a distributed database, we can store all the simulation results in a distributed fashion so that it can be accessed remotely.

Similarly, model results must be communicated to users on demand. Using a database to store the simulation results allows us to separate the simulation execution process from the result processing system. Then, while a process is in charge of running the simulation, a separated process (in the same or in a different computer) could be in charge of fetching the results from the database and sending them to the client on demand.

Cadmium has a logging system that records information about the different simulation events:

- Coupled and atomic model initializations.
- Coupled and atomic time advance: each time an atomic model time advance function is triggered, the simulator logs the model previous time advance and the new time advance. For coupled models, this event is logged when the time advance is triggered in all its submodels and the coupled model new time advance is determined.
- Message routing (IC, EIC and EOC): when messages are sent through a link, the simulator logs the port from where the message is sent, the port that will receive the message and the messages.
- Atomic model state: every time a model transitions, the simulator logs the new model state.
- Atomic model elapsed time: the simulator logs the elapsed time from the last transition on each transition.

- Simulation global time: the simulation starting and ending virtual times are saved.

In order to log these events Cadmium uses a logger class that is parameterized to allow modellers to customize the logs. A logger class is a C++ class that provides the next methods:

- For each event, the logger must provide a formatter function that consumes the event information, the simulation virtual time and the model id as the parameters, and produces a string to log. The string should include the event information.
- The logger must provide a sink object that implements the << operator for strings. This object is where the strings returned by the formatter are printed. For example, if the logger has the *std* :: *cout* as the sink object, the formatted events will be printed in the simulation standard output (the terminal console).

If not custom logger class is given to Cadmium, the default *Cadmium* :: *logger* is used. This logger uses the std::cout as the sink object. Therefore, by default, the simulation results are printed in the terminal console.

The logging-record flow is as next: First, Cadmium obtains the event information (the particular event information specified in the logged events list). Cadmium also obtains the simulation virtual time and the model id. Next, the provided formatter is used to obtain a string from the logged event information, the simulation virtual time and the model id. Finally, the simulator prints the obtained string in the provided sink object.

We have implemented Metric MongoDB Recorder (MeMoRe), a C++/Python module that provides a custom logger to use with Cadmium. This logger formats the simulation events as JSON documents that can be stored in a MongoDB database. MeMoRe also provides a sink object that establishes a connection with a MongoDB database and stores the printed events in the database. Thus, using MeMoRe we are able to record all the Cadmium logged events in a MongoDB database.

The MeMoRe custom logger uses a simulation id passed as a parameter to use as the collection name to separate the different simulations' results in the same database. All events have the simulation virtual global time, model id, and the event name as default attributes. After that, all the remaining information regarding the logged event are set as new attributes. Using the event name and the model id we are able to filter the logs and obtain only those events we are interested in.

There are two options for the sink that records the formatted metrics to the database, the first one is a C++ class that can be passed to the Cadmium simulation as the sink, and the other is a Python module that reads the standard input and saves the documents in the database.

The C++ MeMoRe sink is more efficient because it is integrated into the simulator and the logged events are directly saved in the database. This module initializes a database connection in its constructor and then each time the << operator is used, the sink uses

the connection to store the formatted log into the database.

The Python MeMoRe sink is less efficient because it is a different program that consumes the documents from the standard input, and it saves them into the MongoDB. The simulation must use the standard output as the sink object to print the logged events, and the simulation standard output must be redirected to the Python MeMoRe sink standard input. Using inter-process communication adds some overhead to the recording process, but it is faster to set-up and less complex to install than the C++ module.

6. THE DYNAMIC CADMIUM SIMULATOR

Cadmium is a C++17 header-only DEVS simulator easy to include and to integrate into different projects. Cadmium has been tested on Linux using the GCC and Clang compilers, and it is under the BSD open source license. Cadmium is a pure code library (no graphical interface is provided), which includes C++ classes to create atomic and coupled models, and a runner class that consumes a single coupled model and runs a simulation. Cadmium is composed of three main C++ classes that use the template system:

- **Coordinator**: Coordinates coupled model classes.
- **Simulator**: Simulates atomic model classes
- **Runner**: Runs simulations of a coupled model (the top model).

Instead of receiving the model objects as a constructor parameter, these classes receive the models' types as a template parameter, and they use the model type to construct their own model objects. Cadmium also uses the model type as the model identifier. Consequently, if we have two identical submodels, we cannot use the same model class for both (they cannot be distinguished, as they will have the exact same type). Therefore, we first create two new types that are renames of the original model, and then we use the newly defined types as the submodels.

Having the model type as part of the simulator and coordinators' types, allows Cadmium to create custom structures in compilation time that remove complex calculations at runtime.

On one hand, using a template system is good for running large numbers of simulation scenarios, and to perform multiple static checks at compile time, avoiding some errors at runtime. On the other hand, compiling multiple instances of Coordinator and Simulator template classes (one for each model type) consumes memory and CPU, making the compilation overhead of medium and large models high.

As pointed out in [45], the performance of C++ compilers is not good for programs using templates. If we make strong use of metaprogramming, the compiler cannot compile the program in most computers. In consequence, our main idea behind Cadmium dynamic was to reduce the number of objects defined using templates, and to hide the model types using an abstraction layer (but keeping the same simulation algorithm used in the original Cadmium simulator).

Next, we will explain how the original Cadmium uses the template system, and how we have used abstract types to reduce the number of specializations, solving the compilation time and memory problems. The Simulation and Coordination algorithms have not been changed from the original and the reader can find them in [46].

6.1 The use of the C++ template system in Cadmium

The C++ template system allows developers implementing C++ classes without knowing all the types involved in the class methods and attributes. A template class parameterizes some types, and it gives a name for those types to be referenced. Then, the parameterized types are used in the code as any other type using the assigned name. In order to use the template class, the parameterized types must be instantiated with real types.

Template classes that are instantiated with real types are called specialized classes. C++ does not compile the template classes; instead, it only compiles the specializations. Each time a template class is specialized, the compiler makes a new copy of the entire class and replaces all the type references with the real types to create a new C++ class without template parameters. This makes the compilation process more expensive in terms of memory and time.

Using the template system allows us to use meta-programming techniques to implement checks at compile time. For example, the template class simulator can have two template types called TIME and MODEL to represent the types of the simulation virtual time and the model type. Then, the simulator class can check at compile time if the template type MODEL has a function called *time_advance()* that returns an object of type TIME.

As we have already mentioned, instead of consuming the models as the Simulator and Coordinator constructors' parameters, Cadmium uses the template system to specify the model type. Cadmium implements the Simulator class as a template class that parameterizes the atomic model type. To create a new atomic model, we must first create a C++ class that implements the required methods (internal, external, confluence, time_advance and output). Then, the simulator class is specialized using the atomic model class type. The result of this specialization is a new Simulator class that only simulates the specialized atomic model.

Cadmium implements the Coordinator class as a template class. The Coordinator consumes a list of atomic and coupled model types as the template parameters and it recursively specializes new simulator and coordinator classes for each submodel type passed as a parameter.

There are several optimizations and improvements obtained by passing the model as a template type. First, the model memory usage can be calculated in compilation time avoiding stack overflow problems at runtime. Second, using typed links allows us to define ports with custom message types. Thus, different message types can be sent through different links, avoiding having a single large message type for the entire model. Likewise, static links mapping allows defining efficient data structures to avoid iterating on complex structures dynamically to find the correct link where to send each message at runtime. Finally, using models as types allows implementing static model checking that looks for model inconsistencies at compile time, avoiding simulation crashes due to bugs in the model definition.

The static model checks made by Cadmium include:

- Atomic model static asserts:
  - Model methods: we check whether all the methods of the atomic model specification (internal, external, time advance, output and confluence) are defined, and they have the correct parameters and return types.
  - Ports: we check that the model ports types are correct.
  - Valid model state type: we check whether the model class has an attribute called _state of type model_name::state_type.
- Coupled model static asserts:
  - Link types consistency: we verify if the model does not connect two ports with different message types.
  - Connected ports: we see if the model does not connect invalid ports depending on the model EIC, EOC or IC link structure.
  - Valid coupled and atomic submodels: we check recursively if the submodels are valid by static checking every sub model.

6.2 Cadmium compilation time and memory usage problem using metatemplates and std::tuples

As we have already mentioned, in Cadmium, ports are types derived from *Cadmium::out_port<typename MSG>* and *Cadmium::in_port<typename MSG>* respectively. Each port has a different type depending on the specialization of MSG. In order to define the multiple model ports, Cadmium uses an *std::tuple* [47] because it is a structure that allows us defining different types for each element (each port has a different type). Then a model defines its ports as follows:

Listing 6.1: model port definition example

```
// Defining the input ports specialization with custom message types
struct input_port_0: public Cadmium::in_port<MSG_TYPE_1>{};
...
struct input_port_n: public Cadmium::in_port<MSG_TYPE_N>{};

// Defining the output ports specialization with custom message types
struct output_port_0: public Cadmium::out_port<MSG_TYPE_1>{};
...
struct output_port_n: public Cadmium::out_port<MSG_TYPE_N>{};

class atomic_model {
        // Assigning the ports to the model
        using input_ports = std::tuple<input_port_1, ... , input_port_n>;
        using output_ports = std::tuple<output_port_1, ... , output_port_n>;
        ...
}
```

Dealing with *std::tuple* is easy; elements within the tuple can be directly accessed by their types, and because each port has its own type, we can access them using the port type. Nevertheless, a major problem of *std::tuples* is its compilation memory complexity, which makes almost impossible to compile tuples with a large number of elements.

To understand this problem, we compiled tuples with different sizes, and used the Python Statemodels library [21] to fit 1000 ordinary least squares models [48] of the curve shown in Formula 6.1. This allowed us to determine the best curve that explains the obtained memory compilation complexity.

$$
\begin{array}{|l|}
\hline
size^x \\
\quad \text{with } x \in [2,3]. \\
\quad \text{where } size \text{ is the number of elements of the } std::tuple \\
\quad \text{and x is the variable to modify in order to fit different curves} \\
\hline
\end{array}
\tag{6.1}
$$

Table 6.1 shows the *std::tuple* sizes we have compiled and the RAM usage of each compilation. The maximum size we could compile with 16GB RAM was 400 elements. All element types within *std::tuples* are the same.

After fitting the least squares models for the 1000 different values of $x \in [2,3]$, we got that $x = 2.853$ is the one that produces the curve that best explains the obtained memory usage (in terms of R-square) when compiling a *std::tuple*. The R-square of the model using $x = 2.853$ is 1, which is a perfect R-square.

Results of Figure 6.1 show how the best obtained curve with x (in red) fits the experimental values of Table 6.1. This result shows that the memory usage (in GB) of an *std::tuple* with n elements is $n^{2.853}$ (almost cubic complexity).

Because in Cadmium the models' ports are stored in *std::tuples*, compiling a model with n ports will consume at least $n^{2.853}$GB of RAM. Then, a model with 1000 ports in a single component would be impossible to compile without a computer with less than 175GB of RAM. The previous analysis is a best-case scenario. To compile a model with multiple components, we cannot consider the component with the maximum number of

Std::tuple size	**Used RAM**
100	400MB
150	950MB
200	1.9GB
250	3.2GB
300	5.6GB
350	8.66GB
400	12.4GB

Tab. 6.1: The result of the number of RAM consumed to compile an std::tuple with different sizes.

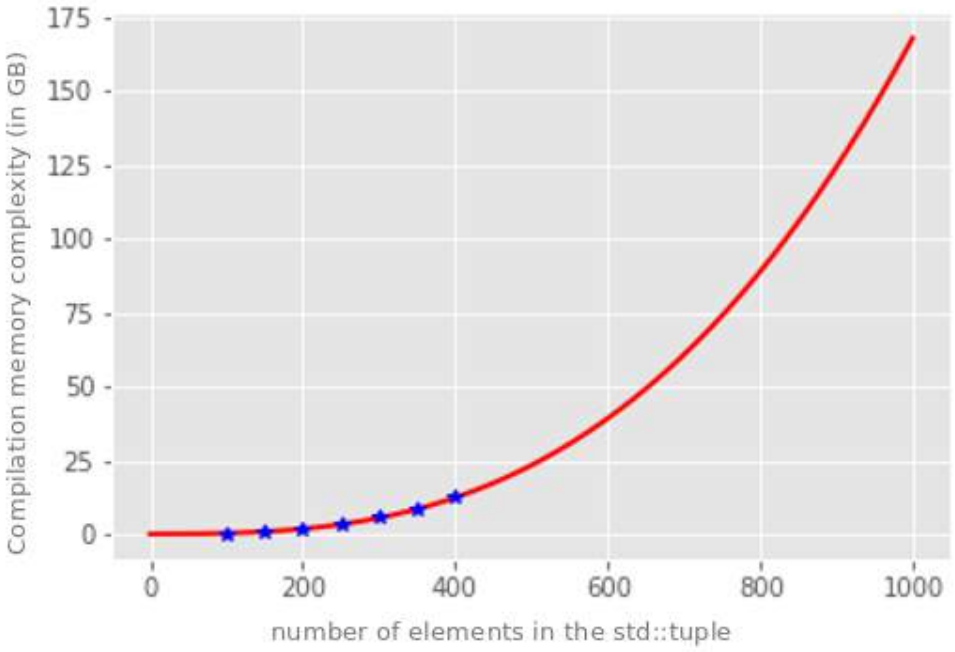


Fig. 6.1: Blue dots are the real data from table 1 plotted and the red curve is the prediction results of the best-fitted curve shown in Figure 24 for std::tuple with up to 1000 elements.

ports but a function of the entire model number of ports.

Tuples are not the only problem when compiling Cadmium models. Because DEVS is hierarchical, the model types are also hierarchical. The type of a coupled model is composed of the type of all its atomic and coupled sub model types. As mentioned earlier, the coordinator class is a template class that takes the coupled model type as a template parameter and walks through all its submodels to recursively specialize all the sub coordinators' and simulators' types to finally define its own type.

The coordinator specialization is an expensive compilation task that has a high memory complexity that makes impossible to compile large models. We conducted some tests using different real models with different sizes to determine the impact of the Cadmium compilation complexity in some real scenarios. The first test included the model of nuclear evacuation emergency plan proposed in [49]. This model simulates the communication channels in an emergency evacuation in nuclear plants. Table 6.2 shows the result using the number of atomic models as the model size.

The second test shows the results for the model introduced in chapter 4 by varying the numbers enzymes in the model (enzymes are atomic models). For this test, we built SBML files where all the compartments have the same number of enzymes. Table 6.3 shows the results of compiling these models with the original Cadmium simulator.

As we can see in table 6.2 and 6.3, the compilation memory complexity is too elevated. Likewise, table 6.2 shows that we need 11.9GB for 1440 atomic models, while table 6.3

Number of atomic models	Compile time RAM usage
1440	11.9GB
2880	22.44GB
4320	32.77GB
5760	43.84GB

Tab. 6.2: Results of compiling the model proposed in [49] with different numbers of atomic models.

Atomic model number	Compilation RAM usage
18	1.64GB
24	2.10GB
30	2.78GB
36	4.21GB
42	>10 GB

Tab. 6.3: Results of compiling the proposed model of this work using SBML files with different numbers of reactions atomic models (from 1 to 5 per compartment).

use more than 10GB for only 42 atomic models. This is because of the hierarchical model type definition of each model. Thus, models can be hard to compile not only because they have a lot of components, but also because of their topology.

6.3 Dynamic adaptation of Cadmium using abstract types

To solve the Cadmium compilation problem, we used a C++ pattern to hide the objects' specific types. For this purpose, we use abstract base classes without template parameters as interfaces, and we define the concrete template classes as derived from the abstract classes. Therefore, we can deal with the objects using their abstract base class without knowing their real type. The C++ code of Listing 6.2 shows an example of this, where *foo* is the abstract base class and *foo_impl<typename NUMBER>* is an implementation of the *foo* interface that has a template type to specify a different type of numbers.

Listing 6.2: Example of using abstract classes to hide the real type of a concrete classes.

```
class foo {
public:
        virtual int get_id()
}

template<typename NUMBER>
class foo_impl: public foo {
private:
        NUMBER id;
public:
        foo_impl()
        int get_id() { return (int)this->id; }
}

function show_id(foo* f) {
        std::cout << f.get_id() << std::endl;
}
```

```
int main() {
        foo* a,b;
        a = &foo_impl<int>(1);
        b = &foo_impl<double>(3);
        show_id(a);
        show_id(b);
}
```

As we can see in Listing 6.2, we can store *foo_impl<int >* and *foo_impl<double>* in the same variable of type *foo* using pointers instead of directly assigning the objects.

The idea is, then, as follows:

- We reduce the multiple specialization instances of the Runner and Coordinator template classes by having a single Runner class for every atomic model and a single Coordinator class for every coupled model. For this purpose, we have implemented an abstract layer to hide the real model types. Then, we have implemented derived classes from the abstract layer. Finally, we can pass the abstract classes to the Simulator and Coordinator, so they do not need to know the real type of the model they are handling.
- We added an id attribute to the atomic and coupled models to stop using the model type as the model identifier. Thanks to this, we can use the exact same model class multiple times.

In order to hide the atomic model type to the Simulator class, we have implemented an atomic model wrapper class derived from the abstract class that consumes the atomic model class type as a template parameter. Because the atomic wrapper knows the atomic model type, it can implement all the static checks that Cadmium implements. The atomic model wrapper also implements the mapping between the atomic model abstract class interface and the real model class interface.

For the coupled models we have just removed the sub model types as template parameters, and we have added the submodels as the coupled model constructor parameters. This decision is not easy because know we cannot make static checks of coupled models as it does the original Cadmium. But we have moved those checks into the Model constructor so they will be validated in run-time before the simulation starts. Also, we have moved the links from the template system to the constructor parameters. Because Cadmium makes a strong use of the *std::tuple* for dealing with links with different message types, instead of removing the *std::tuples* we have also implemented an abstraction layer for the links. Thus, we have an abstract link that is consumed by the coupled model and a derived link that knows the real type of the connected ports and knows how to send messages between them.

Figure 6.2 shows a diagram of the current Cadmium architecture. The T arrows represent that the pointed module has as a template parameter the type of the module that it pointing to it. As we can see in Figure 6.2, in the original Cadmium, the coordinator consumes a coupled model as a template parameter, and the coupled model recursively

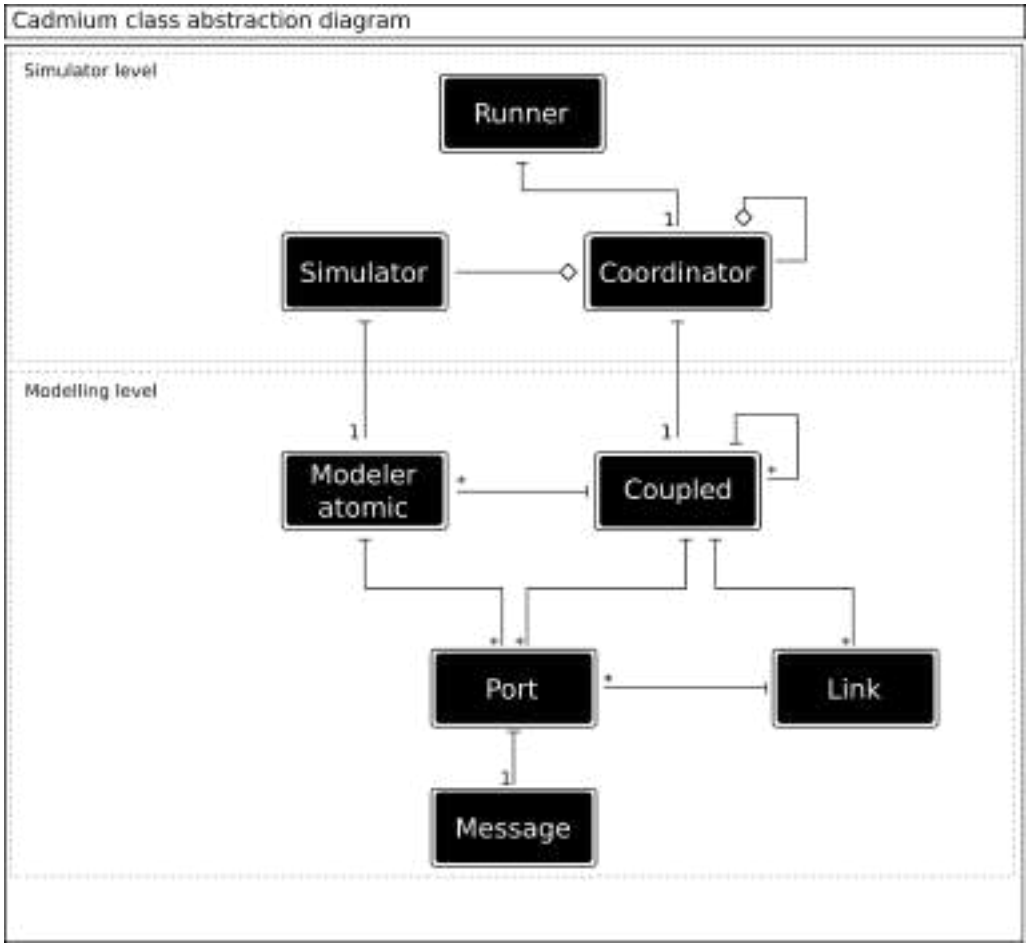


Fig. 6.2: Cadmium abstraction layers.

consumes other coupled and atomic model types as template parameters. Because of this, a coupled model as a complex recursive type structure (as we have already explained). The original Cadmium has only two layers; In the Simulator layer we have the Simulator and Coordinator modules that depends on the type of the model they are Simulating and coordinating, because of this, we have a different Simulator and Coordinator specialization for every atomic and coupled model.

Figure 6.3 shows the proposed abstraction levels diagram of Cadmium dynamic. Abstract classes only define interfaces without implementing any concrete method. In Cadmium dynamic, the coordinator and simulator interact with the abstract level to avoid the need of knowing the concrete model type. The abstract layer uses the atomic wrapper and the link implementation as bridges between them and the concrete model types. In order to hide the concrete model types, the bridge layer is connected to the abstract layer as derived classes (white triangles) and they implement the same abstract class interfaces. Therefore, the Simulator only knows the abstract class, but the concrete classes are the one that will finally implement the required methods.

As shown in Figure 6.3, The atomic and coupled models are derived from the abstract

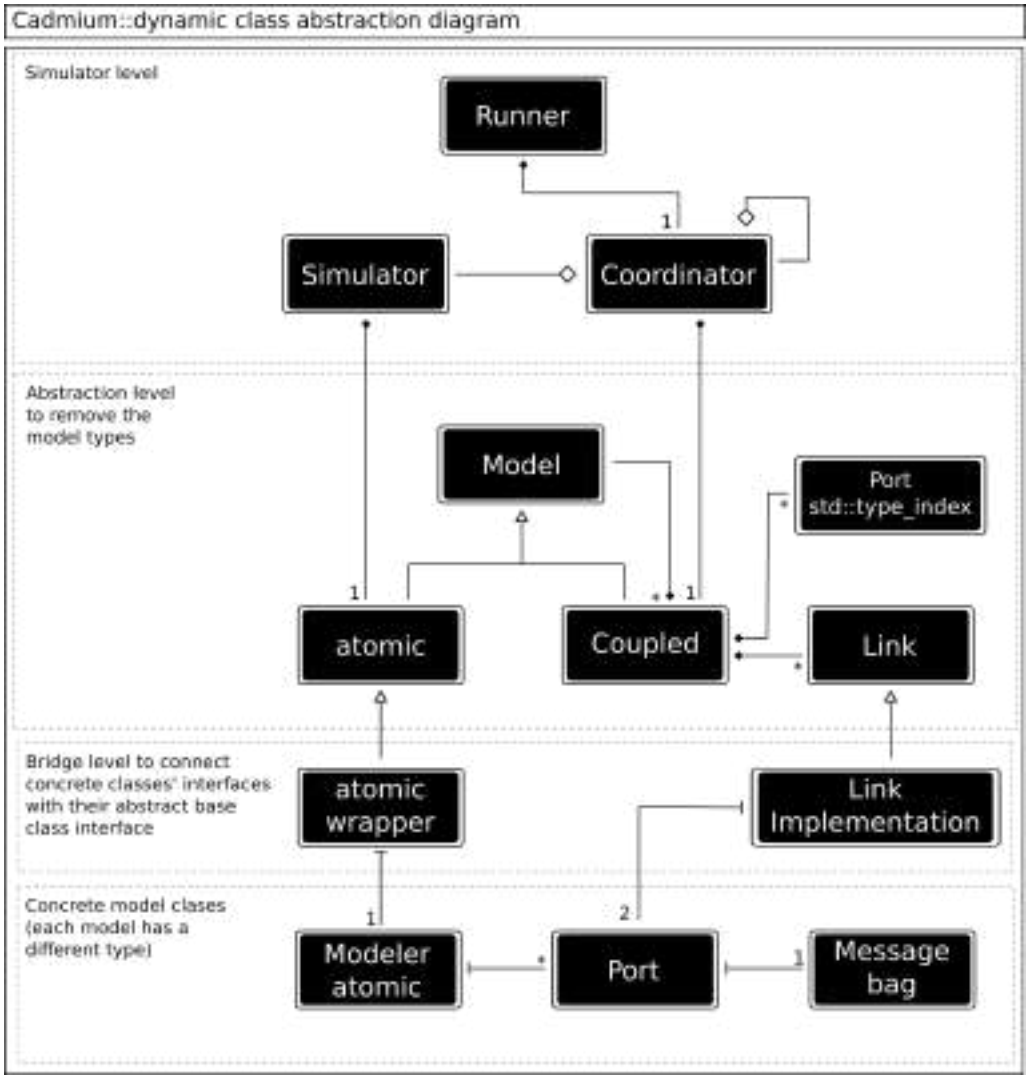


Fig. 6.3: Cadmium::dynamic abstraction layers.

type model, therefore, a coupled model can just consume a vector of abstract models that can either be abstract atomic classes or abstract coupled model classes. Then, for each submodel, the Coordinator tries to cast the submodel as an abstract atomic model to use with a Simulators and if it fails, it cast the model as a coupled model to use with a new Coordinator. Also, the coupled model uses the links models (that is an abstract class) to hide the real type of the link implementation class which relies on the specific ports' types. Finally, the coupled model uses a vector of ports references as the EIC and EOC. Because messages final consumers are always atomic models, the coupled model does not need to know the port type (which depends on the message type). But, to be able to check the links consistency, we use *std::type_index* as the port reference to compare the port type with the link type as we will explain later.

Next, we will explain the atomic wrapper, simulator, coordinator and runner implemented to work with the proposed abstraction mechanism.

6.4 The atomic model wrapper

The atomic model wrapper is a class derived from the real atomic model class and from the abstract atomic class. Its main function is to map the concrete methods of the atomic model class into the methods of the abstract class to connect both levels.

One major problem when implementing this abstraction is that the atomic functions (external, confluence and output) have the message types in their definition. To solve this problem, the abstract interface implements the message bags using the *boost::any* type [50] to allow messages of any type in the same container. *boost::any* allows us to convert any object into a *boost::any* object, the problem is that once the object is converted, we lose the real type, and to come back from the *boost::any* object to the real type we need to explicitly know the real type to cast the object. The atomic wrapper model uses the real port types definition of the model to correctly cast the messages from the *boost::any* object to the correct message type.

To map *boost::any* messages into the correct port message type we use an *std::map* [51] with *std::type_index* [52] as the keys and *boost::any* as the values. This *std::map* holds the messages that were converted into *boost::any* using their port type as the key. The *std::type_index* allows us to get a hashable representation of the port type that can be used as the map key, but a major problem is that we cannot obtain the real type from them. Because of this, for each *std::type_index* key in the *std::map*, we must traverse the *std::tuple* with the model port types, get the *std::type_index* representation of the port type, and compare it with the key to find the correct port. Once we find the correct port, we use the port message type to cast the *boost::any* value into the correct message object. In this way we can go and come back from the Cadmium *std::tuple* port structure to our abstract representation using *std::maps* of *boot::any*.

The obtained atomic model interface is a class that defines the DEVS functions (external transition, internal transition, confluence transition, output and lambda). This abstract class implement a mapping between the interface types and the concrete model types and then, it uses the concrete model methods to obtain the correct results. Atomic models also implement the virtual Model class interface, so they can be treated as abstract models. The Model class interface only has getters for the model ids used by the simulator and coordinators to identify them.

In order to maintain the static model checking of the original Cadmium, the atomic wrapper constructor uses static asserts to validate the model by calling the original Cadmium checking methods. Because these asserts are static, they will be checked in compilation time when instantiating the class type.

6.5 Coupled models

Coupled models are defined in a class that instead of receiving the EIC, EOC, IC, input ports, output ports and submodels as the template parameters, it receives these elements as the constructor parameters. Because all the constructor parameters must have the same type, we use abstract type for the submodels and links (EIC, EOC, and IC). In this coupled

model, we use a *std::vector<std::type_index>* to specify the ports. Then, As we do in the atomic model wrapper, we use *std::maps* to handle the message bags of *boost::any* instead of having *std::tuples*. A difference with the atomic model wrapper, is that for the coupled model we have completely removed the use of *std::tuples*, as shown in Figure 6.3. The submodels' abstract class is a class on top of the atomic and coupled abstract class that allows us to handle submodels without differentiating between atomic and coupled models.

In order to implement the original Cadmium static checks, the coupled model uses the *std::vector* of ports and the abstract links to check whether the ports *std::type_index* of the links correspond with the input and output ports *std::type_index* of the connected models. These are the same checks the original Cadmium does. Because the coupled model takes all the parameters as constructor parameters, no static checks can be made in compilation-time, and the mentioned links checks are made at run-time. Even though the checks are not made in compilation-time as in the original Cadmium, this is not a major problem because they are still done before the simulation starts, preventing crashes during the simulation.

The coupled model class does not need to implement any method because, in DEVS, coupled models are only the model structure. Therefore, the coupled model class only implements the interface requested by the abstract class Model, this is mandatory for both, coupled and atomic models, so they can be passed as a parameter to other coupled models by using the Model interface as we have explained before.

6.6 The dynamic simulator

The main difference between the simulator of Cadmium dynamic and the simulator of the original Cadmium is that in the original Cadmium, the atomic model is a template parameter and the simulator construct the model object, while in the Cadmium dynamic simulator, the atomic model is constructed outside the model and passed as a parameter of the Simulator constructor. For this purpose, the Cadmium dynamic Simulator receives the abstract atomic model. Then, to simulate a concrete model, we first create an atomic wrapper (as we have already explained) and then we pass this wrapper to the simulator as an abstract atomic, hiding the real model type to the Simulator class.

The simulator algorithm remains the same as the original Cadmium algorithm, making sure the simulation run-time performance remains the same, but benchmarks for the original Cadmium are in process and after those benchmarks will be ready, we will be able to compare both implementations.

6.7 The dynamic coordinator

The coordinator class takes the coupled model as a constructor parameter, this model has a vector of submodels of abstract type Model. These submodels are atomic and coupled model pointers derived from the abstract class Model. Then, for each sub-model, the coordinator tries first to cast the model into an atomic model pointer, if the cast works, the abstract model is an atomic model and a simulator is instantiated to handle it. If the atomic model cast fails, the model is a coupled model and a coordinator is recursively

instantiated to handle it.

One of the advantages of the original Cadmium is the use of types for quickly access the coupling links. Because in the Cadmium dynamic, links are dynamic objects. The coordinator uses hashed maps to access them using their model ids. Then, each time a model output function is called, the coordinator uses the model id to obtain all the outcoming links from the model and correctly route the messages to the receiver components. Even though accessing links by their type could be faster, this method is yet more optimal than the commonly used vector of links that must be iterated each time.

The coordination algorithm is the same as the original Cadmium algorithm, therefore the coordination run-time performance should remain the same. We will conduct benchmarks for the original Cadmium, in order to compare both implementations.

6.8 The link, a model component with coordination responsibilities

One of the coordinator responsibilities is to route messages between the model' and submodels' ports using the model links. The problem is that Cadmium dynamic uses dynamic message bags (as we have explained before) that hide the real message bag type. Because of this, we are not able to access the real message bags' structures, and therefore, we cannot insert new messages into their structure. To insert a message into a dynamic message bag, we must first cast the dynamic message bag to its real type to work with them. Because of this, we need to know the dynamic message bag real types to copy messages between them.

As shown in Figure 6.3, message bags' types are known by the Port class, thus, we need to know the Port types to cast the dynamic message bags into their real types. In turn, the Ports' types are known by the Link implementation class but not by the link base class which is an abstract class used to hide the link implementation type (a type using templates) from the coupled model.

Because the coordinator class cannot access the message bags' types, we have moved the simulation responsibility of copying messages ports to the Link implementation class. Then, the coordinator uses the Link abstract interface to request to the Link implementation class to copy messages between its ports' dynamic message bags.

6.9 Results of the camdium::dynamic compilation time and memory use

In order to determine the compilation memory and time complexity of Cadmium Dynamic, we run a few case studies. We tried to run these case studies with the original Cadmium to compare results, but they did not compile.

The main idea of the case studies was to separate the different aspects of a model to see their impact in the compilation memory complexity. We separated the scenarios in two families, those scenarios that add more dynamic objects to the model and those that add more static types to the model.

- Number of Objects:
  - **Atomic models**: we created multiple coupled including from 1 to 1000 identical atomic models.
  - **Coupled models**: We created multiple coupled models from 1 to 1000 empty coupled sub-models.
  - **Links**: We have created multiple coupled models all with 101 atomic sub-models: 100 sub-models have a single output port all with the same type and the remaining sub-model has a single input port. Then, we vary the link number by connecting different numbers of sub-models with the output port to the single sub-model with the input port.
- Number of types:
  - **Port types**: We construct an atomic model with different number of output ports, from 1 to 120. Because each port must have a different type, we are not able to consider scenarios with port types number and ports number separately.
  - **Atomic model types**: identical to the atomic models number scenario but each atomic model has a different type.
  - **Link types**: We also have 101 atomic sub-models, but each of the 100 sub-models with output ports have 100 different ports, and we have always 100 links to connect these models to the single input port model. We then vary change the number of link types by using different ports to connect the 100 links. For example, if all sub-models are connected using their first port, then, the model has 100 equal links. If all sub-models are connected using its first port but the last sub-model that uses its second port, then we also have 100 links, but we got 2 different link types, and so on until all 100 links have different types.

Figures 6.4, 6.5, 6.6, 6.7, 6.8, 6.9 shows the results of the case studies' compilation and memory usage. Time is always in seconds and memory is always in GB.

As we can see in Figure 6.4, Cadmium dynamic was able to compile models with over 10000 atomic in less than 8 seconds and consuming no more than 0.8GB of RAM, but the time and memory complexity grows in linear fashion regarding the number of atomic model objects contained in the TOP model. Figure 6.4.a show a stepped pattern in compilation time complexity, we are not sure what is the cause of this, but we can see that the general time complexity remains linear.

As we can see in Figure 6.5, the number of coupled models also affects the compilation time and memory use linearly, but they are more expensive than compiling a model with multiple atomic models.

As we can see in Figure 6.6.e, the number of links contained in a model has a significant effect in the time complexity. Compiling a model with 10000 links takes almost one hour to compile and the complexity curve is not linear but almost a quadratic curve. Compiling models with more than 10000 links could be a problem in some cases. We think this complexity comes from the Link implementation class, which is a template class containing its ports' types. Then, each link has a unique type that must be compiled. Even though the

compilation time of models with several links is slow, Figure 6.6.f shows the compilation memory complexity remains linear.

As we can see in Figures 6.7, 6.8 and 6.9, varying the number of port types, atomic model types and link types have a linear impact in the compilation time and memory complexity. This is a major improvement regarding the original Cadmium where the model types have a tremendous impact in the compilation time and memory complexity. But because of the problem of *std::tuples* we are not able to compile models with more than 120 ports per component.

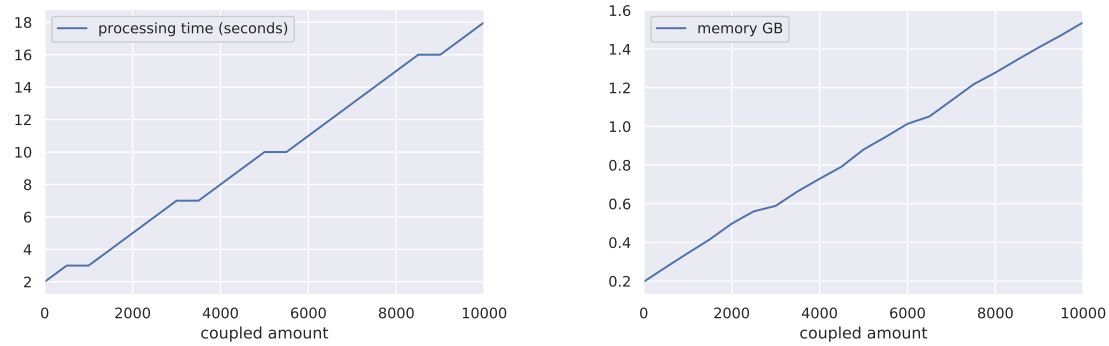
These results show a major improvement in the Cadmium model compilation time; from being unable to compile medium to large models to compile them in a few seconds.

As Cadmium is based on model types, each model component must be explicitly declared without using any flow control system as iteration cycles (for example a loop). This is also a major problem at compilation time, because it is much more effective to parse a four-line loop that declares thousands of models, than parsing thousands of lines (one for each model). In the experiments, we did not use iteration flows to create any part of the model in order to test just the performance improvements of Cadmium dynamic, but we are able to use them. In this thesis, we have used dynamically defined models using for loops to create the models and the compilation memory usage is significantly lower than the presented in the experiments.

Comparing run-time performances of Cadmium with other simulators is in process; but it is outside the scope of this thesis.

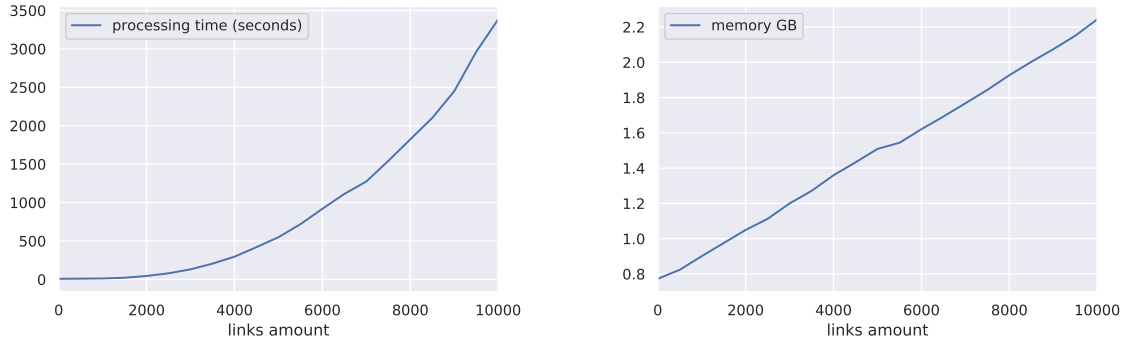
(a) (b)

Fig. 6.4: Results of the compilation time and memory usage of the case study atomic model



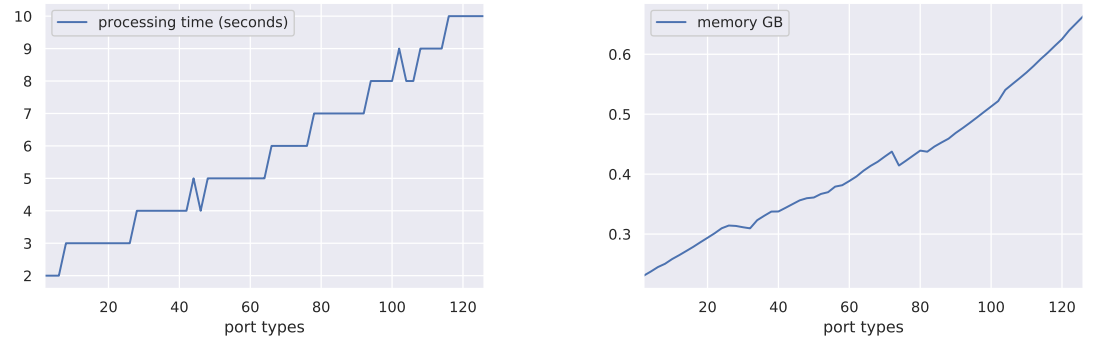
(a) (b)

Fig. 6.5: Results of the compilation time and memory usage of the case study coupled models



(a) (b)

Fig. 6.6: Results of the compilation time and memory usage of the case study links



(a) (b)

Fig. 6.7: Results of the compilation time and memory usage of the case study port types

Fig. 6.8: Results of the compilation time and memory usage of the case study atomic model types

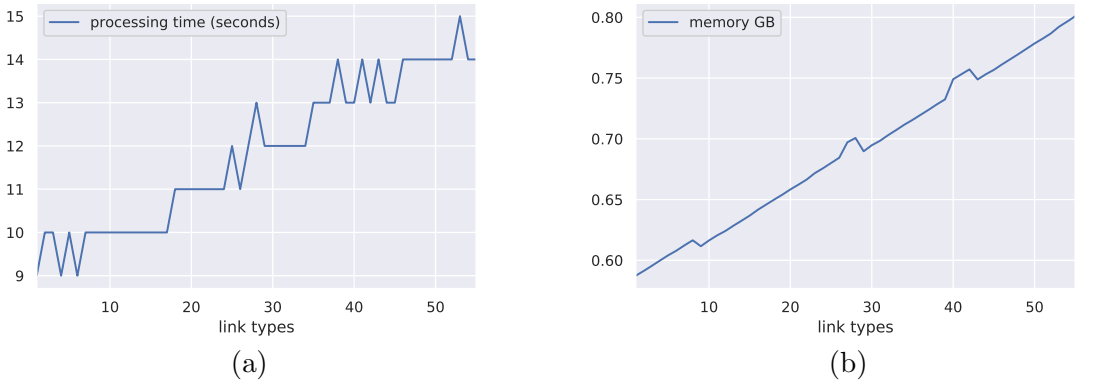


Fig. 6.9: Results of the compilation time and memory usage of the case study link types

7. RESULTS AND MODEL VALIDATION

In this chapter, we present the results of the two scenarios we have generated, compiled and run simulations; The first case scenario is a very simple theoretical model where the expected results are trivial to estimate. The second case scenario is a real structure of the E. Coli bacteria but using some non-real parameters (due to the lack of micro-view information about this bacteria). Likewise, in this chapter, we show the generation, compilation and simulation time of a series of theoretical models where the number of enzymes is the series variable to measure.

7.1 Validation with a theoretical model

In order to test our method, we generated an SBML model of a theoretical biological cell with a single enzyme called b0000 that catalyzes all the cell reactions. In each compartment (Periplasm and cytoplasm) there are multiple enzymes b0000. Likewise, as the proposed model only considers catalyzed reactions, in the extracellular space there are enzymes b0000 catalyzing the reactions occurring outside near the cell. In this model, we have defined the following reactions:

- **A_to_2A_in_cytoplasm**: A reaction that occurs in the Cytoplasm and consumes a single metabolite A_c and produces two metabolites A_c. Because of the proposed stoichiometry, this reaction increases the number of metabolites A in one unit.
- **B_to_2B_in_extracellular_space**: A reaction that occurs in the Extracellular space and consumes a single metabolite B_e and produces two metabolites B_e. Because of the proposed stoichiometry, this reaction increases the number of metabolites B by one unit.
- **C_C_to_2C_3C_in_periplasm_inner**: A reaction that occurs in the periplasm inner-membrane (the membrane connecting the Periplasm with the Cytoplasm). This reaction consumes a single metabolite C_c from the Cytoplasm and a single metabolite C_p from the Periplasm and produces two metabolite C_c in the Cytoplasm and three metabolites C_p in the Periplasm. Thus, it increases by one the number of metabolites C_c in the Cytoplasm and it increases by two units the number of metabolites C_p in the Periplasm.
- **D_D_to_2D_3D_in_periplasm_outer**: A reaction that occurs in the periplasm outer-membrane (the membrane connecting the Periplasm with the Extracellular space). This reaction consumes a single metabolite D_e from the Extracellular space and a single metabolite D_p from the Periplasm and produces two metabolites D_e in the Extracellular space and three metabolites D_p in the Periplasm. Thus, it increases by one the number of metabolites D_e in the Extracellular space and it increases by two the number of metabolites D_p in the Periplasm.
- **E_E_E_to_2E_3E_4E_in_periplasm_trans**: A reaction that occurs in the periplasm trans-membrane (the membrane connecting the Periplasm with the Extracellular

space and the Cytoplasm). This reaction consumes a single metabolite E_e from the Extracellular space, a single metabolite E_c from the cytoplasm and a single metabolite E_p from the Periplasm and produces two metabolites E_e in the Extracellular space, three metabolites E_c in the cytoplasm and four metabolites E_p in the Periplasm. Thus, it increases by one the number of metabolites E_e in the Extracellular space, it increases by two the number of metabolites E_c in the Cytoplasm and it increases by three the number of metabolites E_p in the Periplasm.

- **No_product_in_periplasm**: A reaction that occurs in the Periplasm and consumes a single metabolite F_p and produces nothing. Because of this, the reaction decreases the number of metabolite F in the periplasm by one.

- **No_substrate_from_related_compartment_in_periplasm_outer**: A transport reaction that occurs in the Periplasm outer membrane and transports a metabolite G_e from the Extracellular space to the Periplasm, obtainin a metabolite G_p in the Periplasm.

We have generated a parameter file with 1000 enzymes b0000 in each enzyme set and 600000 metabolites of each type (A to G). Because each reaction uses different metabolites, all the metabolic pathways are composed by a single reaction. We have set the volume of each compartment equal to the estimated volume of the E. Coli bacteria, which is 0.528 cubic micrometers for the Cytoplasm and 0.072 cubic micrometers for the periplasm and extracellular space. We set the Kon and Koff constants equal to 0.8 for all the reactions in order to increase the probability of binding metabolites and decrease the probability of rejecting bound metabolites. Finally, we have set the enzymes reacting, ejecting and the spaces intervals of time in one millisecond, which is completely arbitrary, but do not affects the results as all the reaction are independent of each other.

Figures 7.1 to Figure 7.7 shows the simulation results of the metabolic pathways from the proposed theoretical model until the virtual time 00:00:32:947. In the result we ca see the following:

- All the metabolites increased by one in a reaction have a similar linear curve with a mean final number of 2.500.000 metabolites. These are the metabolites A_c, B_e, C_c, D_e and E_e.

- All the metabolites increased by two in a reaction have a similar linear curve with a mean final number of 4.500.000 metabolites, which is more than the metabolites increased by one. These are the metabolites E_c, D_p and C_p.

- The metabolite E_p, which is increased by three increases more than the metabolites increased by two.

- The metabolite F_p decreases until there are no more metabolites; As shown in figure 28.f, the decreasing curve is not perfectly linear, this is because each time a No_product_in_periplasm reaction occurs, the number of metabolites F_p decreases and therefore, the probability of binding decreases, making the reaction flow rate slower.

- The metabolite G_e decreases at the same rate than the metabolite G_p increases, which is consistent with the transport reaction No_substrate_from_related_compartment_in_periplasm_o In this case, we see the same effect shown in the metabolite G_e where the reaction flow rate is not linear at the end.

On the one hand, the reactions are stochastic processes, because of this, even for those metabolites with the same parameters in their corresponding reactions have some differences in their curves. On the other hand, because we have set a considerable number of enzymes, the stochastic probability reaches values close to the mean and we obtain almost perfectly linear curves without considerable deviations. We think that the stochastic model could be modified to consider this effect and instead of calculating the binding probability individually for each enzyme, we could consider a general formula that calculates the final total number of enzymes that bound metabolites. These would be a major improvement in the simulation time complexity from linear to almost constant.

7.2 Validation with the E. Coli bacteria using the msb201165-sup-0003 SBML model

We have used the SBML model of the E. Coli bacteria presented in the msb201165-sup-0003 file (https://github.com/Laouen/PMGBP-PDEVS/blob/master/msb201165-sup-0003.xml) to generate a real case scenario. Using this file, we have all the information regarding the cell structure, enzymes and reactions involved in the E. Coli metabolic pathways. A major problem we face with this model is that we do not have information about the Kon, Koff constants nor the reaction and reject rates. It is out of this thesis scope obtaining the values of these parameters, and therefore, we cannot compare the results with in-vitro or in-situ results and we only use this case study to show a simulation of a whole-cell running in the platform. Even though we are not presenting results with real parameters, the next stage of this project is to obtain those parameters to run real results to validate the model with in-vitro or in-situ results or with already existent models, as for example, we could

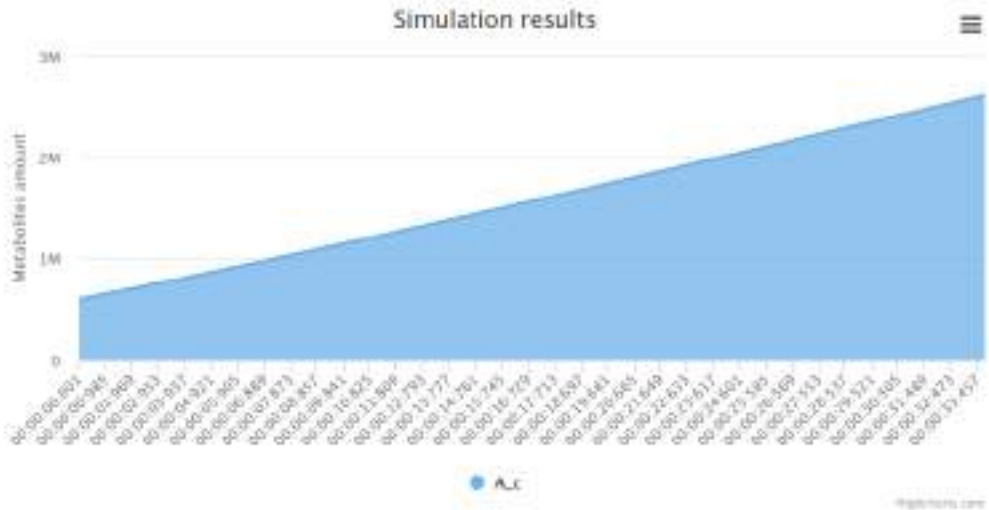


Fig. 7.1: Simulation result of the metabolite A_c in the theoretical model

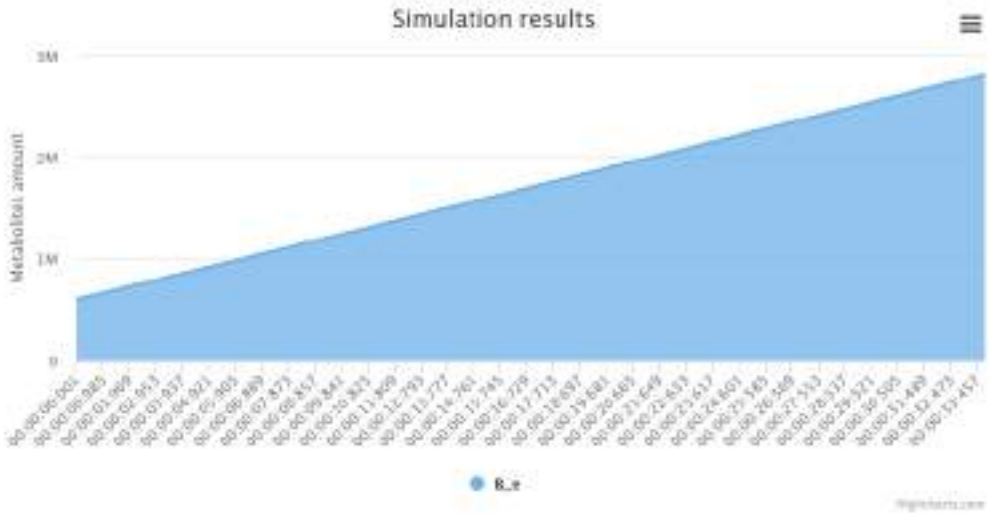


Fig. 7.2: Simulation result of the metabolite B_e in the theoretical model

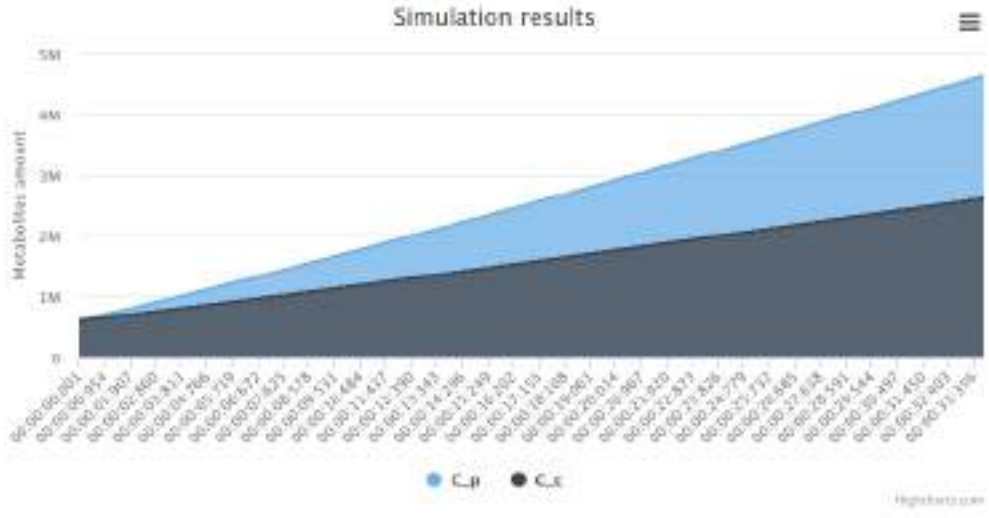


Fig. 7.3: Simulation result of the metabolite C_p and C_c in the theoretical model

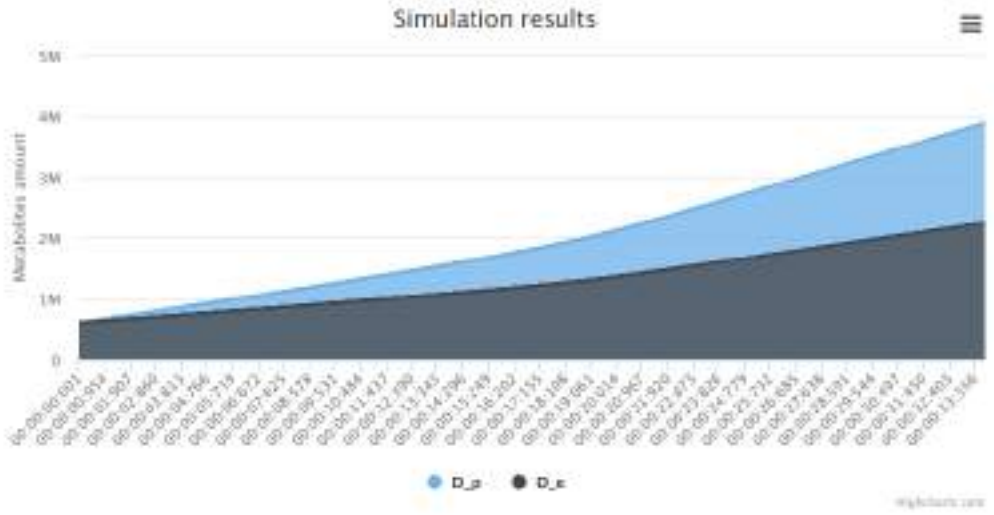


Fig. 7.4: Simulation result of the metabolite D_p and D_e in the theoretical model

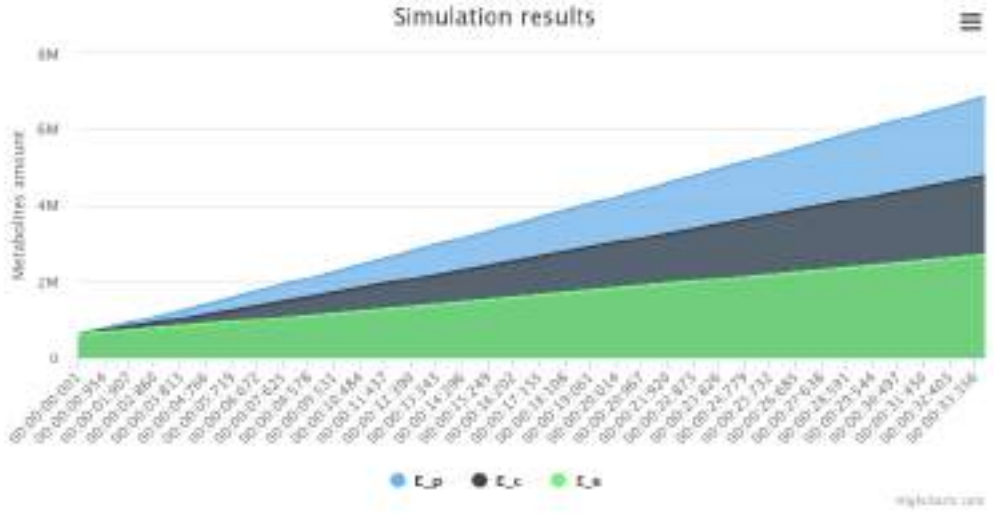


Fig. 7.5: Simulation result of the metabolite E_p, E_c and E_e in the theoretical model

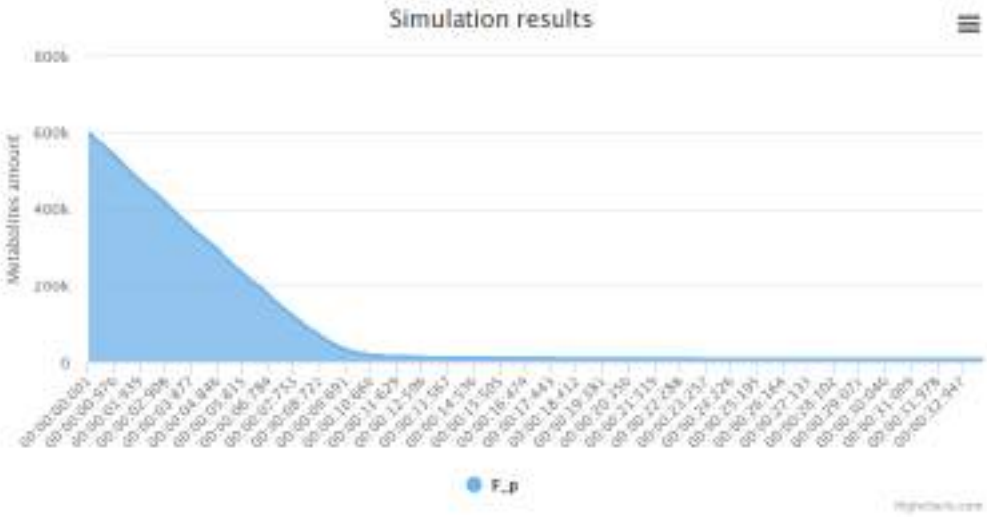


Fig. 7.6: Simulation result of the metabolite F_p in the theoretical model

Fig. 7.7: Simulation result of the metabolite G_p and G_e in the theoretical model

compare the results of our model for the glycolysis cycle with the results presented in [3]. The laboratory of Rafael Najmanovich [1] will work in this next stage as this project was made in collaboration with them.

We have used the following parameters:

- Kon = 0.8 for all the reactions.
- Koff = 0.8 for all the reactions.
- Reaction time = 1 millisecond for all the reactions.
- Reject time = 1 millisecond for all the reactions.
- Space Interval of time = 1 millisecond for all the compartments.
- Initial enzyme amount = 1000 for all the enzymes.
- Initial metabolite amount = 600000 for all the metabolites.

The simulation of the E. Coli bacteria took 50 hours to simulate 6 seconds and 336 milliseconds of simulation virtual time in a Core i7 Asus X751L notebook. During the simulation, 2581 types of reactions were handled by 1577 types of enzymes consuming and producing a total of 1805 types of metabolites. As we have already mentioned this result is just for validation purpose and we did not use real parameters. Therefore, we are not making a complete analysis of the whole metabolic network, instead, we analyze some particular reactions and their metabolites.

Figure 7.8 shows the result of the metabolites handled by the R_FE3DHBZStonex reaction, which is a transport reaction located in the periplasm trans-membrane. The R_FE3DHBZStonex has the following stoichiometry:

$$M\_h\_p + M\_fe3dhbzs\_e \rightarrow M\_h\_c + M\_fe3dhbzs\_p$$

Basically, it transports h from the Periplasm to the Cytoplasm and *fe3dhbzs* from the extracellular space to the periplasm.

As shown in Figure 7.8, the substrate metabolites M_h_p and M_fe3dhbzs_e decrease, but the product metabolites M_h_c and M_fe3dhbzs_p do not always increase. An interesting fact is that the metabolite M_h_c (metabolite h in the cytoplasm) has a fast increasing rate until the virtual time 00:00:01:811, and then it starts decreasing at almost the same rate. Also, the metabolite M_fe3dhbzs_p remains almost constant. In order to understand this phenomenon and considering we have used the same parameters for all the reactions, we studied the total production and consumption rates of these metabolites in the entire bacteria; table 7.1 shows for each metabolite the total production rate considering all the reactions that produce that metabolite and the total consumption rate considering all the reactions that consume that metabolite. The final production rate is the difference

[1] biophys.umontreal.ca/nrg/NRG/Home.html

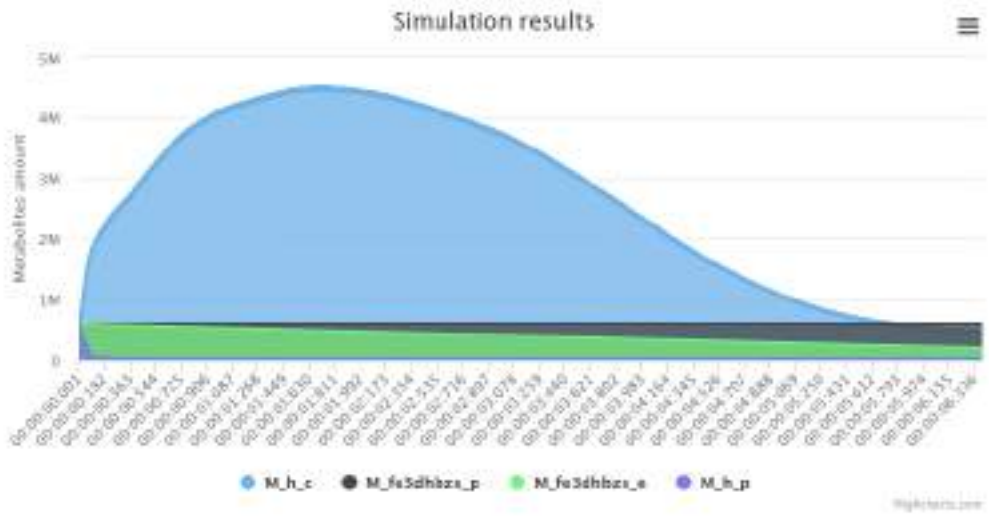


Fig. 7.8: Simulation result of the metabolites involved in the FE3DHBZStonex transport reaction in the E. Coli bacteria.

between the production and the consumption rates. A metabolite that is more consumed than produced tends to decrease its number over the time, while a metabolite that is more produced than consumed tends to increase its number over the time.

The M_h_p has a significant negative final production rate, and as shown in Figure 7.8 it is almost completely consumed in the first milliseconds, while the M_fe3dhbzs_e has a final production rate of −2, which is consistent with the simulation results where it decreases slower than the M_h_p metabolite. The metabolite M_fe3dhbzs_p as a final production rate of 0 and as shown in the simulation result it remains almost constant over the time. The problem is that the M_h_c has a final production rate of 567 which is consistent with the first 1.811 simulation virtual seconds, but it is contradictory with the remaining simulation virtual time. To understand this, we have gone one step further:

On the one hand, if we take the substrate of all the reactions producing the metabolite M_h_c and we analyze the mean of the final production rate of these metabolites, we have that the rate is approximately -0.037; this means the metabolites needed to produce M_h_c tends in general to decrease.

On the other hand, if we take the substrate of all the reactions consuming the metabolite M_h_c and we analyze the mean of their final production rate, we have that the rate is approximately 0.7889; this means the metabolites needed to consume M_h_c tends in general to increase.

Because of this, when the simulation start, we have enough metabolites, and the final production rate of the metabolite M_h_c is the one shown in Table 7.1, but while the time goes on, the remaining metabolites needed to produce M_h_c decrease and the metabolites needed to consume M_h_c increase. As a result, the metabolite M_h_c decreases even though its final production rate is high.

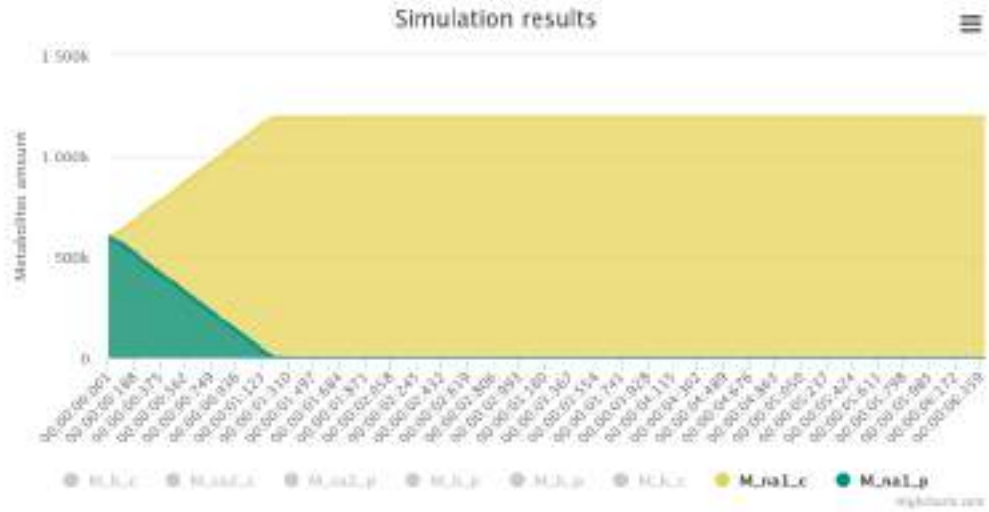


Fig. 7.9: Simulation result of the metabolites involved in the NAt3pp transport reaction in the E. Coli bacteria.

Metabolite	Total production	Total consumption	Final production rate
M_fe3dhbzs_e	0	2	-2
M_fe3dhbzs_p	1	1	0
M_h_p	130	232	-102
M_h_c	920	353	567
M_na1_c	12	4	8
M_na1_p	5	12	-7
M_malthx_p	2	1	1
M_malthx_e	0	2	-2

Tab. 7.1: Metabolite productions and consumption rates in the entire E.coli bacteria.

Figure 7.9 and Figure 7.10 shows the simulation result for the transport reactions NAt3pp (in the periplasm inner-membrane) and MALTHXtexi (in the periplasm outer-membrane) respectively. In both cases, the results are consistent with the final production rate of the corresponding metabolites in Table 7.1. In the case of the NAt3pp reactions, the involved metabolites (M_na1_c and M_na1_p) have bigger final production rates than the metabolites of the MALTHXtexi reactions. Because of this, the M_na1_p metabolite shown in Figure 7.9 is consumed faster than the M_malthx metabolite shown in Figure 7.10.

Figure 7.11 shows the top 5 metabolites with the highest final production rates; these metabolites are more produced than consumed, and except for the M_h_c metabolite that we have already analyzed before, they tend to constantly increase over the simulation virtual time. The final production rates are shown in Table 7.2.

Figure 7.12 shows the top 5 metabolites with the lowest final production rates; these metabolites are more consumed than produced. As shown in Figure 7.12, the metabolites M_atp_c, M_h_p and M_h2o_p have a fast decrease rate and then they remain constant almost in 0. The metabolite M_nadph_c also has a fast decrease rate, but after that, it starts

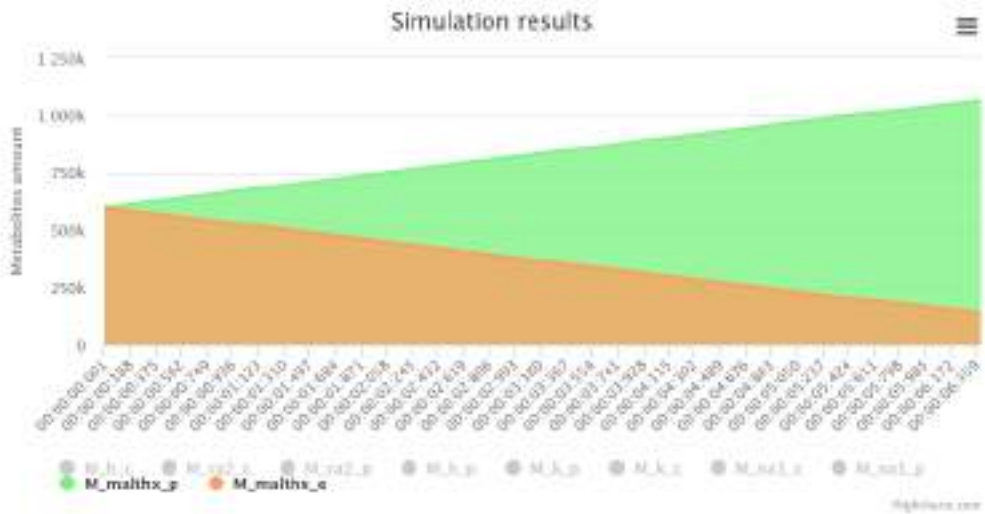


Fig. 7.10: Simulation result of the metabolites involved in the MALTHXtexi transport reaction in the E. Coli bacteria.

to slowly increase. This can be caused by a similar phenomenon to the M_h_c metabolite. The final production rates are shown in Table 7.3.

Figure 7.13 shows the top 5 metabolites with the highest total production rates and lowest consumption rats while their final production rates remain in zero; these metabolites are equally consumed than produced. As shown in Figure 7.13 they present an almost constant rate where they tend to go up and down. The final production rates are shown in Table 7.4.

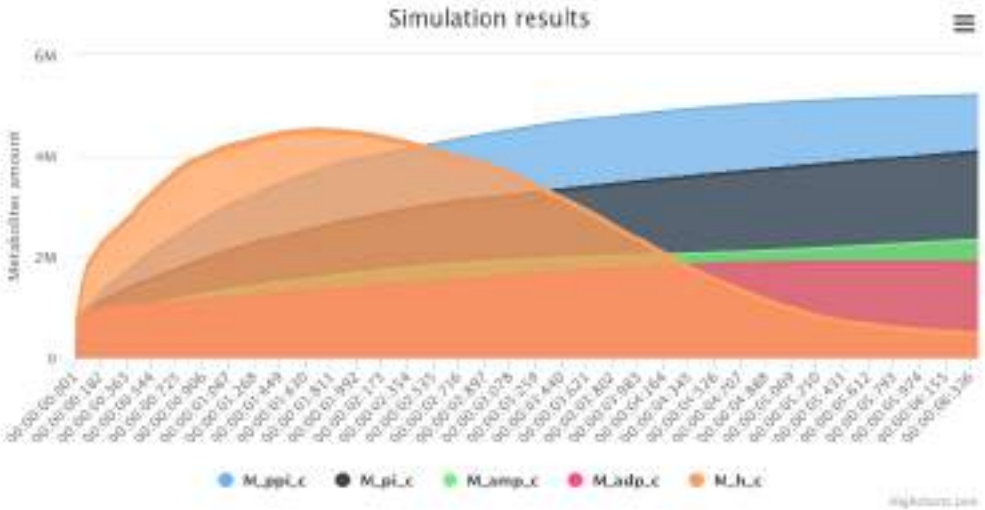


Fig. 7.11: Simulation result of the top 5 metabolites with the highest final production rates.

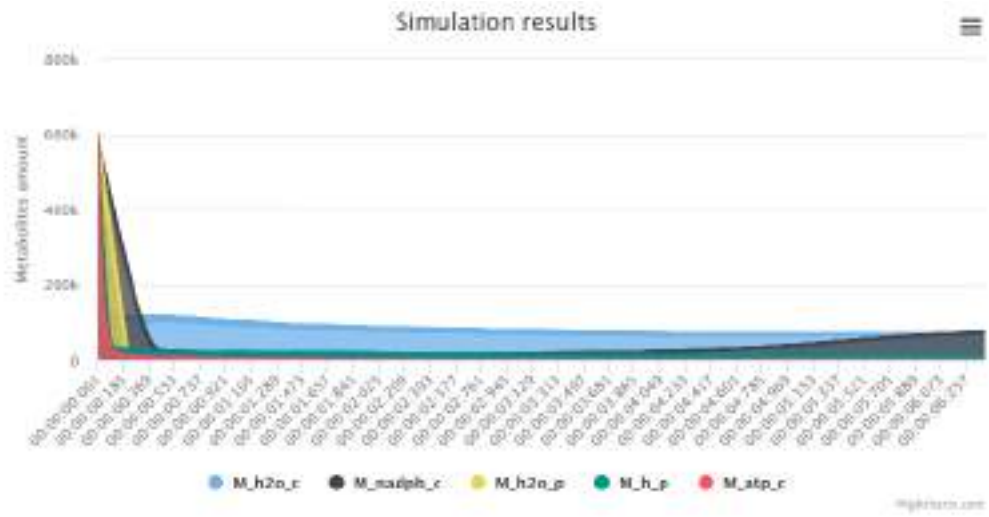


Fig. 7.12: Simulation result of the top 5 metabolites with the lower final production rates.

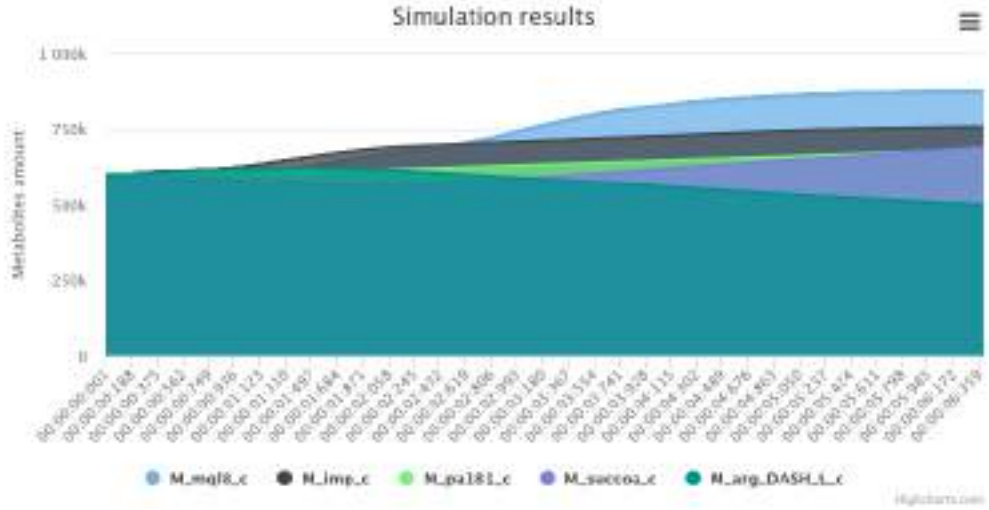


Fig. 7.13: Simulation result of top 5 metabolites with highest total production and consumption rates and with 0.0 final production rates.

Metabolite	Total production	Total consumption	Final production rate
M_h_c	920	353	567
M_adp_c	275	6	269
M_pi_c	276	39	237
M_ppi_c	151	3	148
M_amp_c	91	5	86

Tab. 7.2: Metabolite total productions and consumption rates in the entire E.coli bacteria of the top 5 most produced metabolites.

Metabolite	Total production	Total consumption	Final production rate
M_atp_c	4	358	-354
M_h2o_c	132	474	-342
M_h2o_p	15	163	-148
M_h_p	130	232	-102
M_nadph_c	21	78	-57

Tab. 7.3: Metabolite total productions and consumption rates in the entire E.coli bacteria of the top 5 most consumed metabolites.

Metabolite	Total production	Total consumption	Final production rate
M_mql8_c	13	13	0
M_arg_DASH_L_c	4	4	0
M_imp_c	4	4	0
M_succoa_c	4	4	0
M_pa181_c	3	3	0

Tab. 7.4: Metabolite total productions and consumption rates in the entire E.coli bacteria of the top 5 most total production rate having a final rate of 0.0.

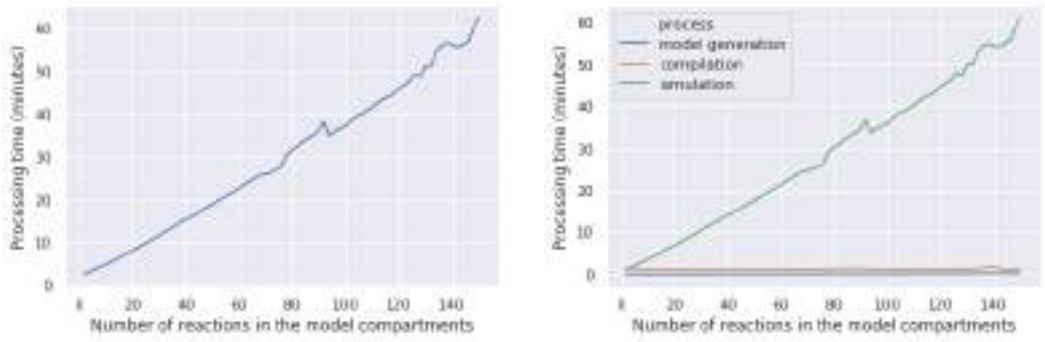
7.3 Performance results

In 5 we have shown results of the compilation time improvements made by our adaptation of the Cadmium simulator; In this section, we present a measurement of the entire process of model generation, compilation and simulation using Cadmium::dynamic. For this purpose, we have generated models with one enzyme per compartment up to 1000 enzymes per compartment and we have measured the entire process time for each model. In order to measure the simulation time, we measure the time it takes to simulate one second of the simulation virtual time.

All the models use the following parameters:

- Kon = 0.8 for all the reactions.
- Koff = 0.8 for all the reactions.
- Reaction time = 1 millisecond for all the reactions.
- Rejecting time = 1 millisecond for all the reactions.
- Space Interval of time = 1 millisecond for all the compartments.
- Initial enzyme amount = 1000 for all the enzymes.
- Initial metabolite amount = 600000 for all the metabolites.

Figure 7.14 shows the processing time (in minutes) of generating, compiling and executing one minute of simulation virtual time of the previously described model varying the number of reactions. This experiment was executed in a Core i7 Asus X751L notebook,



(a) (b)

Fig. 7.14: processing time of compiling identical models where the number of enzymes vary.

and we expect better times when the platform is deployed in well-prepared servers.

As we have already mentioned, in the proposed model for this measurement, each reaction is catalyzed by a different enzyme, thus, the models have as many enzymes per compartments as reactions per compartment. The results shown in Figure 7.14.a show a linear curve where models with more enzymes increase the total processing time, but as we can see in the Figure 7.14.b, most of the complexity belongs to the simulation process, while the model generation and compilation processes remain constant. This result is consistent with our expectations, because as we have pointed out in section 5.2, the model generator uses an iteration cycle (a for loop) to create the enzyme sets, therefore, models with different number of enzymes use a constant number of code lines to generate those enzymes, and the model generation and compilation processes remain constant. In the case of the simulation process, the iteration cycle is executed and produces as many iterations as enzymes it generates, and thus, the simulation has a linear complexity to create all the enzyme atomic models.

8. CONCLUSION AND FUTURE WORK

In this thesis, we presented a web platform for M&S of biological processes using SBML files to upload the models from a remote computer to the server. We have shown that applying computational concepts for M&S of biological processes offers multiple advantages regarding the model integration and reusability, which as far as we know, it not yet very popular in the field. Likewise, this is the first approach for micro and multi-view flexible M&S as a distributed cloud service, which could to improve collaborations and automation of those aspects that can be automated.

We have also proposed a model of a general biological cell structure to use as a model integration framework, a stochastic micro-view model of biological cell metabolic pathways and the concept of multi-state models.

Although we have created different modules to be used in this thesis, they are independent modules stored in individual projects that can be used in other research. For example, the MeMoRe can be used to format and record Cadmium simulation results in a MongoDB database regardless of the model being simulated. Some of these modules as the NDTime and the Cadmium::dynamic have already been used in Cristina Ruiz Ph.D. thesis [49].

The web platform implemented in this thesis has been tested in a Core i7 Asus-X751L notebook and it has been tested with small and large models as shown in section 7, and we expect to have a match better performance in a distributed and specialized server. A problem of the current web platform is that large models produce an enormous amount of data results that must be sent through the internet and handled by the Highchart library, resulting in a slowdown of the user interface response time that could be improved in future work. Likewise, the web platform user interface is currently in beta, and it can be improved in future versions to ensure a better user experience. For this purpose, we need first the platform to be used by multiple researchers to have feedback to work with.

The stochastic model has been validated by the Najmanovich research group[1], but we could not simulate the E. coli model using real parameters because of the lack of some micro-view information as the Kon and Koff constants of some reactions. In the next stage of this work, the real parameters should be obtained in order to run simulations that can be validated with in-situ or in-vitro results.

In the E.coli simulation results 7 we have shown that running micro-view models of a whole-cell metabolic network can be achieved with today computational power, but it remains a slow task that can be improved if we are able to parallelize not only the different processes involved (as we did) but the simulation itself. According to [53] parallelizing the simulation of DEVS models is viable. In future work, this technique could be applied to the Cadmium::dynamic simulator to improve the simulation process of biological cells

[1] biophys.umontreal.ca/nrg/NRG/Home.html

where the number of atomic models is elevated.

Finally, the stochastic metabolic pathway model proposed in this thesis calculates for each enzyme the probability to bind metabolites. As we have already mentioned several times, we do not care about the enzyme's identity. Thus, this formula could be replaced by a single-step formula that calculates the final number of metabolites that bind metabolite. This improvement would be a major speedup for the simulation process while it remains as a micro-view model.

BIBLIOGRAPHY

[1] S. Klamt and J. Stelling, "Combinatorial complexity of pathway analysis in metabolic networks," *Molecular biology reports*, vol. 29, no. 1-2, pp. 233–236, 2002.

[2] A. M. Feist, C. S. Henry, J. L. Reed, M. Krummenacker, A. R. Joyce, P. D. Karp, L. J. Broadbelt, V. Hatzimanikatis, and B. Ø. Palsson, "A genome-scale metabolic reconstruction for escherichia coli k-12 mg1655 that accounts for 1260 orfs and thermodynamic information," *Molecular systems biology*, vol. 3, no. 1, p. 121, 2007.

[3] G. Wainer and R. Djafarzadeh, "Devs modelling and simulation of the cellular metabolism by mitochondria," *Molecular Simulation*, vol. 36, no. 12, pp. 907–928, 2010.

[4] A. C. H. Chow, "Parallel devs: A parallel, hierarchical, modular modeling formalism and its distributed simulator," *TRANSACTIONS of the Society for Computer Simulation*, vol. 13, no. 2, pp. 55–68, 1996.

[5] B. P. Zeigler and S. Vahie, "Devs formalism and methodology: unity of conception/-diversity of application," in *Proceedings of the 25th conference on Winter simulation*, pp. 573–579, ACM, 1993.

[6] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, *et al.*, "The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.

[7] Y. Matsuoka, S. Ghosh, N. Kikuchi, and H. Kitano, "Payao: a community platform for sbml pathway model curation," *Bioinformatics*, vol. 26, no. 10, pp. 1381–1383, 2010.

[8] N. Le Novere, B. Bornstein, A. Broicher, M. Courtot, M. Donizelli, H. Dharuri, L. Li, H. Sauro, M. Schilstra, B. Shapiro, *et al.*, "Biomodels database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems," *Nucleic acids research*, vol. 34, no. suppl_1, pp. D689–D691, 2006.

[9] H. Schmidt and M. Jirstrand, "Systems biology toolbox for matlab: a computational platform for research in systems biology," *Bioinformatics*, vol. 22, no. 4, pp. 514–515, 2005.

[10] B. E. Shapiro, M. Hucka, A. Finney, and J. Doyle, "Mathsbml: a package for manipulating sbml-based biological models," *Bioinformatics*, vol. 20, no. 16, pp. 2829–2831, 2004.

[11] Z. Zi and E. Klipp, "Sbml-pet: a systems biology markup language-based parameter estimation tool," *Bioinformatics*, vol. 22, no. 21, pp. 2704–2705, 2006.

[12] D. B. Kell, "Systems biology, metabolic modelling and metabolomics in drug discovery and development," *Drug discovery today*, vol. 11, no. 23-24, pp. 1085–1092, 2006.

[13] "Django." http://www.djangoproject.com.

[14] G. A. Wainer and N. Giambiasi, "Application of the cell-devs paradigm for cell spaces modelling and simulation," *Simulation*, vol. 76, no. 1, pp. 22–39, 2001.

[15] H.-G. HOLZHÜTTER, G. JACOBASCH, and A. BISDORFF, "Mathematical modelling of metabolic pathways affected by an enzyme deficiency: A mathematical model of glycolysis in normal and pyruvate-kinase-deficient red blood cells," *European journal of biochemistry*, vol. 149, no. 1, pp. 101–111, 1985.

[16] A. M. Uhrmacher, R. Ewald, M. John, C. Maus, M. Jeschke, and S. Biermann, "Combining micro and macro-modeling in devs for computational biology," in *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, pp. 871–880, IEEE Press, 2007.

[17] A. M. Uhrmacher, D. Degenring, and B. Zeigler, "Discrete event multi-level models for systems biology," in *Transactions on computational systems biology I*, pp. 66–89, Springer, 2005.

[18] S. Biermann, A. Uhrmacher, and H. Schumann, "Supporting multi-level models in systems biology by visual methods," in *Proceedings of European Multi-Simulation Conference*, Citeseer, 2004.

[19] X.-m. Zhao, R.-s. Wang, L. Chen, and K. Aihara, "Automatic modeling of signal pathways from protein-protein interaction networks," in *Proceedings Of The 6th Asia-Pacific Bioinformatics Conference*, pp. 287–296, World Scientific, 2008.

[20] A. Belič, D. Pompon, K. Monostory, D. Kelly, S. Kelly, and D. Rozman, "An algorithm for rapid computational construction of metabolic networks: a cholesterol biosynthesis example," *Computers in biology and medicine*, vol. 43, no. 5, pp. 471–480, 2013.

[21] S. Seabold and J. Perktold, "Statsmodels: Econometric and statistical modeling with python," in *Proceedings of the 9th Python in Science Conference*, vol. 57, p. 61, SciPy society Austin, 2010.

[22] A. Bauer-Mehren, L. I. Furlong, and F. Sanz, "Pathway databases and tools for their exploitation: benefits, current limitations and challenges," *Molecular systems biology*, vol. 5, no. 1, p. 290, 2009.

[23] Z. Wang, "Cell biology simulation using devs combined with sbml," *Master Of Science, Department of Electrical and Computer Engineering, The University of Arizona*, 2009.

[24] B. J. Bornstein, S. M. Keating, A. Jouraku, and M. Hucka, "Libsbml: an api library for sbml," *Bioinformatics*, vol. 24, no. 6, pp. 880–881, 2008.

[25] T.-S. Jung, K.-R. Kim, S.-H. Jung, T.-K. Kim, M.-S. Ahn, J.-H. Lee, and W.-S. Cho, "Spdbs: an sbml-based biochemical pathway database system," in *International Conference on Intelligent Computing*, pp. 543–550, Springer, 2006.

[26] J. R. Karr, J. C. Sanghvi, D. N. Macklin, M. V. Gutschow, J. M. Jacobs, B. Bolival Jr, N. Assad-Garcia, J. I. Glass, and M. W. Covert, "A whole-cell computational model predicts phenotype from genotype," *Cell*, vol. 150, no. 2, pp. 389–401, 2012.

[27] 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*, vol. 11, no. 3, pp. 290–300, 2010.

[28] V. S. Ayyadurai and C. F. Dewey, "Cytosolve: a scalable computational method for dynamic integration of multiple molecular pathway models," *Cellular and molecular bioengineering*, vol. 4, no. 1, pp. 28–45, 2011.

[29] W. B. Copeland, B. A. Bartley, D. Chandran, M. Galdzicki, K. H. Kim, S. C. Sleight, C. D. Maranas, and H. M. Sauro, "Computational tools for metabolic engineering," *Metabolic engineering*, vol. 14, no. 3, pp. 270–280, 2012.

[30] A. M. Uhrmacher, D. Degenring, J. Lemcke, and M. Krahmer, "Towards reusing model components in systems biology," in *International Conference on Computational Methods in Systems Biology*, pp. 192–206, Springer, 2004.

[31] H. M. Sauro, D. Harel, M. Kwiatkowska, C. A. Shaffer, A. M. Uhrmacher, M. Hucka, P. Mendes, L. Strömback, and J. J. Tyson, "Challenges for modeling and simulation methods in systems biology," in *Proceedings of the 38th conference on Winter simulation*, pp. 1720–1730, Winter Simulation Conference, 2006.

[32] C. Maus, M. John, M. Röhl, and A. M. Uhrmacher, "Hierarchical modeling for computational biology," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pp. 81–124, Springer, 2008.

[33] C. Sanford, M. L. Yip, C. White, and J. Parkinson, "Cell++—simulating biochemical pathways," *Bioinformatics*, vol. 22, no. 23, pp. 2918–2925, 2006.

[34] A. T. Bittig and A. M. Uhrmacher, "Spatial modeling in cell biology at multiple levels," in *Simulation Conference (WSC), Proceedings of the 2010 Winter*, pp. 608–619, IEEE, 2010.

[35] Y. Harzallah, V. Michel, Q. Liu, and G. Wainer, "Distributed simulation and web map mash-up for forest fire spread," in *Services-Part I, 2008. IEEE Congress on*, pp. 176–183, IEEE, 2008.

[36] M. Rohl and A. M. Uhrmacher, "Flexible integration of xml into modeling and simulation systems," in *Simulation Conference, 2005 Proceedings of the Winter*, pp. 8–pp, IEEE, 2005.

[37] X. Feng, Y. Xu, Y. Chen, and Y. J. Tang, "Microbesflux: a web platform for drafting metabolic models from the kegg database," *BMC systems biology*, vol. 6, no. 1, p. 94, 2012.

[38] "Django applications." https://docs.djangoproject.com/en/2.0/ref/applications/.

[39] "Django url dispatcher." https://docs.djangoproject.com/en/2.0/topics/http/urls/.

[40] M. Hucka, A. Finney, and N. Le Novère, "Systems biology markup language (sbml) level 2: structures and facilities for model definitions," 2006.

[41] "Python advanced string formatting." https://www.python.org/dev/peps/pep-3101/.

[42] "Createjs." https://createjs.com/.

[43] S. Wang and G. Wainer, "A mashup architecture with modeling and simulation as a service," in *International Conference on Web Information Systems Engineering*, pp. 247–261, Springer, 2015.

[44] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.

[45] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Pearson Education, 2004.

[46] "Cadmium p-devs simulator." https://github.com/SimulationEverywhere/cadmium.

[47] "std::tuple." https://fr.cppreference.com/w/cpp/utility/tuple.

[48] "statsmodels.regression.linear_model.ols." https://www.statsmodels.org/dev/generated/statsmodels.regression.linear_model.OLS.html.

[49] C. Ruíz Martín *et al.*, "A framework to study the resilience of organizations: a case study of a nuclear emergency plan," 2018.

[50] "Boost.any." https://www.boost.org/doc/libs/1_61_0/doc/html/any.html.

[51] "std::map." https://fr.cppreference.com/w/cpp/container/map.

[52] "std::type_index." http://en.cppreference.com/w/cpp/types/type_index.

[53] S. Jafer, Q. Liu, and G. Wainer, "Synchronization methods in parallel and distributed discrete-event simulation," *Simulation Modelling Practice and Theory*, vol. 30, pp. 54–73, 2013.