ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

THE UNIVERSITY *of* EDINBURGH

Master's Thesis

Modular Simulations and Timing Issues in Whole-Cell Models

Dominik Bucher
dobucher@ee.ethz.ch

Supervisors
Prof. Vincent Danos
Prof. Heinz Koeppl

Informatics Life-Sciences Institute
University of Edinburgh

Automatic Control Laboratory
ETH Zürich

August 31, 2013

"I could exceedingly plainly perceive it to be all perforated and porous, much like a Honey-comb, but that the pores of it were not regular [...] these pores, or cells, [...] were indeed the first microscopical pores I ever saw, and perhaps, that were ever seen, for I had not met with any Writer or Person, that had made any mention of them before this."

Robert Hooke, *on First Discovery of Cells*

UNIVERSITY OF EDINBURGH, ETH ZURICH

Abstract

Informatics Life-Sciences Institute, Automatic Control Laboratory
School of Informatics,
Departement of Information Technology and Electrical Engineering

Master of Science

Modular Simulations and Timing Issues in Whole-Cell Models

by Dominik Bucher

Many simulation models, especially in biology, could benefit from integrating different smaller modules into a bigger system. This is partly because the complexity of the overall system is immense, so that a single monolithic block would be difficult to describe and reason about, and also because single modules might use different modeling approaches, like ordinary differential equations, boolean networks, flux balance analysis and more. Of special interest are so-called whole cell models, which try to describe and simulate everything that happens withing a biological cell. This thesis presents an analysis of existing whole cell models and a generalized framework to make integration of various modular simulations easy and fast. The work originated from the paper “A Whole-Cell Computational Model Predicts Phenotype from Genotype” by J. Karr et al. [1]. The principles of distributing variables and resources among processes are formalized, analyzed and refined. A reference implementation was developed as part of this work.

Acknowledgements

I would like to thank all people involved in making this thesis possible. Most of all my supervising professor Vincent Danos for constantly motivating me, discussing every possible aspect, integrating me into his research group, bringing the results of the thesis to people outside of the team and for allowing me to write the thesis at the University of Edinburgh.

Also, the whole research group has always been around with a helping hand and lots of feedback - namely Ricardo Honorato, Ilias Garnier, John Wilson-Kanamori, Milana Filatenkova, Guoli Yang, Andrea Weisse and Sandrine Regiec. A special thanks also goes to Hristiana Pashkuleva for studying my system in greater detail and using parts of it for the iGEM competition.

Thanks to Peter Swain, Ramon Grima, Matthew Scott and Elco Bakker from the synthetic biology department, for hosting me in the beginning and for discussing applications and feasibility of my ideas.

Thanks to Jordan Cronin and Shailendra Singh for reading through my thesis and pointing out mathematical and grammatical errors. A special thanks also to my family who always assisted me, also with all the administrative things still running in Switzerland, and for providing funding for my stay in Edinburgh.

Last but not least, I'd like to thank professor Heinz Koeppl for setting me up with Edinburgh and Vincent Danos in special, for supervising me from the ETH side and for making the whole thesis possible!

Contents

List of Figures

List of Tables

Chapter 1

An Introduction to Whole Cell Models

1.1 An Informal Definition of a Whole Cell Model

The biological cell is a building block of life and is present as the basic structural and functional unit in all known living organisms. It was first discovered in the 17th century by Robert Hooke and has been subject to extensive studies ever since. Even after 350 years of research a lot remains to be unveiled and discovered.

Most cells are of a size of between one and hundred micrometers [2], and thus only visible by use of a microscope or other experimental devices. This small size means a human has about 100 trillion cells [3]. It was already discovered very early that organisms are built from an aggregation of cells which perform all vital functions and are able to transfer all information necessary to do so to the next generation of cells, thus allowing life to be passed on [4].

Cells are split in two categories: eukaryotic and prokaryotic cells. Whilst eukariotic cells can be found in all higher forms of life - plants, animals, fungi and so on, prokaryotic cells are either bacteria or archaea. The primary difference is the cells' complexity - eukariotic cells are of much higher complexity. The most significant contrast is the apparent lack of compartmentalization in prokaryotic cells. Eukaryotic cells contain membrane-bound organelles, such as the nucleus, while prokaryotic cells do not. The nucleus is a compartment circumvented by a membrane and holds the cellular DNA. In prokaryotic cells the DNA, just as all other cell parts, is floating around freely in the cytoplasm. As prokaryotic cells are simpler, they are currently the primary research units when talking about whole cell modeling.

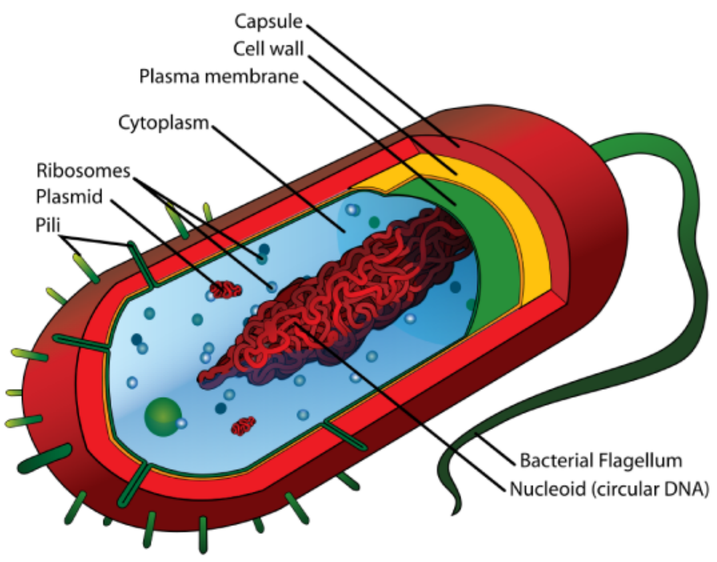

FIGURE 1.1: A prokaryotic cell - A bacterium or an archaeon

Whole cell models try to simulate every single thing that happens within a biological cell. This means all known processes and reactions on all known resources within a cell. Exemplary processes of a cell are translation, transcription, replication, decay, metabolism but also many more. Metabolites, proteins and DNA are examples of biological resources. All these basic elements have to be kept track of and made available to the different processes. Such a simulation obviously gets enormously complex, for which reason various approximations have to be made.

So why do we want to model a system of such huge complexity? Will it be possible to deduce any valuable information from a whole cell model? If we need to know all processes and resources beforehand, what use is the combination of it all?

A whole cell model tries to provide a means for inspecting everything that happens within a cell, how different processes interact and how the system behaves under various initial conditions. Much of its application lies in seeing the interplay of the different processes. Biological models are often created under some assumptions about the host they are taking place in - often neglecting direct interaction with it. However, this interaction can have non-negligible effects [5] and should thus be taken into account whenever possible. As more and more information and knowledge is available for cell processes, hosts can be modeled with greater accuracy, making it possible to integrate many processes into a bigger host system.

It doesn't stop with analyzing interplay and studying effects of different processes on each other. A very important application is the *in silico* experimentation with cells

[6]. The initial conditions of a cell can be changed, cell processes can be modified and changes can be applied to cells during a simulation without any need for biological experimentation. Sample experiments are gene knock outs, insertion of bio-synthetic components or change of cell surroundings. Results can be if the cell is still fit for survival, which processes change in what ways or behaviors that the cell starts to show. Of course, as whole cell modeling is a very recent trend in research, simulations have to be accompanied by biological experiments, but already now they are able to point out interesting things that can then be studied in more detail with *in vitro* experiments [1, 5].

As whole cell models are of great complexity, several approximations and simplifications have to be made. A first and important approximation is the *modularization* of the model. As research has been done on many of the basic processes, but not on the interplay of all of them, the idea is to keep processes as modules which interact in certain ways on a shared state. Modularization of models has been done for a long time and is discussed for example in [7] and [8]. A formalism often used is the *discrete event system* (DEVS), which involves modularization of event-based systems, but can also be used to simulate continuous systems via some extensions and quantization methods. A popular software that allows modularized modeling is Mathworks' Simulink [9]. It allows to specify models in a hierarchical manner and connect the different parts via wires.

In general, modularization has various benefits:

- Modules can be modeled using different modeling techniques. There is often a preferable way of describing a certain process, either by its nature itself or by the amount of research that has been done on it. Chemical reactions can be described using mass kinetics and differential equations, whilst the overall metabolic network of the cell is best described using flux balance analysis. Especially for processes that involve a limited number of reactants stochastic modeling is favorable.
- As every process in itself should be consistent and has most likely been tested in isolation by some researcher, the process can be unit tested within the whole cell simulation as well. This allows to make some statement about correctness of the simulation.
- Modules can be translated more or less directly from literature. This allows the whole cell modeler to integrate different research results in a relatively short amount of time, as no time has to be spent on translating a model into another structure.

Of course modularization also comes with several downsides:

- The different processes have to be fitted into the whole simulation. This is usually tricky as the research on different processes doesn't necessarily agree on all parameters and aspects.
- The modularization also introduces complexity in the simulation engine. This will be described later in this thesis in great detail.

Further, a lot of biological processes are still unknown or poorly described, with data often only available for a few species. In order to be complete, a model has to make assumptions about such processes and their parameters. This can possibly have huge effects which are difficult to study.

Various approximations are not only made from a biological, but also from a computational and mathematical point of view. A general topic of interest is the concurrent access to a shared state by many processes [10]. This computer science issue has been studied in great detail for the synchronous execution of programs. However, the case of biological modeling lets us make assumptions that can greatly simplify the parallel simulation. As an example, the model of Jonathan Karr [1] lets the processes act sequentially, on a unified time scale. But as often in the field of numerical computation, a lot of other solutions exist, several of them allowing the processes to run in parallel. The study of those will be a major point within this thesis. Important aspects covered are the overall simulation performance, the model description language and the integration of the processes. From the computational point of view, the most important factor to modularize a system is that the individual computation time of the processes should be high in comparison to the time needed for the intertwining logic. Separation and parallelism can only shine in such a setting.

A classification of whole cell models in a traditional way isn't possible, as whole cell models often will encapsulate many different modeling techniques. The best classification goes along the line of a huge model that is too complicated to describe as a monolithic block and is instead split into modules. Here, several points need to be denoted:

- This category doesn't only include whole cell models, but all models of great complexity and a lot of unknown factors.
- The term whole cell model is sometimes also used in smaller and less complicated models. Though not complying with the above definition, they try to make general statements about cell behavior as a complete cell, and not single parts thereof. Such models will also be discussed for their use as a host model, where more and more processes can be plugged in.

With this surrounding set, the thesis talks about the following topics: chapter two will give a detailed description and review of Jonathan Karr's whole cell model [1]. As this model is the first real whole cell model it has stirred up the scientific community quite a lot. Some of its assumptions, strengths and weaknesses will be dissected within this thesis. Chapter three will introduce a more formal description of modular simulations, especially with respect to whole cell modeling, where the underlying biology allows for some simplifications in the mathematical structures. Chapter four then discusses interesting properties of the formalism introduced in chapter three. Chapter five and six provide a concrete implementation and experiments run using different algorithms and systems. Chapter seven concludes the thesis and raises some questions that need to be tackled in the future - the field of whole cell modeling is just emerging!

Chapter 2

A Detailed Description of the Karr Model

2.1 A Peek at the Karr Model

The model developed by Jonathan Karr in [1] is the first model that attempts to simulate a whole biological cell, in this case a single cell of a *Mycoplasma Genitalium*. The cell is simulated from a "just-split" state until the cell splits again, describing exactly one cell cycle. In order to do so, the model molds data from over 900 research papers into a single huge system. The model is realized in object-oriented Matlab [11].

As the dynamics of the whole system as a monolithic block are too complicated to reason about and also to integrate, the system is split into 28 processes which are unit tested and verified independently. The processes are modeled using diverse mathematics and fundamentally different experimental measurements. The complete system state is stored in 16 states which provide a namespacing architecture to group single state variables. They also define methods of calculation for various dependent cell properties, e.g. the total cell mass. Processes and states synchronize every 1 second, i.e. processes run independently for one second before returning their results into the main simulation. Processes and states are displayed in figure 2.1.

The model provides insights into the interaction of the different processes and thus into many previously unobserved cellular behaviors. It also allows to change the cell state at any point in time to see how the cell reacts. Changes can range from simple adaptions of the environment to the introduction of new genes or processes. Scientific results mentioned in the original paper include protein-DNA association rates, relationships between the duration of DNA replication initiation and replication, and more. The model also

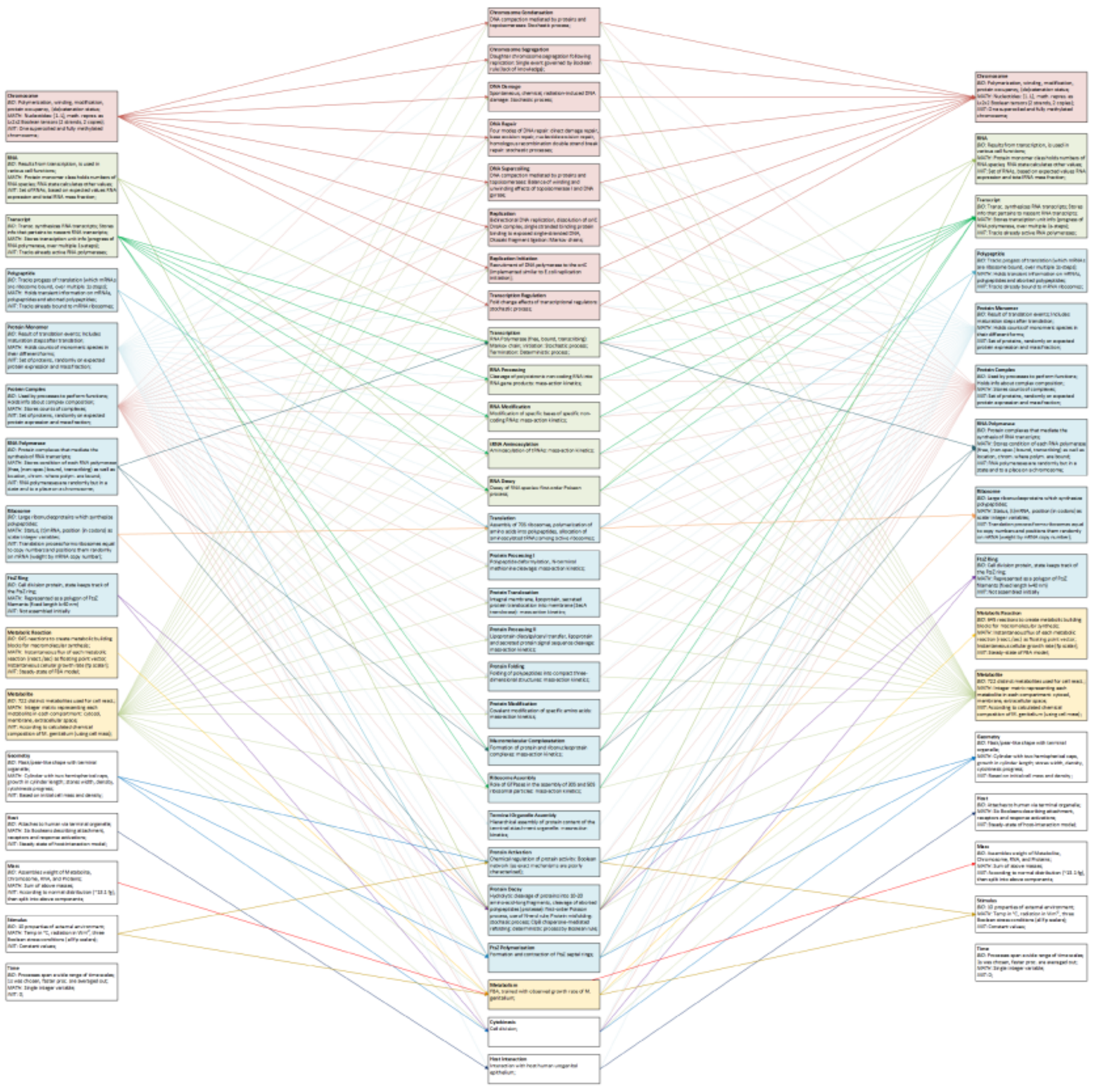

FIGURE 2.1: Model structure of the Karr whole cell model

pointed out previously undetected kinetic parameters and biological functions. Those can then be studied in greater detail doing real biological experiments.

2.2 The Simulation

The simulation is controlled by the simulator object that contains the state vector and calls the different processes. The state vector is split into 16 states mostly for grouping (or namespacing) purposes, but can formally be seen as one entity, except for the metabolite state which will be discussed shortly. States are grouped by biological categories like proteins, ribosomes, metabolites etc. They additionally allow so-called dependent variables which are calculated from state variables. This is done by letting them define arbitrary methods that can be called by processes or the simulator. The 28 processes are specified by defining various methods which are then called by the simulator object.

The simulator object is also responsible for loading the parameter set and initial variable assignments, splitting resources into shares for the different processes and to reintegrate all state changes made by processes. Initial values and parameters come from the knowledge base, a database tailored for storage of whole cell model data. The knowledge base is described more in a later section and extensively in the corresponding paper [12].

The simulation object also stores the complete simulation from the beginning in a state matrix which can be analyzed or stored to disk.

For a thorough description of the simulation system see the supplementary documents of the original paper [1]. The documents are updated on a regular basis and should thus describe the most recent state of the model.

2.2.1 Loading Initial Parameters and Values

In order to run a simulation, initial parameters and values have to be loaded by the simulator. There are various ways to do so, the easiest being to set up the initial state by loading a previously stored state that was initially prepared for this purpose by Karr et al. Changing single parameters and values from this initial setup is easy and can be done via a web interface [13], which generates an xml file with parameters to overwrite. Alternatively, the values of any variable in the simulation can be changed directly in the Matlab environment.

The extended version of loading the initial parameters is to run all the start up scripts, which load parameters from the knowledge base, run the fitting algorithm and thus set up the simulation.

2.2.2 States

States provide a mechanism for grouping and namespacing of the simulation state vector. The grouping is done by logical and biological associations. Additionally, states allow to define functions to calculate various dependent cell properties. States store their variables in vectors, which all together form the complete state vector (the Karr model never speaks of a complete state vector, but this simplification works well and will be used further later on). Variables are also grouped by their dependency on others. States further can provide constants to be read from the knowledge base.

Group	Name	Description
DNA	Chromosome	Represents the polymerization, winding, modification, protein occupancy, and (de)catenation status of the chromosomes.
RNA	Transcript	Stores all of the information that pertains to nascent RNA transcripts and aids in the time evolution of Transcription.
	RNA	Stores the quantities of RNA in different immature/mature and functional/non-functional forms.
Protein	Polypeptide	Holds information about which mRNAs are ribosome bound, as well as ribosomal progress of translating a transcript across timesteps.
	Protein Monomer	Holds counts and attributes of the monomeric species in the system.
	Protein Complex	Holds fixed parameters including the complex subunit composition, molecular weights, amino acid composition, half-lives, localization and minimum expression of each complex.
	RNA Polymerase	Holds precise positions on the chromosomes where polymerases are bound.
	Ribosome	Represents the status of all ribosomes, species ribosomes are bound to and translating and the position of each ribosome.
	FtsZ Ring	Describes the polygonal FtsZ ring built for cytokinesis.
Metabolite	Metabolic Reactions	Stores the instantaneous flux of each metabolic reaction in reactions per second.
	Metabolite	Holds the copy number of each metabolite in each of 3 compartments.
Other	Geometry	Stores the physical shape of the cell including its width, length, volume, and surface area.
	Host	Represents the instantaneous configuration of the human host.
	Mass	Calculates the total cell mass from the other states.
	Stimulus	Represents the status of 10 properties of the external environment.
	Time	Stores the time elapsed in the simulation.

Table 2.1: States used in the Karr model

Table 2.1 summarizes the 16 states used in the Karr simulation. The group isn't used within the simulation and is only there to describe the model in a structured way. However, one could imagine a generalized namespacing scheme where variables can be accessed by *groupname.statename.variablename* or even longer chains.

2.2.3 Processes

Processes implement parts of the model in different ways, e.g. using Markov chains, ODEs or stochastic systems. They have to specify several methods to be integrated into the simulation system. The *global requirements* function is used to determine the initial cell state and total flux of reactions over the complete cell cycle. Together, all the global requirements functions describe the processes' overall contribution to the cell function.

The *current requirements* function tells the simulator the instantaneous flux of metabolites (small molecules used as building blocks in cells) and is used to split the total amount of metabolites proportionally for the different processes. The current requirements in the simulation are usually defined assuming an infinite metabolite supply (resembling Hill type reactions, see also appendix A), thus assuming reactions in a saturated region and only determined by enzyme rates or similar.

The most important function for a process to specify is the *evolve* function which performs the changes on the state. Within the evolve function a process can use any simulation technique to calculate a change that happens on the simulation within the simulation step size Δt $(= 1s)$. It is important to denote some specialties about the evolve function:

- As processes run sequential in the Karr model, any change on the system is valid. There are no concurrency problems and each process can access the whole state vector during evolve. However:
- Metabolites are split. This means that a process will get a fraction of the total metabolites to work upon. This should be considered when simulating mass equations for example, as the real copy number of a metabolite is not available.
- Processes can also specify so-called side effects. Those are of additive nature and are reintegrated by the simulator after the process finishes its evolve. They provide a decoupling of states and processes, but are not extensively used at the time of writing this thesis. This is mainly due to the fact that processes can perform the same changes also directly on the states. A result of this thesis is a greatly enhanced concept of side effects, which is later used as the primary way of integrating processes into the big system.

Tables 2.2 and 2.3 summarize the 28 processes used in the Karr simulation. Again, grouping is for explanation purposes and is not used within the simulator.

Group	Name	Description
DNA	Condensation	Implements “clamping of DNA”, DNA supercoiling, macromolecular crowding and charge neutralization.
	Segregation	Models migration of the chromosomes to opposing sides of the cell before division.
	Damage	Simulates spontaneous DNA modification due to base loss, deamination and influence by exotic agents.
	Repair	Describes machinery to detect and repair damaged DNA.
	Supercoiling	Forces the DNA to be at a certain level of helicity.
	Replication	Produces a complete chromosome for each daughter cell.
	Replication Initiation	Determines when the replication starts.
	Transcriptional Regulation	Regulates the synthesis rates of RNA by modulating the affinity of RNA polymerase for promoters.
RNA	Transcription	Models the first step in the synthesis of functional gene products where RNA polymerase and enzymes translate transcription units into RNA.
	Processing	Models operonic RNA cleavage into individual RNA gene products.
	Modification	Simulates tRNA and rRNA modification.
	Aminoacylation	Simulates the conjugation of amino acids to the tRNAs and the aminoacylation of the tmRNA which delivers the amino acid alanine to stalled ribosomes.
	Decay	Describes decay of all species of RNA.
Protein	Translation	Describes the production of amino acid polymers using ribosomes, enzymes, tRNA and metabolites.
	Processing I	First step after protein translasion for modifying the created amino acid polymers. Models N-terminal formylmethionine deformylation and N-terminal methionine cleavage.
	Translocation	Moves the amino acids to other compartments (second step in post-translational processing).
	Processing II	Models the third step, lipoprotein diacylglyceryl adduction and lipoprotein and secreted protein signal peptide cleavage.
	Folding	Folds the proteins into energetically favorable three-dimensional structures.

TABLE 2.2: Processes used in the Karr model

Group	Name	Description
Protein	Modification	Models protein covalent modification including phosphorylation, lipoyl transfer, and α-glutamate ligation.
	Complexation	Models the formation of macromolecular complexes.
	Ribosome Assembly	Describes the enzymecatalyzed formation of 30S and 50S ribosomal particles.
	Terminal Organelle Assembly	Simulates the assembly of the protein content of the terminal organelle.
	Activation	Activates proteins based on concentration of small molecules, DNA, RNA, other proteins, temperature and pH.
	Decay	Describes dilution of proteins, macromolecular complexes, signal sequences and polypeptides as well as misfolding and refolding of protein monomers.
	FtsZ Polymerization	Assembles the FtsZ monomers into long polymers.
Metabolism	Metabolism	Models the import of extracellular nutrients and their conversion into macromolecule building blocks.
Other	Cytokinesis	Simulates the pinching of the cell membrane until separation and forming of two daughter cells.
	Host Interaction	Models several aspects of interaction with the human host and their influences on the cell.

Table 2.3: Continuation of processes used in the Karr model

2.2.4 Interaction Between States and Processes

A central aspect of the Karr model simulator is the way interactions between the state and different processes are handled. Figure 2.2 shows the basic execution of the model. As is visible the simulation doesn't run in parallel but is instead strictly sequential. What happens is that the state vector is modified by one process after another. Only the metabolites are treated in a special way, they are split between processes at the beginning of a simulation round.

As this introduces quite some unfairness between processes, they are randomly permuted every simulation round, i.e. a process can come at any position between 1 and 28. As an average simulation runs for about 29000 simulation steps (in this case that equals 29000 model seconds, which is about 8 hours), each process' execution positions will be uniformly distributed over the 28 possible ones. However, it gets more involved with

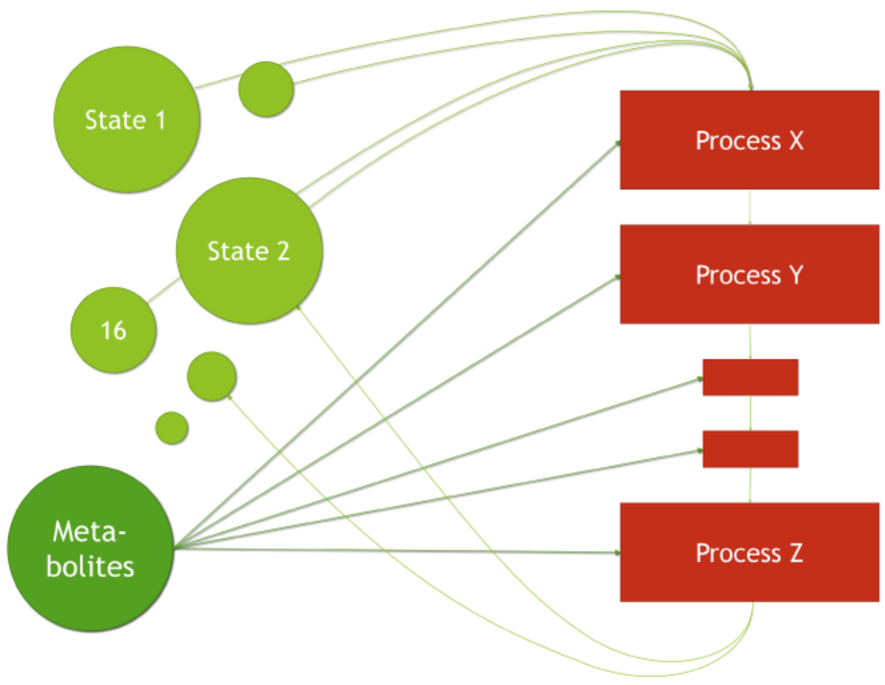

FIGURE 2.2: Simulator structure of the Karr whole cell model

metabolites, as those are usually used in higher frequency reactions and a process in the front of the execution order could use up all of them.

Thus the splitting of certain resources, namely metabolites, plays an important role in the Karr model. Processes have to specify a requirements function that tells the simulator how much of a given metabolite a process would like to consume in a simulation round. An informal argument why metabolites need to be split is made as follows: as metabolites are used in higher-frequency reactions a process could use up all of a given metabolite within a single simulation round. Now one can argue that this should induce a smaller simulation time step. However, the system works like a system of differential equations with different time steps: the fast reactions reach some equilibrium state within Δt and are then reintegrated into the slower system. This works well and brings a lot of advantages also because processes can be simulated independently.

Apart from metabolites, where processes only get a share, they can access and act upon the complete simulation state as desired. Conflicts between the biologically simultaneously running processes are resolved solely by them being executed sequentially. This might give the first process to be simulated an advantage over the others, but as the execution is random in each simulation step a certain degree of fairness is achieved. This is also heavily influenced by the fact that processes (apart from their metabolic reactions) often run slower than the Δt of 1 second. This and the fact that most processes are quite decoupled (they only share a few variables) makes this model feasible.

Processes can additionally introduce so-called side effects on the simulation. Side effects are of completely additive nature and provide some idea of complete decoupling and parallelism of processes. We shall see extended variants of side-effect integration later in the thesis.

2.2.5 A Comparison with Numerical Methods

The Karr system is comparable to a system solver that uses an Euler method without time step adaption. An Euler system gives agreeable results for stable systems and small enough Δt's (for a discussion of various numerical methods and their implementations see [14]). There is one huge difference which will also be emphasized throughout this thesis: it's an Euler method, where the different processes can modify the same variables (one could also split a conventional Euler system, but then the splitting would be according to the various differential equations). Furthermore there is a difference in the methods as usually only differential equations are considered when talking about the Euler method. However, in general the comparison holds and what's left is deciding if the 1 second time step is small enough to keep the error within some bounds. The analysis isn't trivial though, because of the different modeling techniques involved, and also because the processes are modifying variables at the same time. Chapter 4 gives some more detail on this.

2.3 The Knowledge Base

The knowledge base, as described in [12] provides a unified access for whole cell simulations. At the moment of writing this thesis it is only available for *Mycoplasma genitalium*, but the purpose is to provide a platform that can host data for many cells. With the Matlab code available on Github, compiled versions of it also for download, the idea is that at one point people select the material they need, assemble it together with their own models and simulate the generated model.

The knowledge base contains data about compartments, chromosomes, transcription units, genes, chromosome features, metabolites, proteins, reactions, transcriptional regulations, pathways, stimuli and all quantitative parameters needed for the simulation. In addition, though this is somewhat model-specific and not really organism-specific, the knowledge base contains information about the states and processes. This information isn't about the real content of the corresponding Matlab file, but just a comment on what the process / state does and which parameters and reactions it uses. Further modifications of the platform will most likely include source code as well, and custom plug and play of processes and states.

It is possible to generate Matlab objects from the knowledge base via some classes in the framework. However, it often is easier to take a standard object and modify it using the online modification form [13]. This form produces an Xml file that specifies the parameters to be changed and is applied before the simulation starts. It is especially valuable when running many simulations in parallel, as parameter modifications get easy when using this technique.

The knowledge base was originally written in php and has now been rewritten by Jonathan Karr using the Python [15] framework Django [16]. For this reason, there is a lot of redundant code on the web, also for setting up and communicating with the underlying MySql database [17]. In order to integrate the knowledge base into the framework developed in this thesis, a thin database access layer on top of Scala [18] was written as well. As the background database changes fast, it might be advisable to just use the web interface though. It is able to produce database contents in Json [19] and Xml [20], however, downloading the whole database is time consuming.

2.4 Analysis and Visualization

The analysis tools that come with the simulator perform analytic tasks on the simulation output to search for interesting properties of the cell evolution. They are also written in Matlab and provide measures for general cell behavior for model verification. There is also an extensive visualization suite that can be found on the wholecell website [21]. It shows different simulation behaviors in an intuitive way and can be used to display any data generated by simulations. Biological system analysis is not considered further in this thesis, as it is a huge topic in itself.

2.5 Strengths and Weaknesses of the Model

Jonathan Karr's whole cell model is an impressive feat. The work and information that flowed into it is enormous, it combines research from over 900 papers, including over 1900 parameters, the complete set of genes of *Mycoplasma Genitalium* and much more. As stated in [1] it has a predictive capacity, allows to state novel hypotheses, provides a fast means to make biological discoveries and helps designing in a rational way.

However, its huge complexity is also the biggest hindrance to get started with the model. As the Matlab code is really verbose and redundant, mixes simulation with model logic, allows anything to access everything and uses complicated matrix accesses all the time, just to get an understanding of the model takes a long time. The model and simulator are strictly bound to Matlab which requires licences for simulation. Finally, there is no theoretical base for the simulator assumptions (of the 1 second time step, the random ordering, splitting of metabolites and so on).

In this thesis some of the points are tackled, new ideas are tested, a theoretical framework is developed and finally implemented and analyzed.

Chapter 3

A Formal Description of Modular Simulations

3.1 Existing Modular and Distributed Models

There is a huge range of modeling techniques available: dynamical systems, differential equation models, Boolean networks, stochastic models, flux balance analysis, finite automatons, Petri nets and many more. The models of interest in this thesis are the ones which can be simulated in a distributed manner. Trivially this includes a lot of network models (though the distributed simulation of them doesn't always make sense) but also differential equations for example, which can be modularized by calculating immediate steps in a distributed manner [22] or even by using more elaborate tricks like distribution of the ODE solution vector [23]. Also, there is some research being done on ways to modularize existing models - primarily for analysis purposes, but also to simulate in distributed manners. This is especially viable for huge networks as can for example be found in biological reaction pathways [24].

Of particular interest for modular simulation are techniques that allow integration of various different modeling techniques. The main purpose of such techniques is to allow a modeler to choose a favorite modeling technique for each component in the system and to build a bigger system by selecting and aggregating different components.

3.2 Formalizing the Karr Model

The Karr model is primarily sequential with some techniques to help achieve fairness between processes. Apart from running the processes in random order at each time step, the main technique used in this respect is *pre-allocation* which splits up (high-frequency) resources and assigns them to different processes for a simulation round of Δt. High-frequency in this respect means parts of the simulation state that are often accessed by many processes within Δt (for a more detailed discussion see section 3.2.1). In order for this to be possible, processes have to specify a *requirements function* that denotes the current need or flux for each resource. Resources are then distributed proportional to the requirements of the different processes.

Another possibility for processes to introduce changes into the system are so-called *side effects*. They have been introduced to reduce coupling between processes. A side effect is of additive nature and is integrated into the system after the process has finished executing. However, side effects can easily also be implemented by the general architecture, as each process has exclusive access to the whole simulation state (with exception of metabolites as those are split). For the following formalization the notion of side effects and general change on the system are unified.

As the requirements function has to be specified for each process in addition to the evolve function and because processes then only get their share of the whole system to work upon, several new techniques based on the common formalism are developed and tested in this thesis. The *change merging* technique for example merges the results of different processes into a consistent overall result after each Δt. This is done by an intersection oracle that has access to the complete simulation state (however, this can also be implemented in a completely distributed way for efficiency).

3.2.1 The Building Blocks of the Karr Model

This section introduces several common terms across different methods of modular simulations.

As already mentioned before, we basically have *state variables* and *processes*. In addition, in the Karr model, there are *states*, *side effects* and the global *simulation object*. For the formalization it is important to notice several things though:

- States provide bundling of resources (comparable to namespacing), but are of no relevance to the formal arguments and will thus not be included in the following outline.
- Side effects can easily be described by the general way processes interact with the state, as processes have complete access to all state variables at all times. Also, the mechanics of processes can be completely described in terms of side effects

(thought a little more generalized than in the Karr simulation), which will be used in the formalism.

- The simulation object is an all-knowing ‘oracle’. It is considered as thus in the formalism, as long as applicable (later, in distributed versions, the simulation object will be distributed too).

A *state variable* v is any modifiable variable within the model. This can for example be a copy number of a metabolite (described by an integer) or a piece of the genome (a String), a link from a ribosome to a certain part on the DNA (a reference) and many more. In the Karr model, most variables are Integers representing copy numbers of elements in the cell, but some are Strings and Doubles as well. For the formalization, a state variable s_i is any variable that can be changed by process interaction:

$$v_i(t) \in S, i \in \{1, \dots, n\} \tag{3.1}$$

Where S is the state space of the variable, e.g. $\mathbb{R}$ or $\{true, false\}$.

There is a special type of state variables, called *resources.* A resource $r_j(t) \in S$ is a variable that can be split into pieces which are distributed over processes:

$$r_j(t) \in S, j \in \{1, \dots, m\} \tag{3.2}$$

In the Karr model this happens with variables in the *Metabolite* state. Those variables are all Integers denoting copy numbers of metabolites. At the beginning of a simulation round they are distributed proportional to some requirements of each process. Distribution in this sense means giving fractions of the total Integer value to each process P_k:

$$r_{j,k}(t) = \alpha_{j,k}(t) * r_j(t), \qquad \sum_k \alpha_{j,k}(t) = 1 \tag{3.3}$$

The global state vector $S(t)$ contains all state variables and resources:

$$S(t) = \begin{pmatrix} v_1(t) \\ \dots \\ v_n(t) \\ r_1(t) \\ \dots \\ r_m(t) \end{pmatrix} \tag{3.4}$$

For the purpose of easier notation there are two sub-vectors of $S(t)$: $V(t)$ and $R(t)$, denoting variables and resources respectively. The state vector can thus be written as $S(t) = V(t)||R(t)$ (using the vector concatenation formalism where vectors are regarded as lists).

A *process* $P_k \in \{P_1, ..., P_p\}$ is an encapsulation of functionality. The primary function is $Evolve(\cdot)$ which produces a new state $S_k(t + \Delta t)$ of the simulation system after a time period Δt starting from initial conditions $S(t)$ at time t. S again denotes the space that resources span:

$$\begin{aligned} P_k.Evolve: \quad & S^n \times \mathbb{R} \times \mathbb{R} \to S^n \\ & (S(t), t, \Delta t)) \mapsto Evolve_k(S(t), t, \Delta t) \end{aligned} \tag{3.5}$$

Where $Evolve_k(\cdot, \cdot, \cdot)$ is the concrete evolve function implementation that transforms the state vector. A process with this simple definition doesn't have any state itself. However, it is important to note that in the current model private state for processes is easily possible by just writing it in the global state vector. The global state vector has some benefits for argumentation as we will see later.

An addition needed to the process definition in order to comply with the Karr model are *requirement functions*. A requirement function denotes the change a process would like to introduce to the system before it really does so. In the Karr model, this only happens for resources, where processes denote how much of a given resource they would like to consume. Another noteworthy point, requirement functions are calculated in saturated regions (meaning infinite supply of resources), thus decoupling the distributed resources from their actual values (they only depend on variables). This is done to reduce computational overhead. However, the requirement functions could also just be an evaluation of $Evolve(\cdot, \cdot, \cdot)$. The formal definition of the requirements function is thus:

$$\begin{aligned} P_k.Require: \quad & S^n \times \mathbb{R} \times \mathbb{R} \to S^n \\ & (S(t), t, \Delta t)) \mapsto Require_k(S(t), t, \Delta t) \end{aligned} \tag{3.6}$$

With the two basic components of variables and processes noted down, let's introduce some special components.

Variables (and thus also resources) can be outfitted with *restrictions*. A restriction is a function that evaluates to *true* or $false$ for a variable at a given time. It evaluates to

Type	Difference Definition
Numeric (Double, Integer, ...)	$old \triangleright new = new - old$
Boolean	$old \triangleright new = new$
Char	$old \triangleright new = new$
String	$old \triangleright new = if$ (length has changed) $then\ new$ $otherwise\ Diff(old, new)$ Could also use some shortest distance algorithm.
List	Elemental difference

Table 3.1: Change definition for various primitives

true if the restriction isn't violated, to $false$ otherwise:

$$R : S \rightarrow \{true, false\} \tag{3.7}$$

An example restriction could be that the variable value may never fall below zero ($s_i(t) \mapsto s_i(t) \geq 0$). We can use restriction functions to determine violations in the model and to integrate the state changes of different processes. A *violation* is a restriction violation, i.e. a restriction that evaluates to $false$. Requirements are of particular interest when gradually integrating changes into the global state vector.

In order to make the formalism a little more approachable, the notion of a *change* is introduced. A change C_k has the same structure as the state vector $S(t) \in S^n$ and describes the change a process P_k would like to perform on the simulation state during $t + \Delta t$. The change a process calculates is basically the collection of all different smaller changes that happen on the simulation state during the execution of $Evolve_k(\cdot, \cdot, \cdot)$. In the above definition of a process the difference between the old and new state $S_k(t + \Delta t) - S_k(t)$ corresponds to the change C_k. As the difference operator can only be used for numeric variables, it makes sense to introduce the more general operator $\triangleright$ for state difference to calculate changes. The difference in this sense is generalized to apply for all primitive objects, see table 3.1. The generalization is not complete and can be extended to be defined on arbitrary complex objects. The change vector then becomes:

$$C_k(t) = S_k(t) \triangleright S_k(t + \Delta t) \tag{3.8}$$

A change on the system can either be linearly interpolated over Δt or happen at a user defined instant (so-called *atomic* change). As can be seen, the important approximation made here is that the changes on the system state within Δt are linear. This allows for faster reintegration of different processes, especially in the case where violations happen.

Table 3.2 summarizes the introduced formalism. Using this general formalism the following sections show interesting properties and algorithms that build on those.

Name	Formula	Description
State Variable	$s_i(t) \in S$	Anything that can be changed during the process of simulation.
Resource	$r_i(t) \in S$	Special version of variable that can be shared between processes (e.g. additive).
State Vector	$S(t) = V(t) \| \| R(t)$	Global vector containing all variables and resources.
Process	$P_k : S^n \times \mathbb{R} \times \mathbb{R} \to S^n$	Function that evolves the state.
Requirement	$R_k : S^n \times \mathbb{R} \times \mathbb{R} \to S^n$	Requirement function of process P that denotes how much resources a process wants to consume.
Restriction	$R : S \to true, false$	Restriction on variable.
Change	$C_k(t) = S_k(t) \triangleright S_k(t + \Delta t)$	Change that process introduces on system.

TABLE 3.2: Terms used in the ubiquitous description of modular models

3.2.2 Connecting the Blocks

In the Karr whole cell models, the processes and states are linked in a quite straight forward way (see also figure 2.2). Variables from the *Metabolite* state (which denote copy numbers of metabolites) are treated as resources from the above definitions. The rest of the state vector are simple variables that can only be accessed by a single process at a time. Algorithm 1 describes the execution of the Karr model (without side effects, due to reasons described above, mainly because side effects can be seen as additive changes on variables).

Algorithm 1 The Karr Model Execution Algorithm

Precondition: $S[t]$ the state vector with initial values $S[0]$. It will be calculated and filled in for $t \in \{1, ..., t_{MAX}\}$. P the set of processes P_k

```
1:  function RunSimulation($S[0]$)
2:      for $t \leftarrow 1$ to $t_{MAX}$ do
3:          $S[t] \leftarrow S[t-1]$
4:          for $P_k \leftarrow P_1$ to $P_p$ do
5:              $Req_k \leftarrow P_k.Require(S[t], t, \Delta t)$
6:          for $P_k \leftarrow P_1$ to $P_p$ do
7:              $R_k \leftarrow Split(R[t], Req_k, \{Req_1, ..., Req_p\})$        $\triangleright$ $Split(\cdot)$
8:          for $P_k \leftarrow Permutate(P)$ do
9:              $(V[t], R_k) \leftarrow P_k.Evolve((V[t]||R_k), t, \Delta t)$
10:         $S[t] \leftarrow (V[t]||Merge(\{R_1, ..., R_p\}))$        $\triangleright$ $Merge(\cdot)$
11:     return $S$
```

As is visible, the algorithm 1 needs two more functions to work, $Split(\cdot)$ and $Merge(\cdot)$. It turns out those two functions are of particular interest as different implementations of those allow to completely decouple processes. In the Karr model however, the functions are defined as follows (figure 3.1 illustrates the process):

$$Split(R(t), Req_k, \{Req_1, ..., Req_p\}) = \frac{Req_k}{\sum_k Req_k} \cdot R(t) \tag{3.9}$$

$$Merge(\{R_1, ..., R_p\}) = \sum_k R_k \tag{3.10}$$

Where the fraction in equation 3.9 is denoted as:

$$\alpha_k = \frac{Req_k}{\sum_k Req_k} \quad \in [0, 1] \tag{3.11}$$

As $R(t)$, the Req_k's and the R_k's are all vectors, the values returned by the functions are vectors too. Namely, equation 3.9 distributes all resources (metabolites) proportional to the need of the different processes and equation 3.10 sums up all excess resources that haven't been used by the processes.

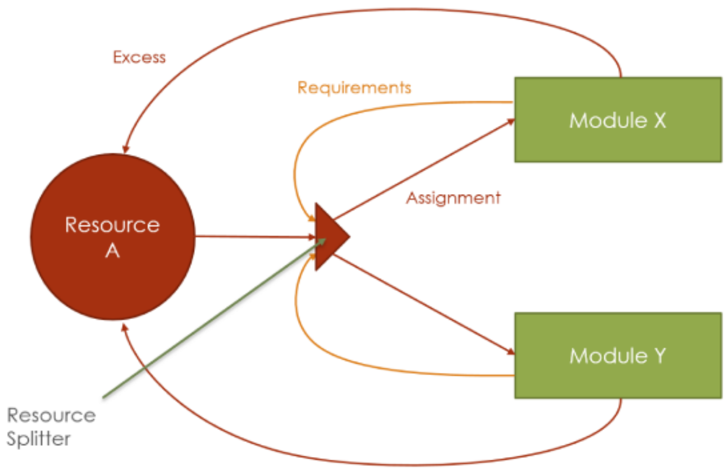

FIGURE 3.1: The resource splitter used in the Karr model

Further noteworthy in algorithm 1 is the fact that the simulation state is permanently being overwritten by the evolve functions of the different processes (in particular the $V[t]$ part of it). Another interesting point is the returning of the complete simulation state over all t's after the function terminates. This can now be used to further analyze or to just store the simulation output.

3.3 A Ubiquitous Description of Modular Models

This section boils down the above formalism by cutting out the *requirements* function and letting the *evolve* functions only return changes. Also, the step size isn't rigid within the whole simulation anymore, but instead freely selectable for each process at any time.

First of all, the notion of process and state is reduced to the following: A state vector consisting only of resources (everything is distributable now, as this allows complete parallelism), a set of processes which define an evolve function and the notion of restrictions that can be checked on resources. The state vector thus is:

$$S(t) = \begin{pmatrix} r_1(t) \\ ... \\ r_n(t) \end{pmatrix} \tag{3.12}$$

This vector is defined for various t, but is usually not continuous. The processes define an evolve function that returns a change they would like to introduce on the system:

$$\begin{aligned} P_k.Evolve: \quad & S^n \times \mathbb{R} \times \mathbb{R} \rightarrow S^n \\ & (S(t), t, \Delta t)) \mapsto C_k \end{aligned} \tag{3.13}$$

Where a change is defined as:

$$C_k(t, \Delta t) = S(t) \triangleright S(t + \Delta t) \tag{3.14}$$

Restrictions are exactly the same as before, defined as:

$$R : S \rightarrow \{true, false\} \tag{3.15}$$

In practice, the simulator can enforce processes to specify special functions, like the requirement function in the Karr model. This is not generally required though. The following section will elaborate on other strategies to simulate whole cell models.

3.4 Different Implementations of $Split(\cdot)$ and $Merge(\cdot)$ and General Algorithms to Intersect Changes

With the basic working of the Karr model formalized and explained, let's have a look at different $Split(\cdot)$ and $Merge(\cdot)$ methods, then extend the formalism a little and use that to develop different methods to integrate modules into a bigger simulation.

In algorithm 1 the functions $Split(\cdot)$ and $Merge(\cdot)$ play an important role. The first thing to note is that they both only act on resources, variables are untouched. This leads to some important questions: what qualifies a variable as a resource? What does splitting of a resource mean and why can't variables be treated the same way? Can we treat all variables as resources?

For one can note that if there were only resources, all the processes could easily run in parallel. A possibility thus is to build models only from resources. However, this is not always possible as, in a general model, people might want to use non-shareable variables like Booleans and strings. A second possibility thus tries to handle every variable as resource. For this the splitting and merging functions have to be adapted, which will now be discussed.

Let's first discuss possibilities for $Split(\cdot)$ methods. Resources can be split equally, proportional to some need, randomly, not at all or by some arbitrary function. The equal splitting seems to be fair, however there might be processes that only act on a small subset of resources and thus don't need shares of all resources. Also, in biological systems different processes will use up more resources faster than others. The proportional splitting is used in the Karr model, where the need (requirements) are calculated by a function specified by every process. The drawbacks are discussed in section 4.1.2. Summing them up, the system consumes less resources than a monolithic system. Random splitting primarily makes sense for "unsplittable" resources as those have to be distributed somehow. For resources like metabolites a random assignment will have implications on the system except for very small simulation steps. Arbitrary functions for splitting can have pros and cons, but will mostly have to be implemented using some knowledge about the system, which is not in the spirit of this thesis, where processes are seen as black boxes.

This leaves proportional splitting with the addition of random splitting of variables and no splitting at all, further denoted as pre- and post-allocation. The possibilities for the $Merge(\cdot)$ function depend on the splitting algorithm, but there are many more possibilities for merging.

3.4.1 Pre-Allocation Strategies

In the case of proportional / random splitting as discussed above, the assumption is that resources are of additive nature. A merge function thus needs to be like the Karr merge function for all additive resources:

$$Merge(\{R_1, ..., R_p\}) = \sum_k R_k \tag{3.16}$$

For all other resources, no merging is necessary, as they have been distributed randomly. Thus only one process per time step Δt has access to the resource.

3.4.2 Post-Allocation Strategies

In case we omit splitting the resources at the beginning, the $Merge(\cdot)$ function has to take over the task of assigning resources fairly to processes in retrospect. Remember the notion from section 3.2.1 equation 3.8 that defines the change a process wants to introduce to the system as $C_k(t) = S_k(t) \triangleright S_k(t + \Delta t)$. As every process now acts on the complete simulation state and posts a desired change on the system, the merge function has to find a fair merge of all requests. The merge function can either be thought of a central change integration oracle or even a distributed mechanism. The process of post-allocation is depicted in figure 3.2.

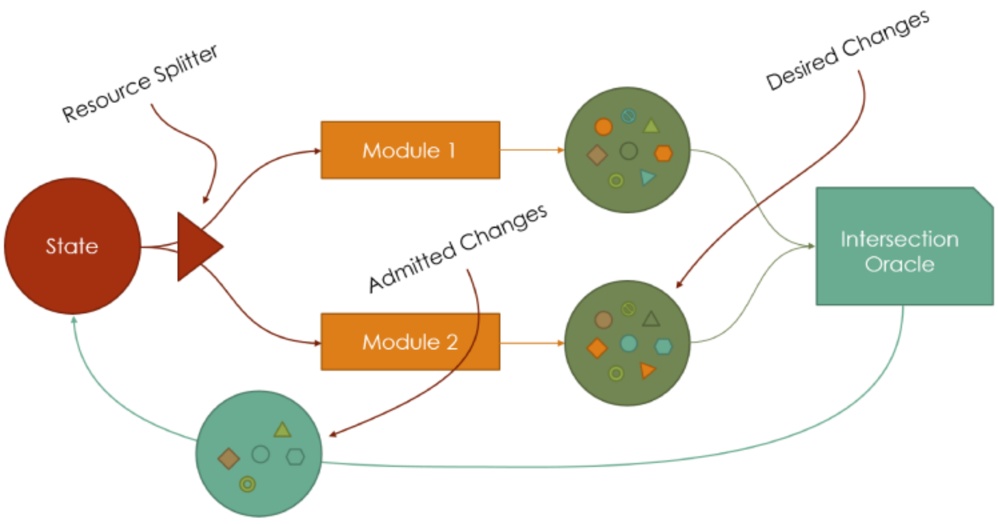

FIGURE 3.2: The structure of post-allocation algorithms with an intersection oracle

In the following, different merging strategies are presented and discussed. They are implemented within the framework and the resulting performance is discussed in chapters 5 and 6.

3.4.2.1 Smash Strategy

This first strategy just squeezes as much change as possible into the system, and discards the rest. Algorithm 2 describes the strategy. There is no notion of *connected* change, i.e. every change (remember that a process can post multiple changes, e.g. atomic ones plus some over time) gets equal precedence. *Connected* changes on the other hand would ensure fairness not on change level, but on process level (i.e. every process gets equal precedence).

This strategy is very fast, but makes use of random selections between changes. Thus it can be used for fast explorations of a system space, for systems consisting of additive resources and for systems with a small time step (where violations are not common), but shouldn't be considered otherwise.

Algorithm 2 Smash Strategy

Precondition: $S[t=0]$ the initial state, P the set of processes P_k, t_{end} the maximum simulation time and Δt the time step

```
1: function RunSimulation($S[0]$)
2:     for $t \leftarrow 0$ to $t_{end}$ in steps of $\Delta t$ do
3:         for $k \leftarrow 1$ to $p$ do
4:             $C_k \leftarrow P_k.Evolve(S[t], t, \Delta t)$
5:         $S[t+\Delta t] \leftarrow Merge(\{C_1, ..., C_p\})$
6:     return $S$

7: function Merge($\{C_1, ..., C_p\}$)
8:     for $i \leftarrow 0$ to $m$ do                ▷ Iterate through resources
9:         if $S_i$ is additive then
10:            $S_i \leftarrow C_{1,i} + ... + C_{p,i}$
11:        else
12:            $S_i \leftarrow RandChoose(C_{1,i}, ..., C_{p,i})$
13:    return $S$
```

3.4.2.2 Change Merging Strategy

Algorithm 3 describes the change merging strategy. As this strategy is mathematically more involved, more details are given in the text below.

Processes produce a change on the system given an initial state, a time and a delta time by their $Evolve_k(\cdot)$ function. To keep the system initially simple, we assume the simulator farms out at t and collects changes at $t + \Delta t$ for all processes synchronously. All changes together now form the change or *flux matrix* $\Phi : m \times p$ (remember that a change C_k is a vector $\in S^m$):

$$\Phi(S(t), t, \Delta t) = (C_1, C_2, ..., C_p) = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,p} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,p} \end{pmatrix} \tag{3.17}$$

This flux matrix sometimes will lead to violations of the system. Namely for every resource r_j all associated violation functions $v_{j,k}$ can be evaluated and violations noted.

The merging is obviously a lot more difficult than in the pre-allocation case. We can now use the fact that the matrix describes a flux though, so the violations happen at

Algorithm 3 Change Merging Strategy

Precondition: $S[t = 0]$ the initial state, P the set of processes P_k, t_{end} the maximum simulation time and Δt the time step

```
1: function RunSimulation($S[0]$)
2:     for $t \leftarrow 0$ to $t_{end}$ in steps of $\Delta t$ do
3:         for $k \leftarrow 1$ to $p$ do
4:             $C_k \leftarrow P_k.Evolve(S[t], t, \Delta t)$
5:         $S[t + \Delta t] \leftarrow Merge(\{C_1, ..., C_p\})$
6:     return $S$

7: function Merge($\{C_1, ..., C_p\}$)
8:     $\Phi \leftarrow [C_1, ..., C_p]$
9:     $\Phi_n \leftarrow [C_1, ..., C_p]$
10:    $t \leftarrow 0$
11:    while $t < \Delta t$ do
12:        $x \leftarrow Ones(p)$                                ▷ Create array of ones
13:        $t_v \leftarrow \Delta t - t$
14:        for $i \leftarrow 0$ to $m$ do                      ▷ Iterate through resources
15:            if Violation in $S_i$ then
16:                $t_v \leftarrow min(t_v, t_{v,i})$
17:            else
18:                Temporarily remove row $\Phi_i$ from $\Phi$
19:        $S \leftarrow AdvanceSystem(t_v, \Phi_n)$
20:        $t \leftarrow t + t_v$
21:        if any violation then
22:            $\Phi_n \leftarrow \Phi \cdot LinProgSolve(max(x), \Phi \cdot x \geq 0, x \in [0, 1])$
23:    return $S$
```

fixed times. Let's denote the time of the first violation with t_v. The first violation can be calculated as the first time t when:

$$\begin{aligned} V\left(c_{1,1} \cdot t, c_{1,2} \cdot t, ..., c_{1,p} \cdot t, s_1\right) &= true \quad | \\ V\left(c_{2,1} \cdot t, c_{2,2} \cdot t, ..., c_{2,p} \cdot t, s_p\right) &= true \quad | \\ \ldots &= true \quad | \\ V\left(c_{m,1} \cdot t, c_{m,2} \cdot t, ..., c_{m,p} \cdot t, s_m\right) &= true \end{aligned}$$

This finding of t can be sped up as we assume the flux within Δt is constant, i.e. the system is linear during this time, so searching for violations is quite straightforward. After the first violation has been detected, the flux matrix could be reduced to zero, however, there is a better solution which involves recalculating the flux matrix so that no violations occur any more (up to a next violation $t_{v'}$).

Let's first look at additive resources. The flux of those resources has to satisfy the following requirements at a violation point t_v:

$$\begin{aligned} &\text{maximize } x \\ &\text{subject to } \Phi \cdot x \geq 0 \\ &\text{and } x \in [0,1] \end{aligned}$$

Which is a problem that can be solved numerically using linear programming. The flux matrix Φ is then multiplied by the vector x which will yield a new flux matrix without violations up to $t_{v'}$. The algorithm can thus be applied repeatedly until the time interval Δt has passed. This allows for arbitrary changes to be pushed through the system in a time interval Δt (the discussion if that is a good thing to do follows later). Figure 3.3 shows an exemplary system consisting of three resources. Even though the time step Δt is 1 second, after 0.5 and approximately 0.8 there are changes in the fluxes, as the resources otherwise would fall into the negative.

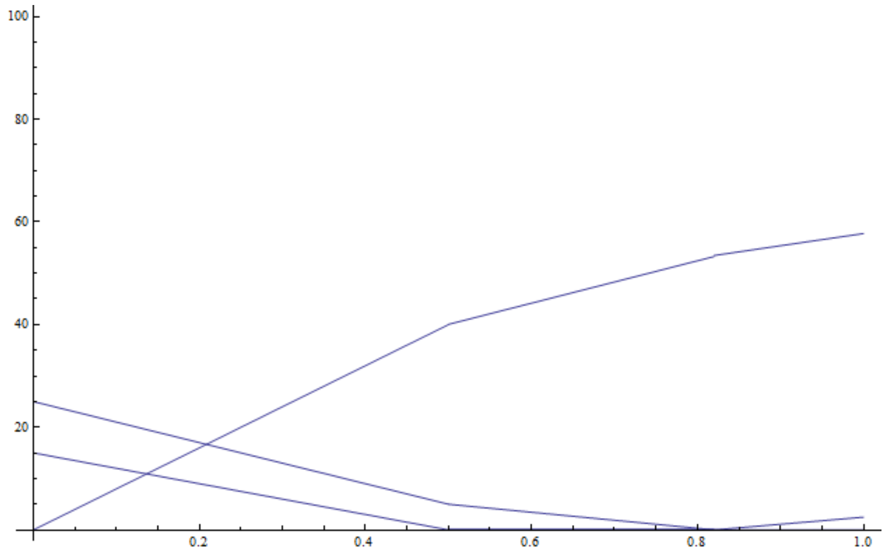

Figure 3.3: The change merging strategy, which adapts current fluxes to satisfy all restrictions but still keeping the flux maximal

Worth to note is also that the linear programming problem above may not have a solution except $x = 0$. However, this is a valid and acceptable solution within the simulation.

Also, the linear programming problem only has to be solved for conflicting and interacting resources, i.e. the flux matrix Φ can be reduced before searching for a solution.

This strategy has an interesting property in that it allows to force an arbitrary time step whilst still making sure that processes are treated fair. It is comparable to the Karr strategy but using post-allocation. However, the recalculation of the change matrix can take lots of time (abolishing the gain from the forced time step) and the resulting system can show oscillating effects for large Δt's (which can happen for all Euler-like strategies without time step adaption though). For well-behaving systems, it turns out to be easier to use a time-adapting strategy or an Euler-like strategy with a small time step.

3.4.2.3 Synchronization Points

The synchronization points algorithm synchronizes the system at given time points. If there are violations, the time points are reduced by a factor of two until there are no more violations. Algorithm 4 describes the program flow. Figure 3.4 shows the strategy. In the figure, processes $P1$, $P2$ and $P3$ initially have conflicting change requests, after dividing the time step, $P2$ and $P3$ still conflict in the first half.

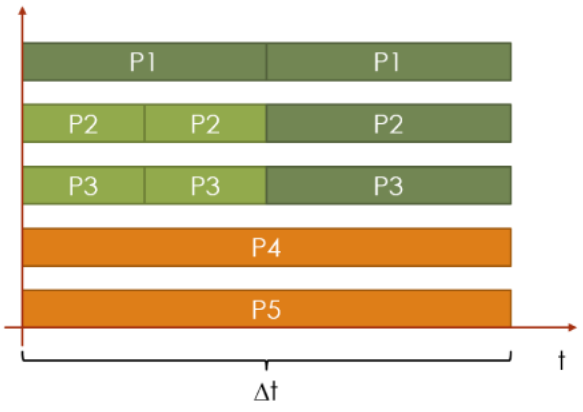

FIGURE 3.4: The synchronization points strategy, orange is the initial time step, green means violations happened and the time step was divided by two

The strategy proves to be a good cut between simulation duration and accuracy as it only reduces time steps for conflicting processes. If the overall time step is selected too large though, recalculations are necessary more often, reducing the efficiency of the strategy. A further drawback is noncompliance with different process time steps.

3.4.2.4 Independent Time Scales

The system gets more interesting and usable when integrating different time scales. The algorithm heavily relies on the linearity assumption within Δt (comparable to any other numerical solving strategy) and uses this to integrate changes from modules with different time scales. Algorithm 5 depicts the logic.

Algorithm 4 Synchronization Points

Precondition: All Δt's multiples of each other. $S[0]$ the initial state, $t = 0$ the simulation time, Δt an array of Δt's of the processes, P the set of processes P_k

```
function RunSimulation($S[0]$)
    for $t \leftarrow 0$ to $t_{end}$ in steps of $\Delta t$ do
        $\{C_1, ..., C_p\} \leftarrow Changes(S[t], \Delta t, \{P_1, ..., P_p\})$
        $S[t + \Delta t] \leftarrow Merge(\{C_1, ..., C_p\})$
    return $S$

function Changes($S, \Delta t, \{P_a, ..., P_b\}$)
    for $P_k \leftarrow P_a$ to $P_b$ do
        $C_k \leftarrow P_k.Evolve(S, t, \Delta t)$
    while $(V \leftarrow Violators(\{C_a, ..., C_b\})).size \neq 0$ do
        $\{C_x, ..., C_y\} \leftarrow Changes(S, \Delta t/2, V)$
        $S \leftarrow ApplyChanges(S, \{C_x, ..., C_y\}, \Delta t/2)$
        $\{C_x, ..., C_y\} \leftarrow Changes(S, \Delta t/2, V)$
        $\{C_a, ..., C_b\} \leftarrow Merge(\{C_a, ..., C_b\}, \{C_x, ..., C_y\})$
    return $\{C_a, ..., C_b\}$
```

Where $ApplyChanges(\cdot)$ applies changes to the state (for a given Δt) and $Merge(\cdot)$ selects max one change vector at any given time for any given process.

The algorithm isn't parallel, but provides a preliminary step for the parallel algorithm following in short. At thus it isn't highly efficient (especially if processes with small Δt's take long to simulate) and not recommended for real use.

3.4.2.5 Distributed Strategies

The final algorithm incorporating all good things from the above ones is algorithm 6, distributing the processes, allowing them to run on completely different time scales. It could further be distributed by additionally distributing the state over multiple instances. Also, within the strategies there is a lot of optimization possible, e.g. by selectively sending only parts of the state. Figure 3.5 displays the algorithm graphically. Orange are the parts where the strategy forces modules to use a smaller time step in order to get to a common synchronization point which then is used to let processes run in parallel again. The integration happens by *slicing*, where changes with large Δt's are sliced up and integrated alongside the shorter ones.

Algorithm 6 provides an accurate, completely distributed variant to simulate models. It provides a possibility to hook uchronic execution as described in [25] (which lets processes guess the changes of other processes) and runs on arbitrary time scales. It is

Algorithm 5 Independent Time Scales

Precondition: $S[0]$ the initial state, $t = 0$ the simulation time, Δt an array of Δt's of the processes, P the set of processes P_k

```
function RunSimulation($S[0]$)
    $t_{prev,k} \leftarrow t$
    $t_{curr,k} \leftarrow t$
    $t_{next,k} \leftarrow t + \Delta t_k$
    while $t < t_{end}$ do
        $i \leftarrow min_k(t_{next,k})$
        $C_i \leftarrow P_i.Evolve(S[t_{curr,i}], t, \Delta t_i)$
        $V \leftarrow Violations(\{C_1, ..., C_x\})$
        if $V$ then
            $\Delta t_i \leftarrow (t_v - t_{curr,k})/2$
            $t_{next,i} \leftarrow t_{curr,i} + \Delta t_i$
            for $P_k \leftarrow V.Violators - \{P_i\}$ do
                $\Delta t_k \leftarrow t_{curr,i} - t_{prev,k} + \Delta t_i$
                $t_{curr,k} \leftarrow t_{prev,k}$
                $t_{next,k} \leftarrow t_{prev,k} + \Delta t_k$
        else
            for $t_x \leftarrow [t_{curr,i}, t_{next,i}]$ do
                $S[t_x] \leftarrow Merge(\{C_1, ..., C_x\})$
            $t_{prev,i} \leftarrow t_{curr,i}$
            $t_{curr,i} \leftarrow t_{next,i}$
            $t_{next,i} \leftarrow t_{curr,i} + \Delta t_i$
            $t \leftarrow min(t_{curr,k})$
    return $S$
```

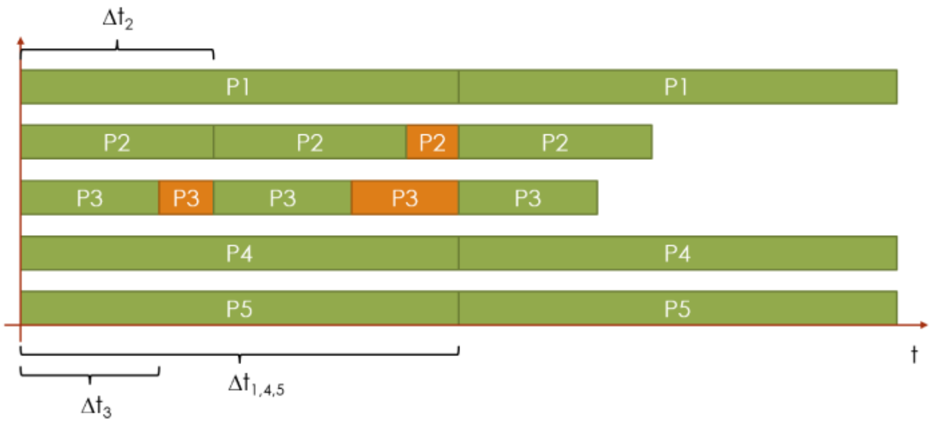

FIGURE 3.5: Independent time scale strategy, orange are parts to catch up with other processes

quite resource intensive and should thus be used for models where the individual process simulation times are high.

Algorithm 6 Distributed with Positive Time Step Adaption

Precondition: $S[0]$ the initial state, $t = 0$ the simulation time, Δt the Δt's of the processes, P the set of processes P_k, α the increase factor with which the simulator tries to increase Δt's every time step, Δt_{max} the maximal allowed time step

```
function RunSimulation($S[0]$)
    $t_{prev,k} \leftarrow t$                                        ▷ Set up simulation
    $t_{curr,k} \leftarrow t$
    $t_{next,k} \leftarrow t + \Delta t_k$
    for $k \leftarrow 1$ to $p$ do
        $running \leftarrow \{k, t, \Delta t_k\}$
        $P_k.Evolve(S[t_{curr,k}], t, \Delta t_k)$

function Receive($C_i$)
    $V \leftarrow Violations(\{C_1, ..., C_x\})$
    if $V$ then
        $\Delta t_i \leftarrow (t_v - t_{curr,k})/2$
        $t_{next,i} \leftarrow t_{curr,i} + \Delta t_i$
        for $P_k \leftarrow V.Violators - \{P_i\}$ do
            $\Delta t_k \leftarrow t_{curr,i} - t_{prev,k} + \Delta t_i$
            $t_{curr,k} \leftarrow t_{prev,k}$
            $t_{next,k} \leftarrow t_{prev,k} + \Delta t_k$
            $StopAndClearRunning(k)$
            $running \leftarrow \{k, t, \Delta t_k\}$
            $P_k.Evolve(S[t_{curr,k}], t, \Delta t_k)$
        $running \leftarrow \{i, t, \Delta t_i\}$
        $P_i.Evolve(S[t_{curr,i}], t, \Delta t_i)$
    else
        for $t_x \leftarrow [t_{curr,i}, t_{next,i}]$ do
            $S[t_x] \leftarrow Merge(\{C_1, ..., C_x\})$
        $\Delta t_i \leftarrow min(\Delta t_i * \alpha, \Delta t_{max})$
        $t_{prev,i} \leftarrow t_{curr,i}$
        $t_{curr,i} \leftarrow t_{next,i}$
        $t_{next,i} \leftarrow t_{curr,i} + \Delta t_i$
        $idx_{min} \leftarrow min_{k's}(t_{curr,k})$
        for $id \leftarrow idx_{min}$ do
            $\Delta t_{id} \leftarrow min(\Delta t_{id}, min(t_{curr} \setminus idx_{min}) - \Delta t_{id})$
            $running \leftarrow \{id, t_{curr,id}, \Delta t_{id}\}$
            $P_{id}.Evolve(S[t_{curr,id}], t_{curr,id}, \Delta t_{id})$
        $t \leftarrow min(t_{curr})$
        if $t > t_{max}$ then
            $StopSimulation()$
```

3.4.3 An Informal Comparison

At a first glance the preallocation mechanism seems like a good choice from a point of efficiency, the whole search for a non-violating merge within a time interval Δt falls away, the integration of different changes reduces to a simple sum. Things get more complicated though, as there are two main concerns:

- Processes need to specify a “requirements function”. This function can be the function C_i itself, however this results in double calculations every time step.
- Processes see “sub-state” of the whole simulation, mainly the resources they get assigned. This can be changed by allowing them to read as much as they want but only to write back the assigned resources.

For those reasons, other strategies were developed. The change merging strategy which basically models the Karr one without pre-allocation suffers from a similar problem of computational overhead. For this reason, strategies with time step adaption are preferable.

A further problem with all modularized strategies is that within Δt processes don’t know anything about consumption and production of other processes, thus there will always be an error introduced into the system. This is covered in more detail in chapter 4.

Chapter 4

Properties of Modular Simulations

4.1 Correctness: Implications of Karr's $Split(\cdot)$ and $Merge(\cdot)$

In the Karr model, split and merge is only applied on metabolites. Metabolites in this sense means their copy numbers, which is the biological term for count. That means the functions can be completely additive, as those resources are used by different processes in reality as well. There is one big benefit of splitting them in advance: Resource restrictions are never violated, as long as processes themselves don't violate the restriction. For this to make sense, let's look at the single restriction on metabolites:

$$\begin{aligned} R_{Met}: \quad & S \to \{true, false\} \\ & r_j(t) \mapsto r_j(t) \geq 0 \end{aligned} \tag{4.1}$$

This means that any metabolite resource in the Karr model always has to be strictly positive. The physical interpretation is that molecules can not exist in negative amounts. The implication of the additive distribution now is that if every process itself respects this restriction, the restriction is globally respected. However, there are two points worthy of discussion here:

- Why are metabolites split? The architecture relies on sequential processing of processes anyways, so why not treat metabolites as variables as well?
- What are the implications of the splitting as it is used in the Karr model? In specific what is the effect of processes seeing only a sub-state of the model state?

4.1.1 Why Metabolites Need to be Split

The answer as stated in the supplementing documentation of [1] is that metabolites are heavily shared between processes. The authors argue for the necessity of it by providing counter examples:

1. Suppose the same algorithm is used, but $Split(R[t], \cdot, \cdot) = R[t]$, meaning that every process gets the complete resource vector. The $Merge(\cdot)$ function is adapted so it sums the changes of each process and applies this total change to the previous state. In total the processes could now violate the restriction R_{Met}, namely use up more than the available metabolites.

2. Suppose metabolites are treated as variables (not resources) and processes are not permuted each round. The first process can now always use up all of the metabolites in a given round, depriving other processes of their intended effect on the system.

3. Suppose metabolites again as variables, but random execution of processes. This would result in high fluctuations, as processes in the beginning would deplete resources making the system uncontrollably oscillate at the Δt interval.

The main point in splitting metabolites is to ensure process fairness. This is only required for so called metabolites, as those are involved in high-frequency reactions. This basically means that at some given point during the simulation:

$$\frac{\delta r}{\delta t} \leq -r(t) \tag{4.2}$$

Which means that during a discrete simulation interval $\Delta t = 1$s the resource would be completely used up. This forms the notion of high-frequency variables and determines if a variable needs to be split (of course only for restricted variables). The concrete strategy to determine which variables can and have to be shared depends on the system, but could generally be implemented using some sensitivity analysis.

4.1.2 Effects of Dividing State

The effects of the splitting technique applied has a huge effect on the system. Let's take a first look by examining a simple mass action system.

4.1.2.1 Consumption Errors in a Simple System

The system

$$\frac{\delta r(t)}{\delta t} = c_1 \cdot r(t) + c_2 \cdot r(t) \tag{4.3}$$

is to be split in two subsystems, whereas each has the requirement c_i. The resulting system is:

$$\begin{aligned}
\frac{\delta r_1(t)}{\delta t} &= c_1 \cdot \alpha_1 \cdot r(t) && (4.4)\\
\frac{\delta r_2(t)}{\delta t} &= c_2 \cdot \alpha_2 \cdot r(t) && (4.5)\\
\frac{\delta r(t)}{\delta t} &= \frac{\delta r_1(t)}{\delta t} + \frac{\delta r_2(t)}{\delta t} = (c_1 \cdot \alpha_1 + c_2 \cdot \alpha_2) \cdot r(t) && (4.6)
\end{aligned}$$

With the (Karr like) intuitive definition for $\alpha_i = \frac{c_i}{c_1+c_2}$ (a mathematically correct solution would use $\alpha_1 = \frac{c_1}{c_1-c_2}$ and $\alpha_2 = \frac{c_2}{c_2-c_1}$, however, this would make one resource negative), the system results in a reduced system as follows:

$$\frac{\delta r(t)}{\delta t} = \frac{c_1^2 + c_2^2}{c_1 + c_2} \cdot r(t) \tag{4.7}$$

As is visible, the system in 4.7 doesn't exactly correspond to the initial system in equation 4.3. Figure 4.1 shows $c_1 + c_2$ (blue) and $\frac{c_1^2+c_2^2}{c_1+c_2}$ (yellow). As can be seen, the split system is hugely sensitive for certain values of c_1 and c_2. However, there are some restrictions on c_i, as mentioned in section 4.1.1 (resulting from the statement that variables are split when at any t: $\frac{\delta r}{\delta t} \leq -r(t)$):

$$\begin{aligned}
(c_1 + c_2) &\leq -1 && (4.8)\\
c_i &\leq 0 && (4.9)
\end{aligned}$$

Applying those restrictions to figure 4.1 shows that we are in a region where under-consumption happens, i.e. the split system will consume less resources than the combined system. Figure 4.2 shows the effect on the system of 4.3 (red) and 4.7 (blue) with an arbitrary initial resource availability of 50. An interesting fact is also that the under-consumption in the simple system is independent of the time step Δt. In reality, momentarily consumption depends on more than a constant, giving the time step again some influence depending on the system (usually the smaller the more accurate).

4.1.2.2 Tackling the Problem of Under-Consumption

In order to prevent under-consumption, every process could include a *correction factor* as follows:

$$\beta_i = \frac{c_1 + c_2}{c_i} \tag{4.10}$$

Thus the system becomes:

$$\frac{\delta r_1(t)}{\delta t} = c_1 \cdot \alpha_1 \cdot \beta_1 \cdot r(t) = c_1 \cdot r(t) \tag{4.11}$$

$$\frac{\delta r_2(t)}{\delta t} = c_2 \cdot \alpha_2 \cdot \beta_2 \cdot r(t) = c_2 \cdot r(t) \tag{4.12}$$

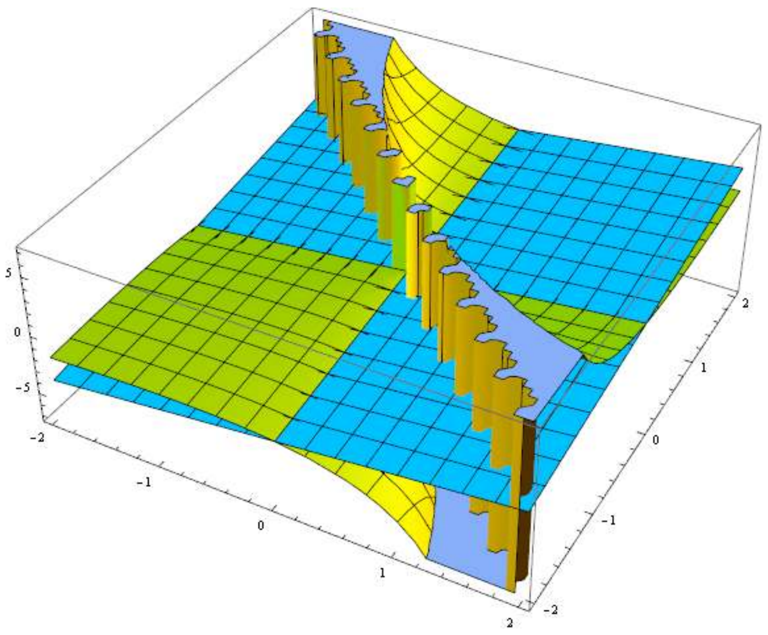

FIGURE 4.1: Effects of splitting state

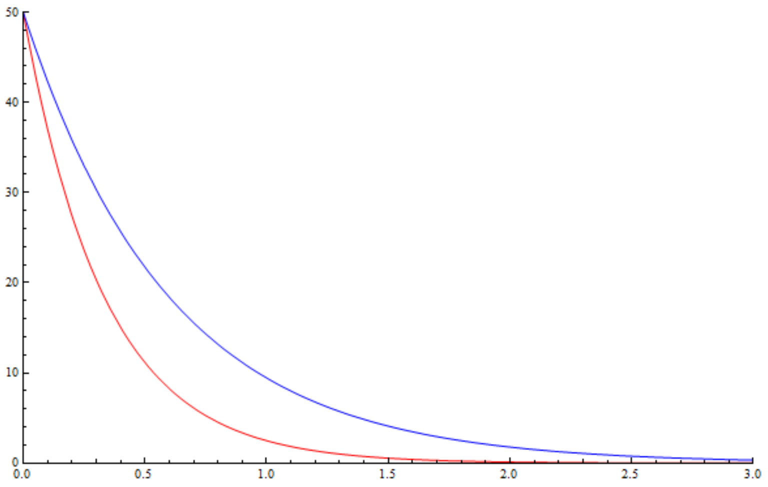

FIGURE 4.2: Effects of splitting state in the exemplary system

And the resulting system corresponds to the initial one. Basically, this is the strategy used in all *post-allocation* mechanisms, as every process gets the whole unmodified state. This, however, leads to so-called over-consumption for any $\Delta t > 0$. The reason for this is simply that processes only synchronize every Δt and thus in reality use up much more resources than the overall system should. The influence of the step size allows to bound this error though and is a major improvement as compared to the split system. Figures 4.3 and 4.4 show the difference of the systems. As is visible, a numerical solution of the pre-allocation system even gets further away from the real system with a smaller time step.

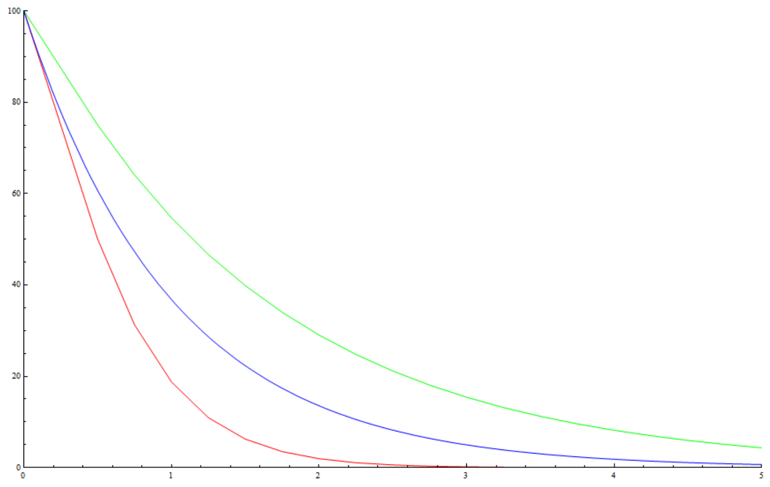

FIGURE 4.3: Effects of splitting resources ($\Delta t = 0.25$), blue is the original system, green the split (pre-allocation) one and red the post-allocation one

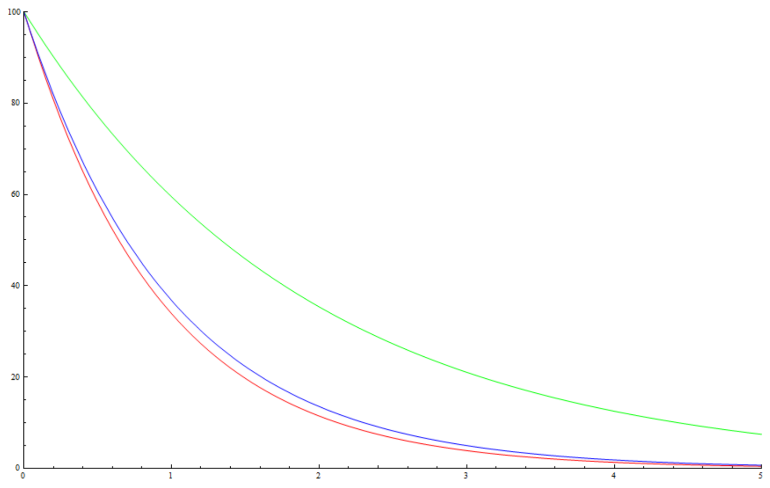

FIGURE 4.4: Effects of splitting resources ($\Delta t = 0.05$), blue is the original system, green the split (pre-allocation) one and red the post-allocation one

4.1.2.3 Generalizing Consumption Errors on Arbitrary Functions

A generalization on arbitrary black box systems is difficult. However, it can be stated that no matter what, a split system always only sees a part of the real state. At thus systems working on a split state will never behave like a monolithic system. In most of the cases under-consumption and under-production will occur. The lack of a overall compensation for this effect was a key factor in the development of post-allocation strategies in this thesis.

4.2 Domain Applicability of Modularized Simulations

This section discusses when it makes sense to have a modular simulation. It also looks at methods for automated modularization. There are two main criteria that need to be fulfilled in order to split a system into various modules:

1. The system must have parts that can be decoupled without too much interaction between modules. This means modules shouldn't access the same variables or show high sensitivity in common variables.
2. The merging of changes (the whole simulator logic) has to take noticeably less time than the simulation time of independent modules.

4.2.1 Non-Interacting Systems

As the splitting of systems introduces mathematical errors in the system (see also section 4.1), the system should be split along a line that minimizes the module interdependencies. Often, the splitting is given by the modeler, which might not be optimal. This doesn't mean the technique should not be applied, however it will usually lead to smaller time steps as the errors get too big otherwise.

In order to give some mathematical backing on what a good splitting of a system can be, this section presents a crude method to modularize processes. First, let's introduce the sensitivity matrix (equal to the *Jacobian* matrix for systems of ODEs):

$$J_k = \begin{bmatrix} \dfrac{\partial C_1}{\partial s_1} & \cdots & \dfrac{\partial C_1}{\partial s_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial C_n}{\partial s_1} & \cdots & \dfrac{\partial C_n}{\partial s_n} \end{bmatrix} \tag{4.13}$$

This matrix is defined for all processes ($k \in 1, \ldots, p$) and calculated by performing a sensitivity analysis on the system. The analysis can for example be performed by fixing all but one variables and inspecting the response of the system to changes of the free variable. By summing up the absolute values in columns of J_k we get the process' sensitivity dependency on a single resource. This yields a row vector of dependencies:

$$Dep_k = \begin{bmatrix} d_{s_1} & \cdots & d_{s_n} \end{bmatrix} \tag{4.14}$$

Doing this for all processes and putting the resulting vectors in a matrix results in the dependency matrix P_{dep} ($p \times n$):

$$P_{dep} = \begin{bmatrix} d_{1,s_1} & \cdots & d_{1,s_n} \\ \vdots & \ddots & \vdots \\ d_{p,s_1} & \cdots & d_{p,s_n} \end{bmatrix} \tag{4.15}$$

Starting from the J_k matrix again, we now sum up the absolute values of the rows, this yields the influence vector of a process:

$$Infl_k = \begin{bmatrix} i_{s_1} \\ \vdots \\ i_{s_n} \end{bmatrix} \tag{4.16}$$

Putting all the influence vectors in a matrix results in the influence matrix P_{infl} ($n \times p$):

$$P_{infl} = \begin{bmatrix} i_{1,s_1} & \cdots & i_{p,s_1} \\ \vdots & \ddots & \vdots \\ i_{1,s_n} & \cdots & i_{p,s_n} \end{bmatrix} \tag{4.17}$$

The above two matrices describe how different processes influence and depend on variables. Multiplying them will give the inter-process dependency matrix that denotes how much a process depends on another process. This can be similarly done for single variables by filling matrices P_{dep} and P_{infl} not with the sums as above but with a single variable only. However, here we only look at the complete state dependencies (for further reference look at [25], which presents a discussion originated from this thesis).

The clustering matrix P_{clus} is defined as follows:

$$P_{clus} = P_{dep} \cdot P_{infl} = \begin{bmatrix} c_{1,1} & \cdots & c_{1,p} \\ \vdots & \ddots & \vdots \\ c_{p,1} & \cdots & c_{p,p} \end{bmatrix} \tag{4.18}$$

In the clustering matrix $c_{1,4}$ would be read as "Amount with which process 1 depends on process 4". An optimal splitting of a system consists of modules that lead to a clustering matrix which has elements only on its diagonal. For obvious reasons this would mean that the modules are completely independent, which is usually not feasible or desired, but it is already good if the clustering matrix has large elements on its diagonal and small ones in the rest.

The clustering matrix also allows discovery of highly shared resources, if created for every single variable. Resources are discovered by searching for non-diagonal elements with large values. Once found, a resource might be shared using one of the methods described in chapter 3.

The clustering matrix also allows to optimize communication times. This can be done by applying a clustering algorithm to the matrix, which will directly show how fast different processes have to communicate (in terms of unit time, which has to be determined by other means, e.g. error analysis). As an example, the following system is studied:

$$X'(t) = \begin{bmatrix} -1 & 2 & 1 \\ -1 & 0 & 1 \\ 1 & -3 & -1 \end{bmatrix} \cdot X(t) \qquad X(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} \tag{4.19}$$

Assume a modeler specifies the above system with the following four processes (with $X(t) = X_1(t) + X_2(t) + X_3(t) + X_4(t)$ the state vector):

$$X_1'(t) = \begin{bmatrix} 0.5 & 1 & 0 \\ 0 & 0 & -0.5 \\ 1 & -1 & 0 \end{bmatrix} \cdot X_1(t) \qquad X_2'(t) = \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \cdot X_2(t) \tag{4.20}$$

$$X_3'(t) = \begin{bmatrix} -1.5 & 0 & 1 \\ 0 & 0 & 0.5 \\ 0 & 0 & -1 \end{bmatrix} \cdot X_3(t) \qquad X_4'(t) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \cdot X_4(t) \tag{4.21}$$

Applying the above discussed method leads to a clustering matrix as follows:

$$P_{clus} = P_{dep} \cdot P_{infl} = \begin{bmatrix} 4.25 & 2.5 & 5.25 & 4 \\ 2 & 1 & 3 & 2 \\ 7.25 & 2.5 & 6.25 & 4 \\ 3 & 3 & 2 & 3 \end{bmatrix} \tag{4.22}$$

Figure 4.5 shows the resulting dendrogram (which comes from the clustering method, in this case calculated using Mathematica's *DirectAgglomerate*). It's easy to see that processes 1 and 3 have to communicate the most. Their results then need to be shared with process 4 and finally process 2. This can be used to determine synchronization points for different processes.

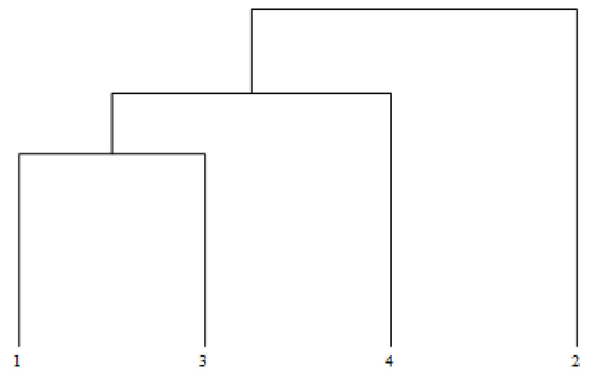

FIGURE 4.5: A dendrogram showing the modularization of a sample system

4.2.2 Heavy-Load Processes

A splitting of a system (from the computational point of view) only makes sense if the gain from parallelization is higher than the price to pay for the process communication and synchronization. The formulas for the two cases of distributed and non-distributed computation are:

$$t_{non-distr} = n_{steps} \cdot \sum_k t_{c,k} \tag{4.23}$$

$$t_{distr} = n_{steps} \cdot (max_k(t_{c,k}) + t_m) \tag{4.24}$$

Where n_{steps} is the number of steps for which the simulation runs, $t_{c,k}$ is the computation time of process k and t_m is the time needed for communication and synchronization. Figure 4.6 shows a plot of the above formulas with values from the Karr model (see also appendix A) and $n_{steps} = 100$. In the Karr model, a single step of one simulation second takes about one physical second ($\sum_k t_{c,k} = 1$), with most time needed by the replication process ($max_k(t_{c,k}) = t_{c,replication} = 0.31$). As can be seen, as long as the communication and synchronization time t_m is below 0.7 seconds, the distributed version is faster than the sequential.

For further analysis, dissection of t_m is necessary. The time needed for communication and synchronization depends heavily on the algorithm used. Making some assumptions allows to do general reasoning, finer algorithms can lead to better results for distributed algorithms though. We assume all processes always have to synchronize on all variables. Synchronization time is assumed linear in the number of variables (as once the simulator received all variables, it can be seen as summing up every one of them). In a first analysis, we assume no violations and time step adaptions (actually both are a way to decrease simulation errors, and thus mostly independent of the distinction of distributed vs non-distributed).

Every time step, all processes have to receive and send all their variables to the central authority: $2 \cdot p \cdot n \cdot t_{send}$. In addition, the simulator has to aggregate all variables: $n \cdot t_{aggr}$. This results in the following total overhead (as the simulator somewhat is a bottleneck that only accepts sequential input of messages):

$$t_m = 2 \cdot p \cdot n \cdot t_{send} + n \cdot t_{aggr} \tag{4.25}$$

With the concrete message passing implementation, there are some optimizations that happen. First, only one message containing all the variables needs to be sent. This message is obviously larger, but the overhead of n single messages vanishes. Second, processes actually only send and receive updates for changed and needed variables. Third, the aggregation is trivial to parallelize for most algorithms. Further, t_{send} depends on the actual network structure used. In this analysis, the focus lies on multicore processors (messages don't have to pass over a network).

For a system containing 2500 variables $2 \cdot n \cdot t_{send}$ evaluated to 22 ns on an Intel Core2 Duo (T9400) CPU with 4 GB of RAM. The aggregation of 2500 variables from 28 processes took 93 ns ($= n \cdot t_aggr$). The total expression evaluates to:

$$t_m = 28 \cdot 22ns + 93ns = 709ns \tag{4.26}$$

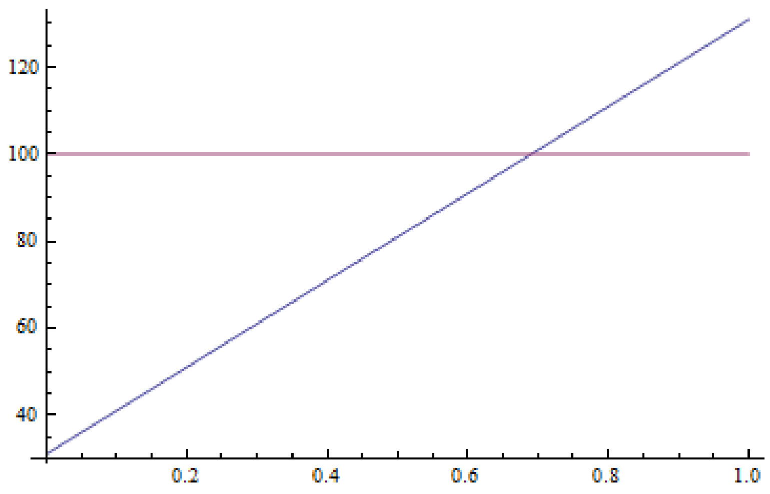

FIGURE 4.6: Total time for non-distributed (violet) and distributed (blue) simulations as function of communication overhead t_m in the Karr model

In the real implementation there are other factors, like the time needed to copy variables, the overhead introduced by the *change method* (of letting processes post desired changes on the system), the memory needed for keeping copies of the state, time needed to communicate with external programs (e.g. a Matlab simulation) and more. They would clearly increase t_m, however, they heavily depend on the implementation. Still, as the final results show, their effects are also negligible compared to processes that individually take up to 0.3 seconds for execution (at least for the implemented strategies and tested systems).

4.3 Effective Dynamics / Time Step Adaption

As the whole splitting mechanism basically is a numerical solving strategy for large systems, a lot of proven methods can be used to calculate time steps (though sometimes they have to be adapted for the distributed system).

4.3.1 Single Process Time Step Adaption

Depending on the underlying modeling technique, time steps can be calculated for every process independently. An exemplary method for ODEs solved with the Euler method is described in appendix A (see also [26] for other and extended adaption techniques). It calculates the function value at $x(t + \Delta t)$ two times, by using Δt directly and by applying $\frac{\Delta t}{2}$ two consecutive times. The next step size is then determined via:

$$d = x(t + \Delta t)_2 - x(t + \Delta t)_1 \tag{4.27}$$

$$\Delta t_{i+1} = 0.9 \cdot \Delta t_i \cdot \frac{\epsilon}{|d|} \tag{4.28}$$

This ensures the local error will be bounded by ϵ. 0.9 is a safety constant to make sure the next step is successful. Similar techniques exist for other numerical methods. For stochastic processes for example, the duration until the next event can be calculated and used as Δt.

However, an important part that cannot be neglected is the dependency of different processes on each other. This dependency will have an influence of the maximal time step given some error bound.

4.3.2 Multiple Process Time Step Adaption

Even though single processes can be controlled via local time step adaption, that doesn't ensure the global error stays within the ϵ bounds. The good news is that with the techniques developed in this thesis, the error estimation and time step adaption can be done in a modularized system just like in a monolithic one.

A simple idea is to let the simulator compute results for every module for time steps as above (h and 2 consecutive times $h/2$) and do the same error analysis. This results in a global time step that is applicable for all processes and proves the solution to stay within some error. However, the time step is primarily influenced by the process requiring the smallest time step. In a large system, such a global time step enforces too small time steps for many processes.

Thus, it's better to use the above results of the clustering matrix (see 4.2.1) to calculate the different interdependent time steps of the modules. What the clustering matrix basically tells is that for two close processes (they need to communicate a lot, as they depend heavily on each other), their time steps have to be close together as well (if the dependency is one-directional, the "master process" forces the dependent process time step). We can thus bound the time steps of processes by processes they depend upon and processes they influence. A process' time step can maximally be a factor or fraction of the distance of the processes in the clustering matrix. Thus given any two processes, the one with the smaller time step forces the other to stay within a bound as given by the clustering matrix.

Taken all into account, this also leads for a global strategy to determine time steps, given some smallest time step of a given process. However, this only upper bounds the time steps of all the processes and still allows for some freedom.

4.3.3 Violations

System violations provide a further technique that can be used to adapt time steps. This time step adaption isn't based on mathematical errors but rather model violations. Violations are discussed in section 3.2.1. The method is simple: As soon as a violation in the system is detected, re-run involved processes with a smaller time step, namely the time step until the violation has happened.

Let's denote the time of the first violation happening with t_v and the set of involved processes with $Violators$. All the $Violators$ have associated t_{prev}, t_{curr}, t_{next} and Δt (they denote the times of the previous calculated step, from t_{prev} to t_{curr}, the times of the current step, from t_{curr} to t_{next} and the process' time step). Processes now have to recalculate their time steps as follows:

$$\Delta t_i = t_v - t_{prev} \tag{4.29}$$

They also have to update the respective times t_{prev}, t_{curr} and t_{next}:

$$\begin{aligned} t_{prev} &= t_{prev} && (4.30) \\ t_{curr} &= t_{prev} && (4.31) \\ t_{next} &= t_{curr} + \Delta t_i && (4.32) \end{aligned}$$

This works for any strategy that ensures that no process can cause violations before t_{prev} of any process, i.e. processes have to wait with their calculations until all processes' current times t_{curr} are past their own t_{prev} (at which time the process has the highest priority as it would cause all other processes to wait for its completion).

4.4 Sensitivity

As processes in general are black boxes, sensitivity analysis follows previously researched methods (for examples see [27] or [26]). Namely all variables but one are fixed in the input state vector of a process. The one free variable is now modified (e.g. by applying a step function) and all changes on the system recorded. This leads to the sensitivity or dependency matrix as described in section 4.2.1. Of course, if the implementations of the underlying processes are known, other sensitivity analysis methods can be used, like the *Jacobian* for systems of ODEs.

As systems in general don't have to be linear but can be of arbitrary complexity, for maximal accuracy the sensitivity analysis would have to be repeated every time step, forcing a recalculation of the whole clustering and step sizes. Depending on the underlying model it's enough to do the analysis only every now and again, as processes (e.g. in biological systems) often can be approximated as linear systems on small time intervals.

The sensitivity can be used to modularize the system, but also as a general way to characterize and argue about the model.

4.5 Efficiency

As some of the discussed techniques allow to lower bound the time steps of the overall model (e.g. the *change merging* or *smash fit* strategy by merging changes discarding impossible choices, or by randomly integrating changes), an upper bound on execution time of the simulation can be evaluated. There is a trade-off with accuracy if the system is bounded in such a way, this can be used to get a quick overview of the model though.

The simulator could be used like this for real-time applications without any need for absolute correctness, or where a trade-off can be made. An exemplary application would be computer games.

Chapter 5

An Implementation of the Integrated Module Simulator

5.1 Defining the Simulation in an Easy Way

An issue with whole cell models (and any model applicable to modularization) in general is their huge complexity. Those models consist of a state containing several thousand variables having tens or more processes acting on them. If the architecture is not carefully designed, dependencies will exist in the system, making the whole model rigid and hard to adapt. As an example, imagine a system where each part can depend on arbitrary other parts of the system (e.g. a process can depend on all states, but also on other processes). In this system, the change of a single property in a state will make updating all other processes necessary.

An often applied method in this case is the definition of well specified interfaces, i.e. processes have input and output ports that accept and yield values according to a specification. However, as was described above, processes in the concept of this thesis only produce a change on the system and don't have a state themselves. Thus they need to be called from the simulator directly. The benefit is having less dependencies and a simpler program structure.

Of further interest is compile time or static type checking, as this allows for model verification (to a minor degree). This is valuable as it makes some of the unit testing dispensable and gives the modeler instantaneous feedback on types used for various properties.

This section tackles the issues mentioned by providing the base for a framework that is modularized with only marginal dependencies between modules. First, the Karr model is examined. In short, it suffers from the above described complexity issues, and in addition uses a verbose programming style plus Matlab-typical complex matrix accesses.

5.1.1 The Complexity of the Karr Whole Cell Model

The Karr model is written in Matlab, splitting up functionality into processes and states using a simple object oriented approach (for an extensive explanation of object orientation see [28]). Process implementations derive from a common `Process` superclass (or a subclass of it called `ReactionProcess`) and states derive from a common `State` superclass. Those classes themselves are written in a excessively complex way which makes modification difficult. They inject basic functionality into their subclasses, but force all subclasses to still implement some basic things like initialization of constants or copying from and to state. The main reason the simulation seems hugely complex is lots of redundancy (sometimes enforced by the way Matlab is designed, sometimes seemingly by choice), lots of different ways to access the same elements, nested access into matrices and Matlab structures and enormously long names. The latter isn't necessarily bad, however, the code looks very complicated and verbose like this.

Most of the functions are implemented on top of multi-dimensional matrices, which are with high performance in Matlab. A state matrix usually has the dimension

$$n_{vars} \times n_{compartments} \tag{5.1}$$

Where the number of compartments is six and the number of variables ranges from a couple tens to about 700. A state can contain several of those matrices, describing various aspects. An example would be enzymes in different states (as found in the protein monomer and complex states). A logger object extends and stores this as a matrix of size

$$n_{vars} \times n_{compartments} \times n_{timesteps} \tag{5.2}$$

This allows to have the whole simulation information stored in a central matrix. As one can imagine this matrix is incredibly large, but useful for analysis of simulation output.

States now select variables they're interested by denoting them in several predefined fields (table 5.1 shows an overview of those fields) which are accessed by the simulator object to copy state information into the processes and vice versa. States and their properties can also be referenced directly in order to introduce changes on them. This already provides several ways to access things.

The whole system gets more complex, as indices change during the copying from states to processes (global to local), and parts of matrices often have to be extracted by reshaping

`stimuliWholeCellModelIDs`	Selects stimuli by their ID, a stimulus is a variable coming from the external environment (currently not used by any process).
`substrateWholeCellModelIDs`	Selects substrates by their ID (metabolites, e.g. H2O, H, MET, FOR).
`enzymeWholeCellModelIDs`	Selects enzymes by their ID (e.g. MG_106_DIMER, MG_172_MONOMER).

Table 5.1: Stimuli, substrate and enzyme variables to be copied from state

or selective access. There is also no notion of static typechecking and the correctness of processes has to be validated by unit tests (variables within the simulation are usually addressed by a string or int key, often both at the same time).

An important thing to note is that the simulation, even though everything is serial and states could be (and are sometimes) modified directly, there is also lots of copying to and from states and processes. There are also side effects which provide an additional mechanism to introduce change on the system. All together - lots and lots of ways to do things, lots of redundancy in the code, no compile (or any time) checking and verbose description.

In the following model description, the system is changed and stripped so that users only see the relevant parts of it and don't have to think about the simulator logic at all. In fact, simulator logic can easily be exchanged for the same model (letting people work on the simulator independent of the modelers, allowing to simulate models with different accuracy and more).

5.1.2 An Easier Model Description

In order to make collaborative work possible with people from different fields and different expertise in software like Matlab, it is important to have a very transparent and simple model description. The model description developed here is loosely based on the Karr model - making some drastic simplifications in the way users have to specify things. It is important to see that the simulator simulating the model can easily be exchanged with the language developed here (as in contrary to the Karr model for example, where the simulator and model are tightly linked).

There are three main components: a state, various processes and a core model. Often, modelers will only modify processes and the core model, as the state usually stays the same over different simulations.

5.1.2.1 State

The state or state vector contains all variables within the simulation. The variables can be split up into states, however, states are purely for namespacing purposes and easier access by the modeler. State variables are defined by a simple instruction:

```
variableName = field[optionalType](value, optionalDesc) optionalRestrictions
```

The variable name is extracted from the above statement via reflection at compile time. If the value doesn't already infer the type, the type of the field can be specified by supplying it directly to the function. A description for the variable can be given within the code, this is valuable for user interfaces. The optional restrictions include variable bounds and other restrictions (e.g. ≥ 0).

An advantage of defining the variables that way is that the simulator can choose how (if even) to split up the state variables. This allows for optimization strategies in distributed systems as the simulator can determine (either stationary or whilst running) which processes access which state variables and optimize for minimal message passing. It can for example assign variables that are only used by certain processes directly to the processes, so that no message has to be passed ever.

5.1.2.2 Processes

A process defines only an evolve state method that produces one or multiple changes on the system (the reason for the notion of change is that it allows a process to specify multiple changes). The evolve state method gets the state vector injected, as well as the starting time and the step size. This functional approach allows processes to run multiple times in parallel (e.g. for error estimation, error correction or simply in accordance with the algorithms described earlier). An exemplary process definition thus looks like:

```
class MyProcess extends Process {
  function evolve(state: StateVector, t: Double, dt: Double)
    returns List[Change] {
    ...
  }
}
```

The process thus has complete access to all properties in the system, but as the simulator automatically determines which variables are required by which processes, this still is highly optimized.

5.1.2.3 Core Model

The core model defines everything that is model specific and is not stored in either the states or processes. This includes the selection of processes participating in the simulation, special observables (the whole state information is available at the end of the simulation, but observables are shortcuts that are immediately displayed to the modeler), a method that calculates dependent variables within the state, some meta information and it selects the simulation strategy to run the simulation. It's outline looks like this:

```
class MyCoreModel extends CoreModel {
  title = "My New Bio-Model"
  description = "We try to show new things with it."
  authors = "John Smith"

  stateVector = new StateVector
  processes = new Array(new MyProcess1, new MyProcess2, ...)

  simulationStrategy = new SomeStrategy(MAX_TIME, DELTA_TIME,
    MIN_DELTA_TIME, MAX_DELTA_TIME)
  observables = List(stateVector.ATP, stateVector.H2O)

  function calcDependencies(state: StateVector) { ... }
}
```

Meta Information Meta information is primarily used for display and interaction within the user (either over the web interface, or some other means). Meta information includes a title and description of the model, the authors and contributors. The intention of meta information is to build a platform for collaboration at some point. Within the platform, modelers can select different models and processes and use them as building blocks to create new models or embed existing ones.

Constants Constants are defined within a Constants class in the model that can be instantiated and accessed by all processes and states alike. It is important to see that the constants are not shared between processes, as those can run in completely different environments, where no sharing is possible at all.

Functions General functions are defined within a Functions class in the model that can be instantiated and accessed by all processes and states alike. The same holds as for constants.

Observables Observables are variables or functions of variables that are presented to the user during the simulation (as in contrast to the whole simulation state vector at all times that is available at the end). Usually this consists of a plot of some resource availability, the state of some object or something similar. The simulator will take care of the observable and present it in an appropriate way.

5.1.2.4 Simulator

The simulator defines everything else apart from the model - different strategies to simulate the model, how outputs are collected and stored, how states and processes are linked, where different state variables reside, and so on. The simulator does all message passing and variable handling in the background, so that a modeler can focus only on the modeling problem. Figure 5.1 shows the simulator architecture.

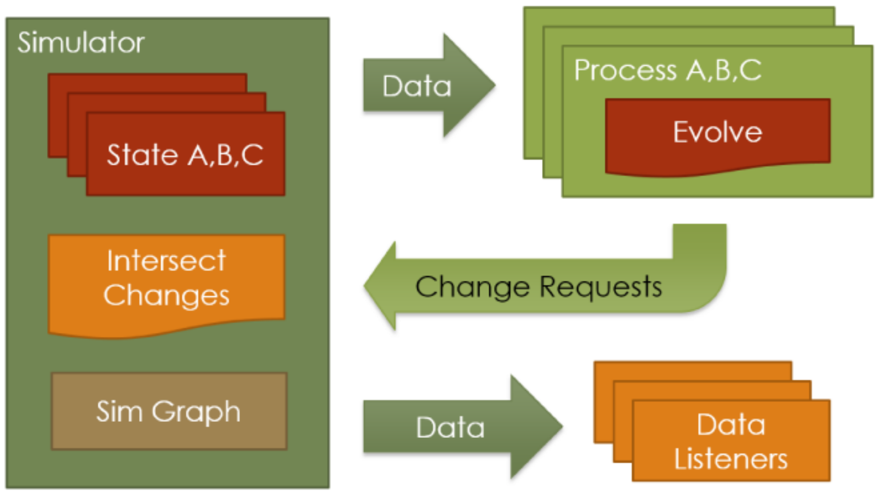

FIGURE 5.1: The simulator architecture

Much of it happens in the so-called simulation strategy, as it is completely interchangeable for the same model description. This has the advantage that the simulator can plug different simulation strategies with different logging mechanisms and models.

Simulation Strategies The notion of a simulation strategy is introduced because a simulator can run a model under various different strategies (comparable to Euler, Runge-Kutta and similar when solving ODEs). Different algorithms are discussed in chapter 3. Different strategies can focus on simulation speed, accuracy of output or different other properties. A noteworthy point is that the model description is completely independent of the simulation strategy, i.e. a single model can be simulated using arbitrary simulation mechanisms. The advantage of such a system is that modelers are independent of computer scientists who develop the underlying simulation strategies.

Logging Systems Logging of state variables can be chosen with various degrees of detail, storage to disk, display in a user interface and more. In the model specification so called observables are defined. Those observables give immediate feedback to the user and are independent of the underlying logging strategy.

5.2 An Implementation in Scala

Scala is an advanced programming language developed at the EPF Lausanne [18]. Amongst others, it offers a very concise and minimal syntax, a primarily functional paradigm (with all the object oriented concepts still available), a strong community and integration (as it runs on the Java virtual machine) and an easy way to specify domain specific languages.

It was chosen for the simulator implementation because of its widespread use in the scientific community, the availability of various libraries (as all Java libraries can be used without limitations), it's minimalistic syntax, strong type checking and easy adoption to domain specific languages.

The framework developed with this thesis can be found on Github under:

https://github.com/dominikbucher/moi-sim.

As the project is quite large, there were some libraries used to speed up development.

5.2.1 Used Libraries

The library used for *message passing* is the one developed by Typesafe under the name Akka [29]. It is now part of the official Scala distribution but has only recently been introduced. A message passing framework implements the message passing paradigm (see e.g. [30] for more details about message passing). This paradigm describes a system in which processes run on distributed computers, communicating only by messages. However, in the Akka implementation those messages don't have to be strings but can be objects of arbitrary complexity. The message passing paradigm allows for easier concurrency arguments and proofs, as all processes can be analyzed separately.

In the simulation, the passed messages are instructions to calculate some state change given a state vector and a time, and the corresponding results:

```
/** Evolves system based on a process.evolve method. */
case class Evolve[T <: StormState[_]](state: T, t: Double,
    dt: Double) extends Message
/** Results from a process.evolve method. */
case class Result[T <: StormState[_]](chgs: List[StormChange], t: Double,
    dt: Double) extends Message
```

As sending the whole state can become an overhead, the concrete implementation only sends the variable needed by the processes. This selection is done by registering all used variables when a process runs and later on only communicating these variables.

A further noteworthy library is the Efficient Java Matrix Library (EJML) [31] which in turn relies on some low-level libraries for math. The EJML allows *matrix calculations* and the solving of *optimization problems* similar to Matlab [11] or Octave [32].

There are also some standard libraries in place to facilitate file input and output, plotting, logging and some additional math operations. Those libraries are Colt [33], jOptimizer [34] and s4gnuplot [35]. Colt is a set of open source libraries for high performance scientific and technical computing in Java. jOptimizer is an open source library that addresses the solution of a minimization problem with equality and inequality constraints. s4gnuplot finally provides a programmatic access to the gnuplot [36] program which is used to plot observables.

The web interface is built on the Scalatra [37] library, a very light *web framework*. It combines existing Java enterprise architecture with the brevity of Scala, making it possible to build high-performance web sites and apis very quickly.

5.2.2 States

States are defined by extension from the `State` superclass. This state class has a self type that defines a `copy` method (a self type in Scala defines what form subclasses need to have, often mentioned alongside it is the *cake pattern*, see also [38]). This copy method must implement an algorithm that returns a copy of the state at any time. The variable values don't necessarily have to be copied, but their pointers have to point to new objects. In Scala, the use of `case` classes facilitates this process, as `case` classes already define their copy method. The state class then uses this `copy` method and provides the additional copy logic to copy values on top of it via the `dupl` method.

The state also defines three maps:

- The *fields* map is a map between a field id and its value. This could alternatively be implemented as an array, as field ids are incremental starting from 0.
- The *field pointers* map is a map between a field id and the field itself (as the field defines additional properties like an id, name and description.
- The *field names* map maps ids to field names and is mostly used for user interfaces.

New fields are simply created via the following statement:

```
val ATP = field(ATP_INIT) <restrictions>
```

This will automatically create a field, give it a unique id within the state, extract the name from the variable and apply the restrictions as specified.

A third responsibility of states is to collect all changes happening. This is done by keeping *change values* for all fields in a map. The map can be reset via `resetChanges` and collected by calling `collectChanges`. Usually it is reset at the beginning of a process' evolve method, and collected in the end.

5.2.2.1 Fields

A field contains any value from the state vector. The advantage of defining everything as a field is that this allows for automatic recording of changes and centralized definition of administrative functions like update, merge and field flagging. In particular Scala allows to define an `update` and an `apply` method that will automatically be called on an object from the following two statements:

```
ATP() = 5.0          --> Calls update and updates the value of ATP
val x = ATP() + 2.0  --> Calls apply and returns the value of ATP
```

This allows for a model syntax almost as if using the variables directly, however, in the background all administrative tasks can be performed. Those tasks primariliy consist of recording changes whenever `update` is called. Also, the fields class defines a `merge` method that takes any change, integrates it into the field and checks for any violations.

5.2.2.2 A Mapping of Variables to Arrays

In order to improve performance, variables in the state vector are automatically mapped to an array (or a map). For the user it looks as if he'd be manipulating variables directly, in reality he's manipulating an array of fields. Optimization primarily happens because arrays allow for faster copying, faster field access and less overhead when sending as messages.

This mapping happens by automatically assigning an id to every field. Further accesses happen either via the field object (which doesn't have to be reinstantiated or overwritten at any time, as the underlying value is stored in the array) or via access by id on the array directly.

5.2.3 Processes

The `Process` superclass very marginally defines a process outline. A process must be given a name and it gets an id automatically assigned (again, processes and their respective actors are stored in an array). The `evolve` method describes this process' effect on the state. In addition, the `Process` class defines some administrative logic, namely clearing all changes of the state vector and collecting them again after `evolve` was executed.

5.2.3.1 Gathering Changes

Change collection is rather easy as the state vector records all changes anyways. Processes have to call the `resetChanges` method on the state before executing `evolve`, and `collectChanges` afterwards.

5.2.4 Model

In addition to the information mentioned in 5.1.2.3, the Scala implementation defines a method `addProcess` (or as a shortcut `++`). This method expects a so called creator function, which is a function without arguments that returns a new process. It also automatically assigns a unique id to the process. Both those steps ensure that the simulator has complete control over process creation and can spawn as many as necessary. The id gives a possibility to access all processes of the same type by id.

5.2.5 Strategies

Strategies only define the `simulate` method, which takes an input model and calculates an output trajectory of the state vector. The output trajectory is stored in a `TreeMap`, a sorted map, in this case by time (i.e. at every time point there is a vector describing the state). The returned map thus has the following form:

$$\begin{bmatrix} t_0 & \rightarrow & [s_0, s_1, ..., s_n]_0 \\ t_1 & \rightarrow & [s_0, s_1, ..., s_n]_1 \\ & \vdots & \\ t_{maxTime} & \rightarrow & [s_0, s_1, ..., s_n]_{maxTime} \end{bmatrix} \tag{5.3}$$

This map can be further analyzed and is also stored to disk. The analysis can be done directly in Scala, as the map is returned from the simulator's `runSim` method.

5.2.5.1 Integrating Changes

The implementation defines a trait `ChangeHelper` that helps integrating changes in different strategies. As many strategies face the same issues of how to integrate different changes on different time scales, this trait defines this common functionality. The function `intersect` slices the different changes (see section 3.4 for an explanation what slicing means) and gradually integrates all changes into the state vector. If a violation is detected, the state is reset to the last non-violating state and the slice (beginning and end time) together with the violating processes are returned.

If less functionality is required, the methods `tryMerge` and `slices` offer functionality to merge a set of changes given a time interval or to slice up a set of changes into smaller non-overlapping slices.

5.2.5.2 Detecting Violators

Violators are detected in the `tryMerge` method, where all changes are applied to the state for the respective time interval (recall that changes are assumed linear on their scale - comparable to many a numerical solving method). If at the end of the integration process a violation is detected, all involved processes are returned.

5.2.5.3 Stepping Back

A step back algorithm is implemented only for change integration, at the slice level. If a strategy wants to provide functionality to step back further in time (e.g. to correct errors, integrate more changes that weren't available before etc.), it has to implement that itself. It's not overly complicated though, as the state vector can easily be copied and stored in another map. Some of the strategies implemented for this thesis do this and can serve as an example for further implementations.

5.2.6 The Simulator

The `Simulator` class finally wraps all the above components into an executable environment. It mainly defines the simulation and logging strategy and the model. For the Scala implementation it adds an Akka actor system that handles all message passing within the simulation. It defines the `runSim` method that runs the simulation asynchronously and returns a future (a deferred result that can either be waited upon or lead to execution of some arbitrary code as soon as it's available). The simulator also automatically logs all data and plots the observables as specified in the model.

5.3 An Environment to Facilitate the Development of Whole Cell Models

Inspired by the Karr model environment, this section describes the whole cell web interface that allows to plug and play with whole cell models. The long time goal is to have a platform for biological whole-cell models where people can plug their own models in different host circuits and test for various effects when interacting with the host. As for now, the web interface provides a user interface to choose and set parameters, run simulations and display their results.

The web interface is implemented using Scalatra [37], a lightweight web framework. On top of that it uses Atmosphere [39], a library that uses WebSockets [40] for server side pushes to the browser. This allows the simulation results to be pushed to the browser and to be updated there in real time.

As the whole simulator is implemented asynchronously, the server backend can spawn simulators as needed and update the user interface whenever results are available. To control the simulation from the browser, simple JSON messages can be sent to the server. Commands are put into the *type* field and can take the following values (additional fields are given below the corresponding command):

```
"type" : "RequestSimInfo"          --> Requests graph of specified simulation
             "simName" : Name of simulation
         "StartSimulation"         --> Starts the simulation
         "ResetSimulation"         --> Resets the simulation
         "UpdateParam"             --> Updates a parameter in the simulation
             "param" : Name of parameter
             "value" : New value of parameter
         "RegisterDataListener"    --> Registers an observable
             "dataPoint" : Name of observable
         "UnRegisterDataListener" --> Unregisters an observable
             "dataPoint" : Name of observable
```

The commands are then relayed to the simulator and invoke the respective actions there. The user interface frontend is written in JavaScript, basically sending the above JSON values on user events, receiving the responses from the server and updating the HTML accordingly. Figure 5.2 shows a screenshot of the web interface.

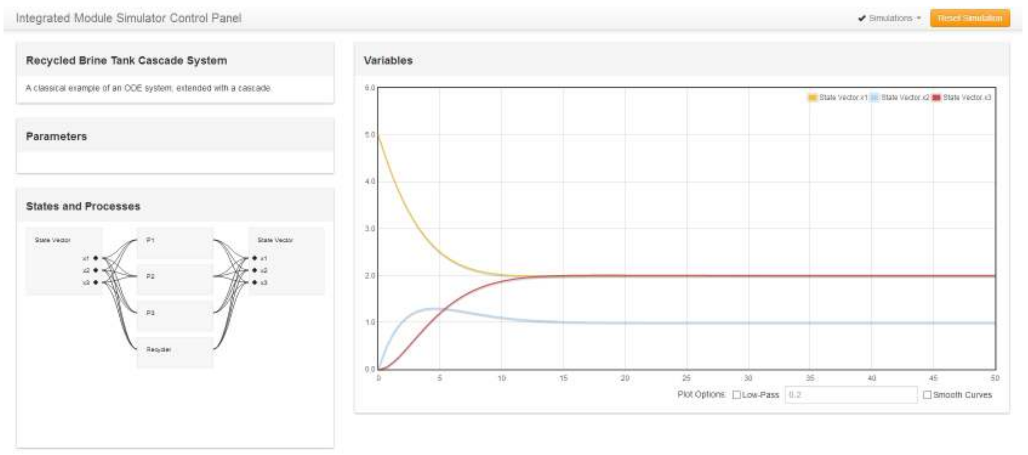

FIGURE 5.2: A screen shot of the web interface for MOI-Sim

Chapter 6

Integrated Module Simulations

6.1 A Brine Tank Cascade System

As the simulator also allows to solve arbitrary systems, the first object of study is a system of ordinary differential equations. The system is a standard system of study called the brine tank cascade, see [41] for example. There are three tanks of different sizes, where water flows from one to two and from two to three. The following equations describe the flow of water in the system (where x_i denotes the amount of water in tank i):

$$x_1'(t) = -\frac{1}{2}x_1(t) \tag{6.1}$$

$$x_2'(t) = \frac{1}{2}x_1(t) - \frac{1}{4}x_2(t) \tag{6.2}$$

$$x_3'(t) = \frac{1}{4}x_2(t) - \frac{1}{6}x_3(t) \tag{6.3}$$

As the system is quite simple, any strategy can be used (e.g. there are no violations that can happen as there are no restriction, so the simulation strategy doesn't even have to take that into account). The system is starts with the following initial values:

$$x_1(0) = 5 \tag{6.4}$$

$$x_2(0) = 0 \tag{6.5}$$

$$x_3(0) = 0 \tag{6.6}$$

The exact solution is as follows:

$$x_1(t) = 5 \cdot e^{-t/2} \tag{6.7}$$

$$x_2(t) = -10 \cdot e^{-t/2} + 10 \cdot e^{-t/4} \tag{6.8}$$

$$x_3(t) = \frac{15}{2} \cdot e^{-t/2} - 30 \cdot e^{-t/4} + \frac{45}{2} \cdot e^{-t/6} \tag{6.9}$$

Figure 6.1 shows a plot of the functions calculated with the Module Integration Simulator (MOI-Sim) developed in this thesis with a Δt of 0.1. Figure 6.2 is the same system simulated with Mathematica. The MOI-Sim version runs distributed on three processes. Of course, for a small system like this that doesn't make much sense and is just for showing of capabilities of the simulation engine.

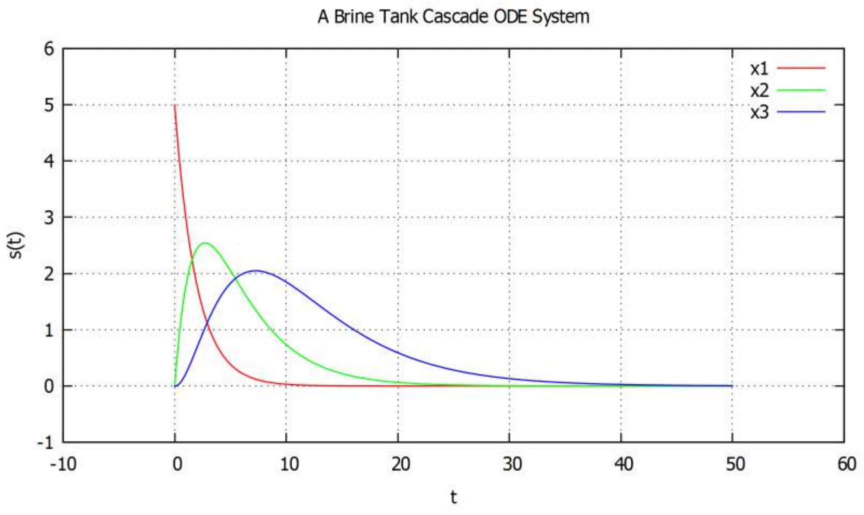

FIGURE 6.1: The exemplary brine tank cascade system

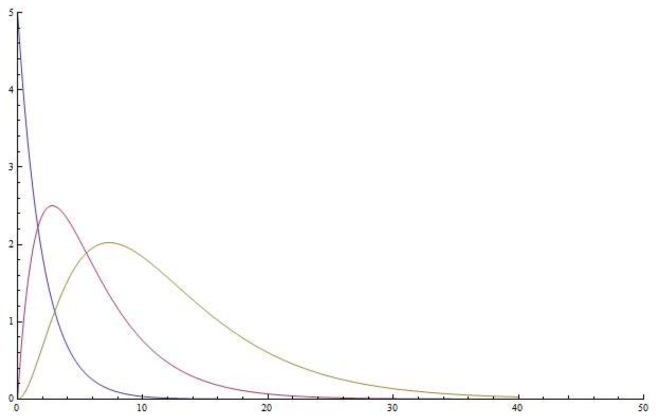

FIGURE 6.2: The exemplary brine tank cascade system simulated using Mathematica

6.1.1 Recycled Brine Tank Cascade

With a little change to the system the loop can be closed, so that tank 3 also fills up tank 1:

$$x_1'(t) = -\frac{1}{6}x_1(t) \qquad + \frac{1}{6}x_3(t) \tag{6.10}$$

$$x_2'(t) = -\frac{1}{6}x_1(t) - \frac{1}{3}x_2(t) \tag{6.11}$$

$$x_3'(t) = \qquad \frac{1}{3}x_2(t) - \frac{1}{6}x_3(t) \tag{6.12}$$

Taking the same initial values ($x_1(0) = 5$, $x_2(0) = x_3(0) = 0$), the exact solution becomes:

$$x_1(t) = 2 + 3\cos\frac{t}{6}\cdot e^{-t/3} + \sin\frac{t}{6}\cdot e^{-t/3} \tag{6.13}$$

$$x_2(t) = 1 - \cos\frac{t}{6}\cdot e^{-t/3} + 3\sin\frac{t}{6}\cdot e^{-t/3} \tag{6.14}$$

$$x_3(t) = 2 - \cos\frac{t}{6}\cdot e^{-t/3} - 2\sin\frac{t}{6}\cdot e^{-t/3} \tag{6.15}$$

Figure 6.3 shows a plot of the system generated with MOI-Sim.

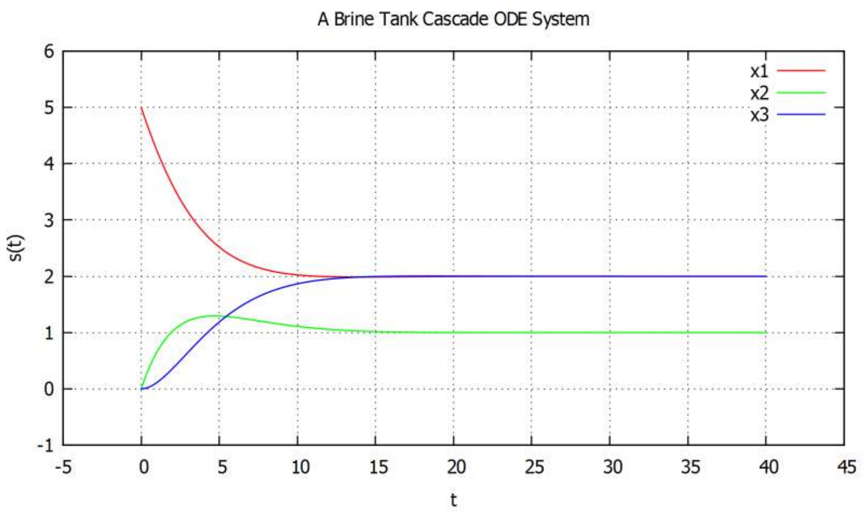

FIGURE 6.3: The exemplary brine tank cascade system with recycling mechanism

The systems show the general functioning of MOI-Sim. The most important aspect is the ability to plug in more processes, acting on the same state, without having to change anything in the base model. A process using up water in tank 1 to drive something could easily be added, without touching the original system.

6.2 A Whole Cell Model After Tobias Bollenbach

Described and used in the paper of Tobias Bollenbach et al. [42], the following model is a very simple whole cell model, describing the host using only 4 variables. P describes the amount of protein, C the total DNA, R the number of ribosomes and A the number of energy molecules (e.g. ATP).

The dynamics of the four variables are defined with the following equations:

$$p'(t) = s_p - g \cdot p(t) \tag{6.16}$$

$$c'(t) = s_c - g \cdot c(t) \tag{6.17}$$

$$r'(t) = s_r - g \cdot r(t) \tag{6.18}$$

$$a'(t) = s_a - (g + k_{deg}) \cdot a(t) - (\epsilon_p \cdot s_p + \epsilon_r \cdot s_r + \epsilon_c \cdot s_c) \tag{6.19}$$

Where g is the growth rate (leads to growth-dependent dilution) and the s_x are production terms, depending on various cell variables and parameters. For a detailed study of the model, please consider the paper and supplementary documentation by Tobias Bollenbach et al.

Figure 6.4 shows a plot of the Bollenbach model. The model can be used as a simple host circuit to test models under a energy-managing host cell. The next section introduces such a circuit and studies its embedding into the Bollenbach model.

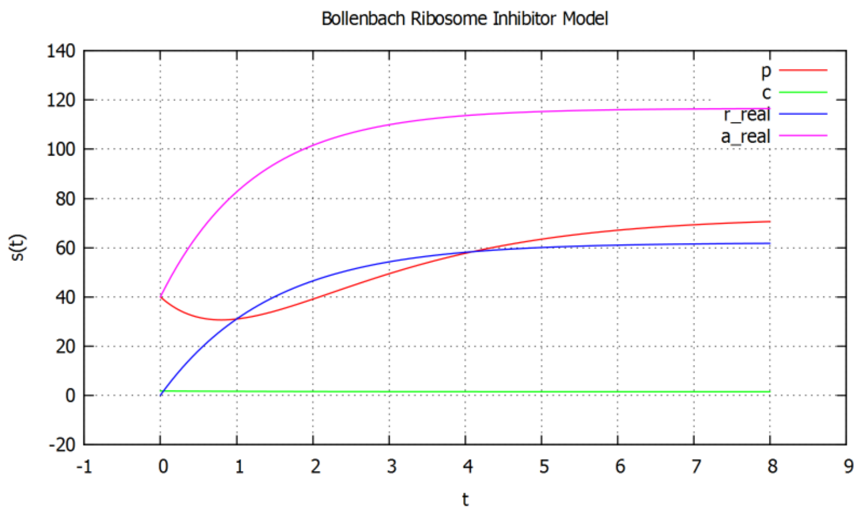

FIGURE 6.4: The Bollenbach whole cell model

6.3 A Resource Processing Model Based on the Bollenbach Model

In order to show how to integrate custom circuits into a host model, the Bollenbach system was extended to process resources. In particular, there is an external resource E_w, a processed resource B_w, a protein that performs the processing P_w and two transcription factors TF_a and TF_{E_w} which measure availability of A and E_w respectively.

Figure 6.5 shows the additions to the original model. The additional functionality is modeled by adding two more processes to the system of section 6.2, namely *ResourceProcessing* and *ResourceProcessingProteinTranslation*. The first process performs the processing itself, whilst the second one models the translation of P_w depending on the transcription factors.

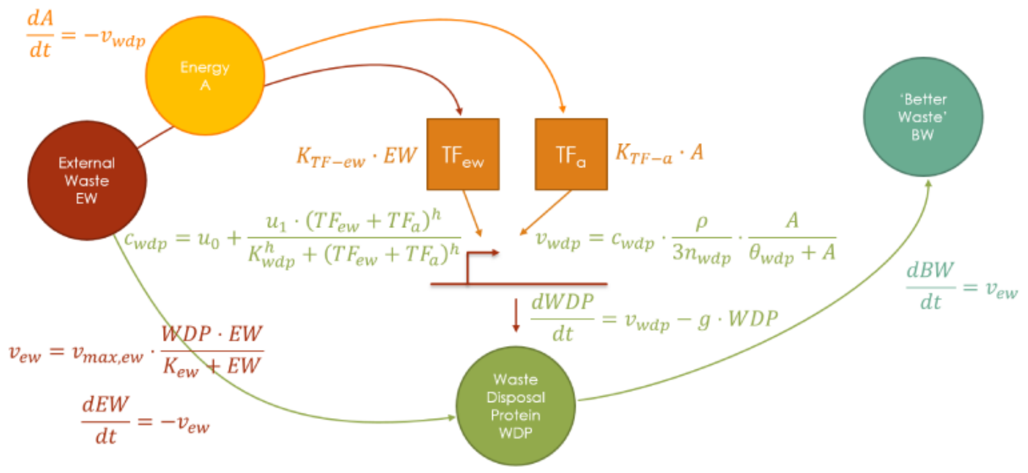
$\frac{dA}{dt} = -v_{wdp}$
Energy A
External Waste EW
$K_{TF-ew} \cdot EW$
TF$_{ew}$
TF$_a$
$K_{TF-a} \cdot A$
'Better Waste' BW
$c_{wdp} = u_0 + \frac{u_1 \cdot (TF_{ew} + TF_a)^h}{K_{wdp}^h + (TF_{ew} + TF_a)^h}$
$v_{wdp} = c_{wdp} \cdot \frac{\rho}{3n_{wdp}} \cdot \frac{A}{\theta_{wdp} + A}$
$\frac{dBW}{dt} = v_{ew}$
$\frac{dWDP}{dt} = v_{wdp} - g \cdot WDP$
$v_{ew} = v_{max,ew} \cdot \frac{WDP \cdot EW}{K_{ew} + EW}$
$\frac{dEW}{dt} = -v_{ew}$
Waste Disposal Protein WDP

FIGURE 6.5: The resource processing model based on the Bollenbach model

The equations describing the additions to the systems are as follows:

$$e'_w(t) = -v_{max,e_w} \cdot \frac{p_w(t) \cdot e_w(t)}{K_{e_w} + e_w(t)} \tag{6.20}$$

$$b'_w(t) = v_{max,e_w} \cdot \frac{p_w(t) \cdot e_w(t)}{K_{e_w} + e_w(t)} \tag{6.21}$$

$$p'_w(t) = (v_{p_w} - g \cdot p_w(t)) \tag{6.22}$$

$$a'(t) = -v_{p_w} \tag{6.23}$$

Where the first two equations are implemented in the *ResourceProcessing* process and the latter two in the *ResourceProcessingProteinTranslation* process. The simulator allows the two equations for $a'(t)$ to coexist in two different processes.

Figure 6.6 shows a plot of the resource processing model. It's easy to see that the external resource is processed depending on the availability of the processing protein into the better resource B_w.

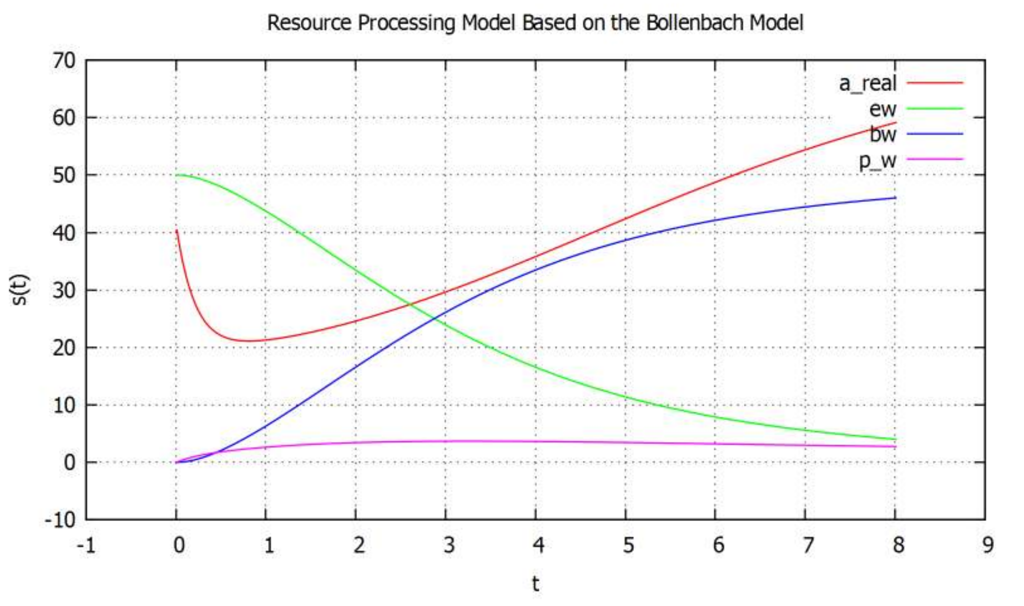

Figure 6.6: Plot of the resource processing model based on the Bollenbach model

The novelty is the extension of the system without any changes to the underlying processes. Also, depending on different needs, the model can be run using different strategies.

6.4 An Artificial Model of Comparable Complexity to the Karr Model

As a re-implementation of the Karr model was out of the scope of this thesis, an automatically generated system of comparable complexity is considered for performance measurements. The system consists of 28 processes and a state vector of 2'500 variables. The 28 processes use the same computing time distribution (see Appendix A) as the original Karr processes and modify similar number of variables each (400 variables).

Table 6.1 shows the execution time of a 60 second simulation using the different strategies. As is visible, distributed strategies can reduce the simulation time near towards the optimum (in this simulation, the longest individual process time is about 300 ms, all processes sequential take about 1 second to calculate, thus an optimal computation time would be 20 seconds).

Strategy	Execution Time (seconds)
Karr Strategy	60.038
Smash Strategy	61.502
Synchronization Points Strategy	21.674
Independent Time Scales Strategy	81.832
Distributed with Time Step Adaption	29.838

Table 6.1: Execution times of different strategies

The *independent time scale strategy* is sequential, but also does some recalculations sometimes (in order to let processes keep up with each other) and thus is the slowest. The computational overhead for the *distributed time step adaption strategy* makes the *synchronization points strategy* preferable. However, the latter can't handle different time scales.

The two simple strategies (*Karr* and *smash*) perform about the same, but aren't able to integrate complex changes. In a system with processes with equal time steps it's thus recommended to use the synchronization points strategy, if independent time scales are needed, the distributed version of the independent time scales strategy should be applied.

Concluding, the strategies allow to give a performance boost up to the maximum induced by the model. Further optimization would be on a model basis, trying to make process execution times about equal.

Chapter 7

Conclusions and Future Works

7.1 Conclusions

Whole cell modeling is a trend that manages to fascinate people from different fields. The framework developed in this thesis provides an alternative and open source way to model whole cells and complex systems in general. It focuses on decoupling of model and simulator, distributed and parallel simulation and an easy model description language (whilst still having the whole Java ecosystem at disposal). It successfully implements these points whilst also managing to improve on simulation execution speed.

Further, some theoretical founding is given for the various choices in the Karr simulator as well as the developed simulator. As the whole field is tightly linked with previous research done on numerical methods, analogies are drawn and ideas adapted. The field of whole cell models is still young, for which reason a lot of aspects had to be left out for investigation at a later stage.

The Module Integration Simulator (MOI-Sim) platform is a first step towards modularized plug-and-play simulations. Feedback and interest have been great, and it is now used for parts of the modeling for the iGEM competition by the University of Edinburgh team. There are plans to continue the research in Edinburgh, as well as via the Flowers Consortium, a union of biology modelers from various universities in Great Britain.

7.2 Future Works

On the theoretical side, future work consists of building a stronger formalism to prove properties of modularized systems. Alongside this, more theory can be adapted from previous research in numerical methods (like error estimation for time step determination and global error bounds, sensitivity analysis of black-box processes, system sensitivity analysis, efficiency arguments and more). Another interesting idea is the automated clustering of arbitrary systems, as outlined in [25], in this case especially by compiling all processes into a monolithic block and re-modularizing it in mathematically optimal ways. The idea of processes guessing the effects of others can be studied, which might be interesting for real-time systems like computer games or talk robots. The strategies developed in this thesis are one possibility and can be optimized and changed.

The implementation can benefit from adapters for various other simulation languages, e.g. Matlab or the Kappa language developed at the University of Edinburgh. This allows to integrate existing models directly into MOI-Sim, without having to change much. Distributed simulation strategies with decentralized state and optimized state sharing (i.e. only sharing parts of the state which are needed by a process) can be implemented for simulation performance. Clustered simulations, where the models run on a whole cluster communicating via a network might make sense for systems that have large individual process simulation times.

The Module Integration Simulator platform can be extended further, to provide a repository of whole cell models that can be used to simulate models under various host conditions. This would lead to a platform similar to BioBricks, where people select functions they want to see included in a cell without having to develop their own synthetic circuits, and then add their own functionality.

There is a lot of work to be done by specifying different host models of various complexities. These models can focus on various aspects of cells and provide integration for various processes.

Appendix A

Appendix

A.1 Time Spent on Different Modules

This appendix gives statistics about a timing analysis made using the Matlab profiler. It was made letting the simulator run for the first 20 second simulation time. Thus it makes no claim at being viable for overall timing statistics. The purpose is to get an approximate view how the time distribution in a whole cell model looks like. Table A.1 shows the distribution.

Process Name	Execution Time (%)
Replication	31.4%
tRNA Aminoacylation	22.8 %
Transcription	10.7 %
Translation	4.8%
Metabolism	3.7%
RNA Decay	3.7%
RNA Supercoiling	2.7%
RNA Repair	2.1%
RNA Damage	1.7%
Chromosome Condensation	1.5%
Replication Initiation	1.4%
Protein Folding	1.1%
Protein Processing I	1.0%
Rest	< 1.0%

Table A.1: Execution time distribution in the Karr model

A.2 Hill-Type Consumption of Resources

The requirements functions use an infinite supply of metabolites. As the reactions can be modeled as Hill type functions ([43]), they reduce in the following way:

$$\frac{dX}{dt} = \frac{-kAX}{C+X}$$

This will give either a linear formula or a saturation at $-kA$ for large X:

$$\frac{dX}{dt} = -kA \qquad X \to \infty \qquad \text{(saturation)}$$

$$\frac{dX}{dt} = \frac{-kAX}{C} \qquad X \to 0 \qquad \text{(linear regime)}$$

The former is used to calculate resource requirements, the latter to evolve the state.

This is not the exact way resource allocation is handled in the Karr model, but just a reasonable interpretation. In the Karr model, different formulas are used for resource requirements and consumption (they might be and are usually related, but not strictly mathematical). The formulas can implement any imaginable method.

A.3 Step Size Adaption for the Euler Method

This section walks through a possible step size adaption method for an ODE solved with the Euler method (for detailed reference see [44] and [14]). Let's start from the initial value problem:

$$x'(t) = f(t, x(t)), \quad x(t_0) = x_0 \tag{A.1}$$

The solution at $t+\Delta t$ is calculated two times: Once with Δt and once with $\Delta t/2$ applied two consecutive times. This leads to the following two solutions:

$$\begin{aligned} x(t+\Delta t)_1 &= x_0 + \Delta t \cdot f(t_0, x_0) + c \cdot \Delta t^2 && \text{(A.2)} \\ x_{\Delta t/2} = x(t+\frac{\Delta t}{2})_2 &= x_0 + \frac{\Delta t}{2} \cdot f(t_0, x_0) && \text{(A.3)} \\ x(t+\Delta t)_2 &= x_{\Delta t/2} + \frac{\Delta t}{2} \cdot f(t_0 + \frac{\Delta t}{2}, x_{\Delta t/2}) + \frac{1}{2} c \cdot \Delta t^2 && \text{(A.4)} \end{aligned}$$

Where c is a constant depending on the functions to be integrated and would change proportional to $x^{(3)}(t)$, which is neglected here though. The next step size Δt can now be determined via:

$$d = x(t+\Delta t)_2 - x(t+\Delta t)_1 \tag{A.5}$$

$$h_{i+1} = 0.9 \cdot h_i \cdot \frac{\epsilon}{|d|} \tag{A.6}$$

This ensures the local error will be bounded by ϵ. 0.9 is a safety constant to make sure the next step is successful.

Bibliography

[1] Jonathan R Karr, Jayodita C Sanghvi, Derek N Macklin, Miriam V Gutschow, Jared M Jacobs, Benjamin Bolival, Nacyra Assad-Garcia, John I Glass, and Markus W Covert. A whole-cell computational model predicts phenotype from genotype. *Cell*, 150(2):389–401, 2012.

[2] N.A. Campbell. *Biology Exploring Life*. Pearson Prentice Hall, 2006. ISBN 9780132508827.

[3] H. Lodish. *Molecular Cell Biology*. W. H. Freeman, 2008. ISBN 9780716776017.

[4] A. Maton. *Cells: Building Blocks of Life*. Prentice Hall science. Pearson Prentice Hall, 1997. ISBN 9780134234762.

[5] Oliver Purcell, Bonny Jain, Jonathan R Karr, Markus W Covert, and Timothy K Lu. Towards a whole-cell modeling approach for synthetic biology. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 23(2):025112–025112, 2013.

[6] AG Wilson, AC White, and RA Mueller. Role of predictive metabolism and toxicity modeling in drug discovery–a summary of some recent advancements. *Current opinion in drug discovery & development*, 6(1):123, 2003.

[7] Bernard Zeigler. *Object-oriented simulation with hierarchical, modular models*. San Diego, CA (USA); Academic Press Inc., 1990.

[8] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

[9] MathWorks Simulink. Simulink - simulation and model-based design, 2013. URL `http://www.mathworks.co.uk/products/simulink/`. [Online; accessed 28-August-2013].

[10] Mark Moir and Nir Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

[11] MathWorks. Matlab - the language of technical computing, 2013. URL `http://www.mathworks.co.uk/products/matlab/`. [Online; accessed 27-August-2013].

[12] Jonathan R Karr, Jayodita C Sanghvi, Derek N Macklin, Abhishek Arora, and Markus W Covert. Wholecellkb: model organism databases for comprehensive whole-cell models. *Nucleic acids research*, 41(D1):D787–D792, 2013.

[13] Karr et al. Wholecell xml simulation configuration file generator, 2013. URL `http://wholecell.stanford.edu/simulation/runSimulations.php`. [Online; accessed 28-August-2013].

[14] William H Press, Saul A Teukolsky, William T Vettering, and Brian P Flannery. *Numerical Recipes in C-2'nd Edition.* Cambridge Univ. Press. Cambridge, 1992.

[15] Guido Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, 2007.

[16] Django. The web framework for perfectionists with deadlines, 2013. URL `https://www.djangoproject.com/`. [Online; accessed 28-August-2013].

[17] MySql. The world's most popular open source database, 2013. URL `http://www.mysql.com/`. [Online; accessed 28-August-2013].

[18] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, Citeseer, 2004.

[19] Json. Javascript object notation, 2013. URL `http://www.json.org/`. [Online; accessed 28-August-2013].

[20] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

[21] Karr et al. Wholecell visualization suite, 2013. URL `http://wholecellviz.stanford.edu/`. [Online; accessed 28-August-2013].

[22] Thomas Rauber and Gudula Rünger. Parallel implementations of iterated runge-kutta methods. *International Journal of High Performance Computing Applications*, 10(1):62–90, 1996.

[23] George D Byrne and Alan C Hindmarsh. Pvode, an ode solver for parallel computers. *International Journal of High Performance Computing Applications*, 13(4): 354–365, 1999.

[24] Julio Saez-Rodriguez, Stefan Gayer, Martin Ginkel, and Ernst Dieter Gilles. Automatic decomposition of kinetic models of signaling networks minimizing the retroactivity among modules. *Bioinformatics*, 24(16):i213–i219, 2008.

[25] Dominik Bucher, Ilias Garnier, Ricardo Honorato, and Vincent Danos. Decomposition of strongly coupled systems. *Young Researchers Workshop on Concurrency Theory*, 2013.

[26] Andrea Y Weiße. *Global Sensitivity Analysis of Ordinary Differential Equations.* PhD thesis, FU Berlin, 2009.

[27] Andrea Saltelli, Karen Chan, E Marian Scott, et al. *Sensitivity analysis*, volume 134. Wiley New York, 2000.

[28] James R Rumbaugh, Michael R Blaha, William Lorensen, Frederick Eddy, and William Premerlani. Object-oriented modeling and design. 1990.

[29] Akka. Message passing for the jvm, 2013. URL `http://akka.io/`. [Online; accessed 28-August-2013].

[30] William D Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. the MIT Press, 1999.

[31] EJML. Efficient java matrix library, 2013. URL `https://code.google.com/p/efficient-java-matrix-library/`. [Online; accessed 28-August-2013].

[32] Gnu Octave. Open source numerical computation, 2013. URL `http://www.gnu.org/software/octave/`. [Online; accessed 28-August-2013].

[33] Colt. Open source libraries for high performance scientific and technical computing in java, 2013. URL `http://acs.lbl.gov/software/colt/`. [Online; accessed 28-August-2013].

[34] jOptimizer. Solving minimization problems with equality and inequality constraints, 2013. URL `http://www.joptimizer.com/`. [Online; accessed 28-August-2013].

[35] s4gnuplot. Small scala wrapper for gnuplot, 2013. URL `https://github.com/Rogach/s4gnuplot`. [Online; accessed 28-August-2013].

[36] Gnuplot. Command-line driven graphing utility, 2013. URL `http://www.gnuplot.info/`. [Online; accessed 28-August-2013].

[37] Scalatra. A tiny, sinatra-like web framework for scala, 2013. URL `http://www.scalatra.org/`. [Online; accessed 27-August-2013].

[38] Dean Wampler and Alex Payne. *Programming Scala: Scalability = Functional Programming + Objects*. O'Reilly, 2009.

[39] Atmosphere. Realtime client server framework for the jvm, supporting websockets and cross-browser fallbacks support, 2013. URL `https://github.com/Atmosphere/atmosphere`. [Online; accessed 27-August-2013].

[40] Ian Fette and Alexey Melnikov. The websocket protocol. *IETF Memo*, 2011.

[41] Grant B. Gustafson. Systems of dierential equations, 2013. URL `http://www.math.utah.edu/~gustafso/2250systems-de.pdf`. [Online; accessed 28-August-2013].

[42] Tobias Bollenbach, Selwyn Quan, Remy Chait, and Roy Kishony. Nonoptimal microbial response to antibiotics underlies suppressive drug interactions. *Cell*, 139 (4):707–718, 2009.

[43] Archibald Vivian Hill. The possible effects of the aggregation of the molecules of hmoglobin on its dissociation curves. *Proceedings of the physiological society*, page iv, 1910.

[44] Walter Gautschi. *Numerical analysis*. Springer, 2012.