SCALABLE AND REPRODUCIBLE MODELING AND SIMULATION FOR HETEROGENEOUS POPULATIONS

by

Leandro Hikiji Watanabe

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

The University of Utah

XXX 20XX

Copyright © Leandro Hikiji Watanabe 2018
All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Leandro Hikiji Watanabe**
has been approved by the following supervisory committee members:

Chris J. Myers,	Chair(s)	XXX 20XX Date Approved
Kenneth S. Stevens,	Member	XXX 20XX Date Approved
Priyank Kalla,	Member	XXX 20XX Date Approved
Tara L. Deans,	Member	XXX 20XX Date Approved
Michael Hucka,	Member	XXX 20XX Date Approved

by **Florian Solzbacher**, Chair of
the Department of **Electrical and Computer Engineering**
and by **David B. Kieda**, Dean of The Graduate School.

ABSTRACT

Advancements in the systems and synthetic biology fields have proved that biology can be engineered. The development of computer-aided design (CAD) tools has contributed to advancements in these fields. Mathematical modeling and simulation methods are important assets of CAD tools that are frequently applied to the systems and synthetic biology fields. Modeling and simulation methods are used to understand or predict the behavior of a biological system being studied. However, many modeling efforts in those fields face a reproducibility problem, where many published models are not reproducible. In order to address such issue, standards have been created for the representation of biological models. A major advantage of standards is that they enable model reuse and sharing. The leading standard representation of biological systems is the *Systems Biology Markup Language* (SBML).

The SBML standard is used to describe how biological processes affect and modify biological entities in a system. Such standard has been widely used to describe biochemical networks, cell signaling path, and gene regulation, among others. Unfortunately, not all models use SBML since there are many biological systems that SBML is incapable of representing efficiently, such as heterogeneous cellular populations. This dissertation explores extensions to SBML for the efficient representation of large heterogeneous cellular populations and simulation methods that can simulate such complex models efficiently. Since cellular populations are inherently hierarchical, this dissertation proposes an efficient simulator for hierarchical SBML models. Since the hierarchical structure is preserved in the proposed simulator, the hierarchical simulator is a perfect fit for handling hybrid models. However, no one has explored the coupling of different modeling formalisms within the same SBML model. Hence, this dissertation proposes a methodology that can be used to describe hybrid models. Such methodology is demonstrated by using dynamic flux balance analysis (DFBA) models as examples and such models can be successfully exchanged between tools. This dissertation also discusses extensions to the SBML data

model to support regular structures in the form of arrays. Arrays is well-suited for population models since population models use large regular structures. Another application of arrays is microsimulation of disease models, where a population of individuals with unique characteristics need to be model. With the proposed arrays extension, simulators need to scale in order to handle the increase complexity that the arrays extension introduces. Hence, this dissertation also proposes an efficient simulation method that takes advantages of arrays.

For my parents.

CONTENTS

ABSTRACT . . . . **iii**

LIST OF FIGURES . . . . **ix**

LIST OF TABLES . . . . **xii**

LIST OF ALGORITHMS . . . . **xiii**

ACKNOWLEDGEMENTS . . . . **xiv**

CHAPTERS

1. INTRODUCTION **1**

1.1 Research Reproducibility 4
1.2 Modeling and Analysis 5
1.3 Contributions 7
1.4 Dissertation Outline 8

2. BACKGROUND **10**

2.1 Genetic Circuits 10
2.2 Modeling Frameworks 13
2.2.1 Chemical Reaction Networks 13
2.2.2 Constraint-based Models 14
2.3 Analysis Frameworks 16
2.3.1 Deterministic Simulation 16
2.3.2 Stochastic Simulation 18
2.3.3 Flux Balance Analysis 19
2.4 Data Standards 21
2.4.1 COMBINE Initiative 22
2.4.2 The Systems Biology Markup Language 22
2.4.2.1 Core Elements 23
2.4.2.2 Hierarchical Composition Package 24
2.4.2.3 Flux Balance Constraints Package 24
2.4.3 The Simulation Experiment Description Markup Language 24
2.4.4 COMBINE Archive 25
2.5 iBioSim Version 3 25

3. HIERARCHICAL SIMULATION **30**

3.1 Algorithm Overview 30
3.2 Hierarchical Stochastic Simulation Algorithm (hSSA) 32
3.3 Example 34
3.4 Extensions to hSSA to Support SBML 39

3.5 Results 46
3.6 Summary 50

4. MODELING AND SIMULATION OF ARRAYS IN SBML 52

4.1 Arrays Extension in SBML 52
4.1.1 Dimension Class 53
4.1.2 Index Class 54
4.2 Creating Models using Arrays 54
4.3 Arrays Simulation 55
4.3.1 Flattening of Array Constructs 55
4.3.2 Arrays Simulation Algorithm 58
4.4 Results 62
4.4.1 Repressilator 62
4.4.2 Genetic Toggle Switch 62
4.4.3 Genetic Toggle Switch with Cell Communication 65
4.4.4 Towards Scalable Modeling and Simulation 69
4.5 Summary 73

5. TOWARDS REPRODUCIBLE HYBRID MODELING 75

5.1 Multi-framework Computational Models 75
5.2 Dynamic Flux Balance Analysis 76
5.3 Exchangeability & Reproducibility of Models 77
5.4 Methods 78
5.4.1 Stationary optimization approach (SOA) 78
5.4.2 Model Reproducibility 80
5.5 Results 81
5.5.1 Scheme for Dynamic Flux Balance Analysis 81
5.5.2 Minimal Example (`toy_wholecell`) 84
5.5.3 Diauxic growth in *E. coli* (`diauxic_growth`) 85
5.5.4 *E. coli* core (`ecoli`) 87
5.6 Summary 90

6. TOWARDS REPRODUCIBLE DISEASE MODELS USING THE SYSTEMS BIOLOGY MARKUP LANGUAGE 93

6.1 Overview 93
6.2 Disease Modeling Examples 95
6.2.1 Example 1: Simple Example 95
6.2.2 Example 2: Three State Markov Model 98
6.2.3 Example 3: Stratified Markov Model 100
6.2.4 Example 4: State Transition Model Dependent on Changing Parameters 102
6.2.5 Example 5: State Transition Model with Treatment and Costs 104
6.3 Results 107
6.4 Summary 108

7. CONCLUSIONS 140

7.1 Summary 140
7.2 Future Work 142

7.2.1 Simulation of Population Dynamics 142
7.2.2 Explore Ways to Improve Performance 142
7.2.3 Improve Arrays Support 142
7.2.4 Extend Hybrid Modeling Capabilities 143
7.2.5 Extend Disease Modeling Capabilities 143
7.2.6 Improve SED-ML support 144
7.2.7 Enriched Cellular Population Modeling 144

REFERENCES **145**

LIST OF FIGURES

1.1 A Genetic circuit. 2
1.2 Cell population with cell-cell-communication. 6
2.1 An example of a transcriptional unit. 10
2.2 Functional description of genetic circuis. 12
2.3 A chemical reaction network for the repressilator. 14
2.4 An overview of constraint-based models for genome-scale metabolic networks. 15
2.5 ODE simulation results for the repressilator circuit. 17
2.6 SSA simulation results for the repressilator circuit. 20
2.7 The timeline of the iBioSim tool. 26
2.8 The `iBioSim` workflow. 27
3.1 An example of a hierarchical chemical reaction network. 31
3.2 Comparison of performance of SSA using flattening and the hierarchical approach for models without replacements and deletions. 48
3.3 Comparison of performance of SSA using flattening and the hierarchical approach for models with replacements and deletions. 49
3.4 Efficient data structure for hierarchical models. 51
4.1 UML diagram for the arrays package in SBML. 53
4.2 Creating model using SBML Arrays. 55
4.3 Comparison of performance of SSA using flattening and the arrayed approach for the repressilator example. 63
4.4 An illustration of the genetic toggle switch design. 64
4.5 Runtime comparison between the arrays simulator and flattening approach for the genetic toggle switch circuit. 65
4.6 Comparison of performance of SSA using flattening and the arrayed approach using the next reaction method. 66
4.7 An illustration of the genetic toggle switch coupled with quorum sensing. 68
4.8 Modeling grid-based models using arrays to represent models that include cell communication. 68
4.9 Illustration of a dependency graph for a reaction network. 70

4.10 Comparison of performance of SSA using flattening and the arrayed approach using the next reaction method. 71
4.11 Comparison of performance of the arrays simulation for models abstracted with stoichiometry amplification. 72
4.12 Modeling grid-based models using arrays to represent models that include cell communication. 73
5.1 Diagram that shows a simulation algorithm for simulation DFBA models using the SOA method. 79
5.2 Scheme for encoding DFBA models in SBML. 83
5.3 Toy model for the whole-cell. 85
5.4 An example of a transcriptional unit. 86
5.5 Simulation results for the diauxic growth model in `iBioSim` and `sbmlutils`. . . 88
5.5 (continued). 89
5.6 DFBA simulation results for core metabolism of *E. coli*. 91
6.1 State transition diagram of a simple Markov model. 96
6.2 State transition diagram of a three state Markov model. 98
6.3 State transition diagram of a simple Markov model. 100
6.4 State transition model dependent on changing parameters. 102
6.5 State transition diagram with functions depending on Age, Male, BP (Blood Pressure). 104
6.6 Results comparison between `MIST` and `iBioSim` for one run. 109
6.6 Results comparison between `MIST` and `iBioSim` for one run. 110
6.6 Results comparison between `MIST` and `iBioSim` for one run. 111
6.7 Results comparison between `MIST` and `iBioSim` for 10 runs. 112
6.7 Results comparison between `MIST` and `iBioSim` for 10 runs. 113
6.7 Results comparison between `MIST` and `iBioSim` for 10 runs. 114
6.8 Results comparison between `MIST` and `iBioSim` for 100 runs. 115
6.8 (continued). 116
6.8 (continued). 117
6.9 Statistical analysis for example 1. 118
6.10 Statistical analysis for example 2. 119
6.11 Statistical analysis for males in example 3. 120
6.12 Statistical analysis for females in example 3. 121
6.13 Statistical analysis for males in example 4. 122
6.13 Statistical analysis for females in example 4. 123

6.14 Statistical analysis for females in example 4. 124
6.14 (continued). 125
6.15 Statistical analysis for males in example 5. 126
6.15 (continued). 127
6.15 (continued). 128
6.15 (continued). 129
6.15 (continued). 130
6.15 (continued). 131
6.15 (continued). 132
6.16 Statistical analysis for females in example 5. 133
6.16 (continued). 134
6.16 (continued). 135
6.16 (continued). 136
6.16 (continued). 137
6.16 (continued). 138
6.16 (continued). 139

LIST OF TABLES

3.1 hSSA Example: Initial State. 37

3.2 hSSA Example: Propensity State After First Iteration. 37

3.3 hSSA Example: Next Reaction Selection. 37

3.4 hSSA Example: Species States on the Second Iteration. 38

3.5 hSSA Example: Reaction Propensities on the Second Iteration. 38

3.6 hSSA Example: Species States in the Third Iteration. 38

3.7 hSSA Example: Reaction Propensities on the Third Iteration. 40

3.8 hSSA Example: Species States on the Fourth Iteration. 40

3.9 hSSA Example: Reaction Propensities on the Forth Iteration. 40

3.10 A summary of the time that takes to flatten out different sizes of the population model of repressilator circuits. 46

4.1 A summary of the SBML document sizes corresponding to the population of repressilator circuits before and after flattening the document. 59

4.2 A comparison of a cell switching to the wrong state when cells communicate and when they do not communicate. 73

6.1 SBML events for Example 1. 97

6.2 SBML events for Example 2. 99

6.3 SBML events for Example 3. 101

6.4 SBML events for Example 4. 103

6.5 SBML events for Example 5. 105

6.5 (continued) SBML events for Example 5. 106

LIST OF ALGORITHMS

2.1 SSA . . . 18

3.1 hSSA . . . 33

3.2 hSSA initialization . . . 33

3.3 hSSA Compute Propensity . . . 35

3.4 hSSA Reaction Selection . . . 35

3.5 hSSA Perform Replacements . . . 35

3.6 Extended hSSA. . . . 42

3.7 hSSA: Assignment Rules Computation . . . 42

3.8 hSSA: Constraints Computation. . . . 42

3.9 hSSA: Events Initialization. . . . 43

3.10 hSSA: Events Handling. . . . 43

3.11 hSSA: Firing Events. . . . 45

4.1 Arrays Flattening . . . 56

4.2 Expanding Dimensions . . . 57

4.3 Arrays Simulator Initialization . . . 59

4.4 Arrays Simulator Compute Propensities . . . 59

4.5 Arrays Simulator Compute Next Reaction Time . . . 61

4.6 Arrays Simulator Next Reaction Selection . . . 61

4.7 Arrays Simulator Update State . . . 61

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help of many individuals. First, I would like to thank my advisor Chris Myers for giving me the opportunity to conduct the research presented in this dissertation. I am thankful to Chris for introducing me to the systems biology and synthetic biology fields. Not only have I learned a great deal about these fields, but also about many areas within the computer engineering field, such as asynchronous circuit design, design automation, and others. The main impact of his guidance was the improving my critical thinking to apply computer engineering concepts to other fields, such as biology. I joined his lab when I was a junior at the University of Utah and stayed in the lab since then to pursue a graduate degree. Under his guidance, I learned fundamental skills to succeed in an academic setting. Through this research, I was able to visit many places, meet new people, and learn about what other research groups are working on. In addition, I would also like to thank my supervisory committee, Priyank Kalla, Ken Stevens, Tara Deans, and Michael Hucka. Their feedback and help have improved my work greatly.

As a member of the Myers research group, I have worked alongside with many talented students in the lab: Curtis Madsen, Nicholas Roehner, Andrew Fisher, Zhen Zhang, Tramy Nguyen, Meher Saminemi, Zach Zundel, Michael Zhang, and Pedro Fontanarrosa. I would like to thank them for making this experience positive by engaging in research discussions, chatting about random facts, playing board games, among other things. I would also like to thank former students in the lab who have contributed to iBioSim: Nathan Barker, Scott Glass, Kevin Jones, Hiroyuki Kuwahara, Nam Nguyen, Tyler Patterson, and Jason Stevens.

There are many people who have made a major impact on different parts of this dissertation. In particular, I would like to thank Hiroyuki Kuwahara for his for his guidance on my research, advice on how to improve the hierarchical simulator and new ideas on where I can take this research. I would also like to thank Andreas Dräger and Nicolas

Rodriguez for the opportunity to work on JSBML. Nicolas Rodriguez, in particular, helped me greatly with the arrays package implementation. He is a great software engineer and helped me become a better software developer. I would also like to thank Lucian Smith and Sarah Keating for discussions related to the SBML arrays package. I have also had great collaborators in the past. Namely, Matthias König had a major contribution to the hybrid modeling in SBML and Jacob Barhok had a major contribution to the representation of disease models in SBML. Lastly, I would like to thank the LCP communication sub-group (Katherine Kiwimagi, Junmin Wang, Justin Letrende, Ben Weinberg, Tim Jones, Allen Tseng, Wilson Wong, Chris Myers, Jacob Beal, and Ron Weiss) for an amazing collaboration. I would also like to thank Lauren Woodruff, who was an instructor for the synthetic biology course at the Cold Spring Harbor Laboratory (CSHL). She taught me a great deal about genetic circuit design. I also thank her for providing the template to create DNA sequence drawings that are used throughout this dissertation.

I would like to thank Tramy Nguyen for always helping me when I needed help. She has been there for me throughout my entire graduate degree and I know I can always count on her. I appreciate her moral support and encouragement in my pursuit to this degree. She made this experience a fun adventure.

I would like to thank my parents, Eigi and Suzana Watanabe. It's impossible to put into words how much they have done for me. Their love and support have encouraged me to do my best. They have sacrificed a lot to give me a better opportunity for my education and I will be forever grateful. I would also like to thank my brother, Alex Watanabe, for helping me in life and in school. He answered many questions about chemistry and biology questions I've had in the past.

This research is based upon work supported by the National Science Foundation under Grants CCF-1218095 and CCF-1748200. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 1

INTRODUCTION

Humans are yet to fully comprehend biological systems due to their complexity. *Systems biology* is a field that studies how biological components interact with each other at the systems level to create more complex functions [1]. In order to do so, scientists develop the necessary technologies for experimentation and computation to gain further understanding of biology. Using computational approaches, scientists make discoveries by collecting data and applying data-driven techniques to analyze the collected data to form hypotheses on how the system works or using model-based approaches to make predictions on how the system behaves [2].

While systems biology tries to understand biology in its natural state, there is another field in biology called *synthetic biology*. The *synthetic biology* field builds upon the systems biology and *genetic engineering* fields by using engineering principles to develop novel biological designs [3, 4]. Advancements of synthetic biology have contributed to a substantial impact in the biotechnology industry [5] and this trend is likely to continue. As an example, synthetic biology has inspired the creation of novel products, such as wine without grapes by Ava Winery, plant-based burgers by Impossible Foods, and cow milk without cows by Perfect Day, just to name a few. The most exciting part of synthetic biology is that these products are just a fraction of the applications of the field. Synthetic biology has the potential of a much greater impact in society and be used for natural products [6, 7], therapeutics [8, 9], agriculture [10, 11], and biofuels [5, 12], among others.

One of the goals of synthetic biology is the systematic design of gene regulatory networks, also known as *genetic circuits*. The idea is to perform computation within a living cell [13] as shown in **Figure 1.1** on the following page. Namely, cells have input sensors and use internal logic circuits to control the dynamic expression of output actuators. Since *electronic design automation* (EDA) software tools have had success in the construction of

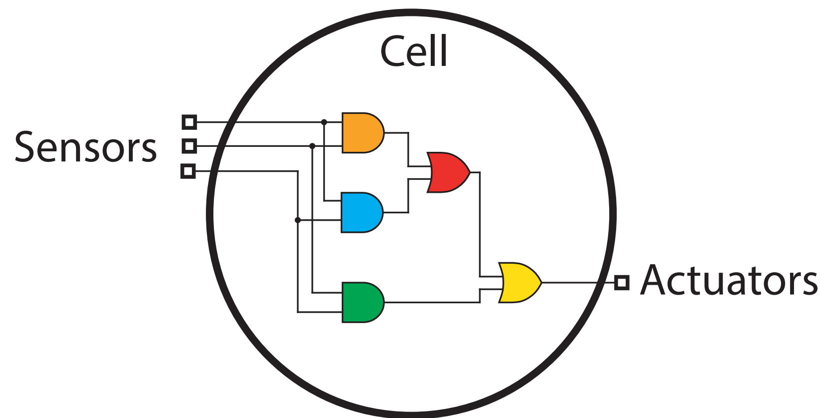


Figure 1.1: Genetic circuits are used to perform computations within a cell. The input of the circuit, include chemicals, light, heat, and others. Internal logic circuits are used to control the output of the circuit, where the output can be reporters, cell growth, chemicals, and others.

complex electronic circuits, the synthetic biology field has introduced the development of *genetic design automation* (GDA) tools [14], sometimes also referred as *bio-design automation* (BDA) [15], to enable the design-build-test cycle of genetic circuit design in a more reliable fashion. A multitude of GDA tools have been developed to assist in genetic circuit design [16–23] for different purposes, such as design, visualization, modeling, and analysis. While synthetic biology has had many positive outcomes, building genetic circuits accurately and efficiently is challenging.

In systems and synthetic biology, mathematical models and dynamic simulation are widely used to study the behavior of biological systems. Mathematical models have been successfully used in both fields for the analysis of complex systems. In systems biology, an example is the whole-cell model of the human pathogen called *Mycoplasma genitalium* [24]. This model captures all of the genes and molecular processes of this bacteria and it has been validated using experimental data. These facts made this project an important breakthrough in the systems biology field. In synthetic biology, many scientists are interested in cell-cell communication of cellular populations [25, 26], and such population models are highly complex because they are often large scale and they a large number of components in the system that interact with each other.

As mathematical models become more complex, simulation methods need to scale in order to address such complexity. Throughout the years, many simulation tools have

been developed for these purposes. For example, Stochastic Pi Machine (SPiM) [27] is a language that is used to design and simulate biological processes. The language is based on pi-calculus, which belongs to the family of process calculus, a formalism used to express concurrent systems. BioNetGen is a tool that can be used to simulate large complex models based on *rule-based systems* [28]. The gro tool [19] can be used to express and simulate the behavior of cellular populations. CompuCell3D [29] is another tool for simulating large-scale models of cellular populations that supports the representation of several dynamic processes, such as cell growth, division, and diffusion among others.

Although several modeling and simulation tools have been developed, many of them use tool-specific data representations that other tools are not capable of interpreting. Some tools, such as the BioNetGen tool can be exported into a standard representation, but the generated model suffers from a large state space inherited from the modeled pathways and loses all the benefits of rule-based modeling.

The lack of tool interoperability is problematic since it prevents different groups from collaborating with each other. In order to address this issue, many tools have adopted the use of data standards. Data standards offer many benefits. For instance, they allow model exchange and model reuse, which are critical for building complex models and designs. In addition, data standards allow for research reproducibility, which is key for model and design validation and for sharing knowledge.

The leading standard representation of models in systems and synthetic biology is the *Systems Biology Markup Language* (SBML) [30]. There are many SBML-compliant simulation tools. This includes iBioSim [31–33], a GDA tool for genetic circuit design, COPASI [34], a tool for simulation and analysis of biochemical networks, libRoadRunner [35], a high-performance simulation engine, SBMLSimulator [36], a Java simulator that is built on top of the Systems Biology Simulation Core Library [37], BioUML [38], a Java platform for biomedical research, and many others.

Although these tools are efficient for the analysis of SBML core models of chemical reaction networks, these tools do not scale well for more complex models, such as cellular populations. Cellular populations have regular structures that are best represented using hierarchy and arrays. However, tools flatten out such constructs. Not only the flattening procedure is computationally intensive, but also the model loses important structural

information.

1.1 Research Reproducibility

Over the past years, concern about research reproducibility has risen in multiple fields and such concern is justifiable. A recent Nature survey has shown about 70% of researchers from a pool of 1,576 individuals have tried and failed to reproduce a published experiment [39]. More astonishing is the fact that more than half of these researchers have unsuccessfully attempted to reproduce their own experiment. Irreproducible research is a problem encountered in many fields, such as computer science [40], economics [41], and psychology [42]. The outcome of the survey is alarming. Nonetheless, the fact that there exists many research experiments that are not reproducible is not a surprise. After all, the definition of the term reproducibility is non-standard and it is often misinterpreted by replicability or repeatability.

According to [43], reproducibility should be defined as three distinct terms: methods reproducibility, results reproducibility, and inferential reproducibility. *Methods reproducibility* refers to the application of the same set of procedures, dataset, and tools when repeating an experiment. *Results reproducibility* refers to obtaining the same results when following the same methodologies. Finally, *inferential reproducibility* refers to an independent study or reanalysis of the original study that results in qualitatively the same results. Data standards help with all three reproducibility facets. Reproducibility is achieved because standards allow the reuse of the same models in the same set of tools and procedures of an experiment. Models should give the same results quantitatively when analyzed in different tools. This is due to well-defined modeling semantics in well-established standards. Well-defined semantics provides anyone the ability to interpret a model unambiguously, which allows anyone to apply the same methodologies of an experiment described by a standard without reusing the model represented in the standard.

In systems and synthetic biology, in particular, the use of data standards, standard-compliant software tools, and data repositories for sharing published models and designs is critical [44–46]. Unfortunately, many researchers do not see the benefits of reproducibility [47]. In the modeling community, many researchers prefer publishing their own interpretation of their results without sharing their models [48]. Luckily, there has been an

increase in efforts towards reproducibility in the systems and synthetic biology fields with the adoption of standards [30, 49–51].

1.2 Modeling and Analysis

Mathematical models of genetic circuits and other biological designs have played an important role in biology, since they allow scientists to perform numerical analysis to gain intuitions of a design that is being studied [2]. Several modeling formalisms have been used to represent biological designs, such as chemical kinetics [52], Boolean networks [53], Bayesian methods [54, 55], constraint-based methods [56], rule-based methods [28], and many others. Chemical kinetics modeling, in particular, is one of the most widely used. This type of modeling describes how molecules change in terms of rates of chemical reactions. The rates can be realized as a set of *Ordinary Differential Equations* (ODEs) using the *Law of Mass Action*. ODEs are used to retrieve the time-course data in a deterministic fashion. Another way to reason about biological designs is to use stochastic simulation since biological processes are inherently noisy. Stochastic simulation can capture noise effects. The most widely used simulation method is Gillespie's *Stochastic Simulation Algorithm* (SSA) direct method [57].

A major problem of SSA is that it is computationally intensive. Hence, many variants of the algorithm have been proposed, such as tau leaping [58, 59], composition/rejection [60], and next reaction method [61]. Additionally, there are reaction-based abstraction methods to simplify the models and speed up simulation [62].

The SBML standard is quite expressive and certainly not limited to chemical reaction networks. The standard has been used for cell signaling pathways, Boolean models, and petri-nets, among many other applications. A major feature of SBML is the support for hierarchical models [63]. Hierarchy is an important engineering concept in many fields. For example, electrical and computer engineering use hierarchy to design electronics circuits at different levels (e.g. transistor, gate, and module) and software engineering use the principle of separation of concerns. Hierarchy helps with abstraction, which is critical for complex systems like the ones found in biology. Biological models can be realized at the molecular, cell, population, tissue, organ levels, and so on. Determining the correct level of abstraction is key in modeling.

In synthetic biology, it is reasonable to use a bottom-up design strategy when designing genetic circuits since designs are created from individual parts and then designs are built off from existing designs of these parts. Several designs have been constructed in a single-cell [64, 65]. However, there are applications that require a population of cells, such as biomedical applications [66, 67]. Modeling cellular populations hierarchically is the most straightforward way to do so, where a top-level model is composed of many sub-models, each representing a cell as shown in **Figure 1.2** on the current page. This is an example of cell-cell communication for population-based models. In this case, Cell A and Cell B can be modeled separately and connected in a top-level model. While SBML can express such models, existing tools flattens out the hierarchy. Not only is flattening a computationally expensive, but also the model loses important structural information. A better solution is to implement a hierarchical simulation method that takes advantage of hierarchy for better scaling up of simulation. A hierarchical simulation method also allows the simulation of different formalisms using hybrid methods. This is particularly important because different applications are better suited for different formalisms.

Although SBML is an expressive modeling language, there are many limitations. For instance, SBML cannot represent regular structures efficiently. Such a limitation makes it difficult to express population-based models efficiently and even makes complex models

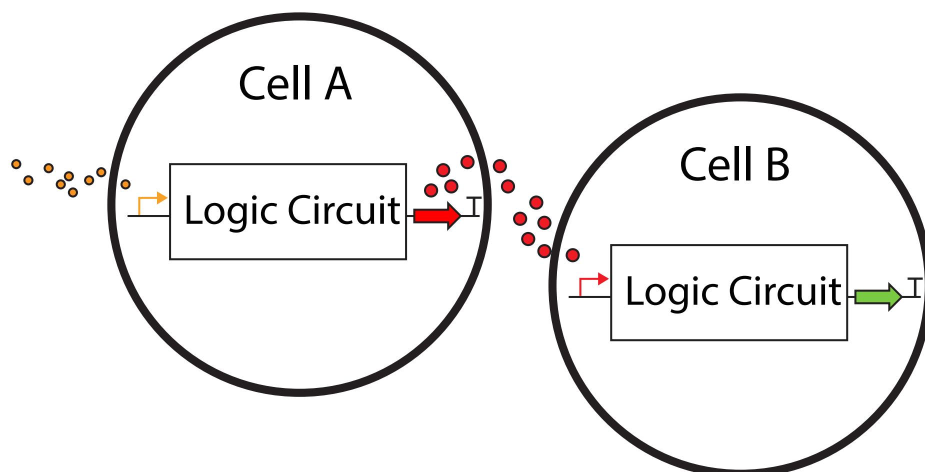


Figure 1.2: This is an example of how hierarchical modeling can be used for biological systems. Hierarchical models are helpful to encode population models. In this example, there are two different cells that communicate via a molecule: Cell A and Cell B. Each cell can be modeled separately and put together in a top-level model.

infeasible to represent such as the whole-cell model. There were efforts to encode the model in SBML [68] but such encoding proved to be nearly impossible for several reasons. For example, different parts of the model are encoded using different formalisms and some parts require arrays of hundreds of thousands of elements, such as for representing the genome of the *Mycoplasma genitalium* bacteria.

A major drawback of standards is the fact that data standards limit flexibility of design and model representation since standards backed up by a community need consensus before introducing new changes. Unfortunately, this slows down progress of many research groups, and it is a major factor that discourages them to adopt standards. Nonetheless, the benefits of standards and standard-compliant tools far-exceeds the drawbacks. SBML, in particular, is constantly evolving and new extensions can be proposed if the community finds them beneficial, but in order for an extension to be officially part of the standard, it is necessary to have tools using the extensions.

1.3 Contributions

The main contributions of this dissertation are divided into two groups: building tool infrastructure and making use of it. Since hierarchy is a good modeling practice and many researchers rely on it, the first contribution is a hierarchical simulator for SBML models. Unlike existing simulation tools that flattens out the hierarchy, the hierarchical simulator skips the expensive flattening routine. The second contribution is a proposed extension to the SBML modeling standard to support regular structures using arrays. This dissertation also presents an efficient simulation method to support the arrays in SBML by avoiding arrays flattening and exploiting the array structure for representing the state variable. This dissertation presents a simulation algorithm that supports the arrays extension and simulates it efficiently. The third contribution is a reproducible mechanism to encode hybrid models in SBML. In order to simulate such models, the hierarchical simulation method is necessary because different formalisms need to be decoupled from each other during simulation. Lastly, use-cases of SBML compliant tools for population modeling. Namely, arrays in SBML are used for encoding disease models. Without arrays, the representation of disease models at the individual level is impossible.

In summary, the main contributions of this dissertation are:

- an efficient hierarchical simulation for SBML hierarchical models;
- an array extension to SBML and an efficient simulator method that can analyze arrays structures more efficiently;
- an SBML-compliant hybrid simulator;
- exploration of SBML for population-based models.

The proposed research work has been integrated in `iBioSim` version 3. This tool is freely available for download at: `http://www.async.ece.utah.edu/iBioSim/`.

1.4 Dissertation Outline

This dissertation is organized as follows. Before discussing the contributions of this dissertation, Chapter 2 provides a background on genetic circuit design, which includes information about genetic circuits that are necessary to understand the later chapters. In addition, this chapter explains how genetic circuits are modeled, and how such models can be simulated. This chapter also provides an overview about research reproducibility and how standards are used towards reproducibility. Lastly, a high-level overview of the `iBioSim` tool, which is the tool where the main contributions of this dissertation have been integrated.

Chapter 3 discusses the hierarchical stochastic simulation algorithm. First, it discusses the algorithm and shows how the algorithm works with an example. Then, it compares the efficiency of the hierarchical simulation method compared to a flattening methodology.

Chapter 4 discusses the proposed arrays extension to SBML. This includes an overview on how the data model is modified to support arrays, a simulation algorithm for SBML models that use arrays, and an example. The efficiency of the arrays simulation is compared to a flattening methodology.

Chapter 5 discusses the encoding of hybrid models in SBML. By extending the hierarchical simulator infrastructure, the simulation of hybrid models is enabled. While the encoding works for any arbitrary modeling formalism, the hierarchical simulator only supports stochastic models coupled with constraint-based models and ODE-based models coupled with constraint-based models. This proposed encoding is demonstrated to be exchangeable between two tools as proof-of-concept.

Chapter 6 describes some use-cases of this work. First, this chapter shows an application for SBML arrays by using the extension for the encoding of disease models where microsimulation is enabled by the arrays simulator. This chapter also demonstrates how SBML can be used in a tool workflow for population-based models of genetic circuits for communication systems.

Finally, Chapter 7 concludes this dissertation with a summary of the accomplishments of this work and potential future directions of this research.

CHAPTER 2

BACKGROUND

In this chapter, some concepts are explained to better understand the contributions of this dissertation. Section 2.1 describes genetic circuits as they are used in examples in the following chapters. Section 2.2 discusses different modeling formalisms used to describe biological systems. Section 2.3 describes analysis methods for analyzing biological models. Section 2.4 discusses data standards in systems and synthetic biology. Finally, Section 2.5 describes the `iBioSim` tool, which is the where the main contributions of this dissertation have been integrated.

2.1 Genetic Circuits

All organisms are made up of cells. Some organisms are composed of a single cell (e.g. bacteria) and some are composed of many cells (e.g. humans). Within each cell, a *deoxyribonucleic acid* (DNA) molecule includes *coding sequences* (known as *genes*) that provide instructions on how to construct *proteins*. Proteins are macromolecules made from chains of amino acids that serve many important functions in all organisms. An example of a DNA molecule is shown in **Figure 2.1** on this page, which shows the necessary parts for coding a *ribonucleic acid* (RNA) molecule and the synthesis of RNA to protein. A DNA

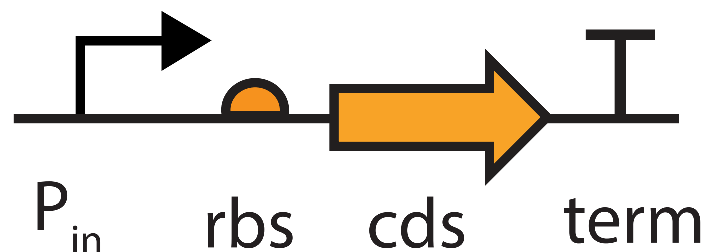


Figure 2.1: This is a representation of a transcriptional unit. A transcriptional unit typically contains a promoter (P_{in}), ribosome binding site (rbs), coding sequence (cds), and a terminator (term).

sequence with parts necessary for protein synthesis is known as a *transcriptional unit*.

A high-level description on how genetic circuits operate is shown in **Figure 2.2** on the next page. The first step to understand how genetic circuits work is to know how protein synthesis work. The process of protein synthesis is shown in **Figure 2.2a** on the following page. Protein synthesis begins with a process known as *transcription*. In a cell, there is an enzyme called *RNA polymerase* (RNAP) that initiates transcription (Step I in **Figure 2.2a** on the next page). Transcription begins when RNAP binds to a specific sequence in the DNA called a *promoter* (Step II in **Figure 2.2a** on the following page). RNAP walks the DNA to produce a single-stranded *messenger RNA* (mRNA) (Step III in **Figure 2.2a** on the next page). Transcription ends at a region within the DNA called a *terminator*. The resulting mRNA sequence is converted into a sequence of amino acids by a *ribosome* using a process known as *translation*. This amino acid sequence then folds into a protein (Step IV in **Figure 2.2a** on the following page).

The rate of this protein synthesis process can be regulated through the binding of proteins known as *transcription factors* to regions on the DNA called *operator sites*. That is, transcription factors can facilitate or inhibit the binding of RNAP to certain promoters. **Figure 2.2b** on the next page illustrates how transcription factors are used for regulation. In this example, when the input protein is not bound to the operator site (oper), then the output protein is synthesized. However, when the input protein binds to the operator site, then the downstream promoter (P_{out}) is unable to attract RNAP and this prevents transcription of the output protein. The interaction of transcriptional units can be used to create networks that control the transcription rate of the genes. These genetic regulatory networks are known as *genetic circuits*.

One well-known genetic circuit is the *repressilator*, which was constructed in *Escherichia coli* (E. coli) [65]. In the repressilator, there are three proteins produced from three promoters in which each protein acts as a transcription factor for one promoter creating a loop that forms an oscillator. Namely, the first protein, LacI, inhibits the production of the second protein, TetR. TetR inhibits the production of the third protein, CI, which inhibits the production of LacI. **Figure 2.2c** on the following page depicts the genetic circuit at the DNA-level and a high-level behavioral description of the circuit. In the high-level description, vertices are proteins and the edges represent repression relationships. Note

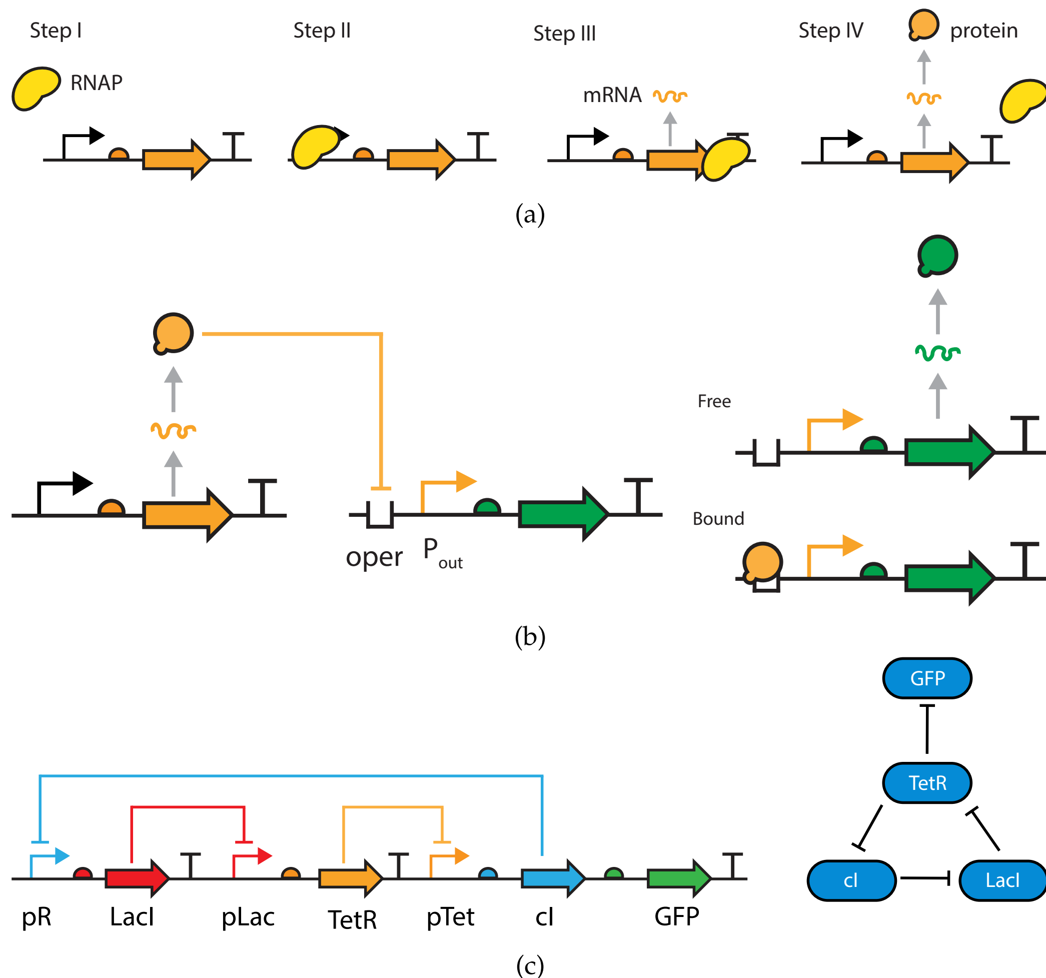


Figure 2.2: This is an illustration of how genetic circuits work. (a) This is a high-level description of protein synthesis. In a cell, there are RNAP enzymes and transcriptional units as shown in Step I. The transcription process begins when the promoter in a transcriptional unit recruits RNAP as shown in Step II. The recruited RNAP walks along the transcriptional unit to produce a single-stranded mRNA as shown in Step III. Transcription ends when RNAP reaches the terminator. The produced mRNA is then converted into a sequence of amino acids by a ribosome during translation. The sequence of amino acids is folded in a protein as shown in Step IV. (b) Proteins known as transcription factors can influence the rate of protein synthesis. In this example, there is a transcriptional unit that produces a protein that can bind to a region near the promoter called operator site (oper) of a downstream genetic circuit. When the operator site is free as shown in the figure above, the downstream transcriptional unit is able to produce proteins. However, when the operator site is bound, the promoter (P_{out}) is unable to recruit RNAP, which effectively blocks protein synthesis. This is an example of a repression regulation in gene regulatory networks. (c) The repressilator circuit is a genetic oscillator formed by three proteins (LacI, TetR, and cI) that repress each other in a loop. On the left, there is an illustration of the repressilator at the DNA-level. On the right, there is a diagram that demonstrates a high-level description of the behavior of the circuit. In this circuit, there is an additional protein called GFP, which is a reporter protein that is used to observe the circuit in a wet lab.

that a fourth protein, *green fluorescent protein* (GFP), is included in this genetic circuit to be produced at the same time as CI. The purpose of this protein is to make the cells glow green when CI is high, allowing an observer to see the oscillation in a wet lab experimental setting.

2.2 Modeling Frameworks

Biology is complex and many underlying processes in biological systems remain a mystery. However, advancements in technologies resulted in the collection of massive data that let scientists tackle such complexity. One of the uses for such massive data is to develop computational models of biological systems. Computational models use mathematical formulations to approximate the behavior of biological systems on the computer. Computational models play an important role in biology. Specifically, scientists can validate hypotheses, predict the behavior of genetic circuits before building them in a wet lab, which can be time consuming, and gain intuitions of a complex system that is being studied

2.2.1 Chemical Reaction Networks

A widely used modeling framework for genetic circuits and biological systems is the chemical reaction network model. Chemical reaction networks are mechanistic models that combine *species* (DNA, RNA, protein molecules, etc.) to form new species. In order to simulate the repressilator, the model must be converted into a set of *chemical reactions*.

The species and chemical reactions for the repressilator circuit in **Figure 2.2c** on the previous page are shown in **Figure 2.3** on the following page. Note that edges from species to reactions indicate that a species is a *reactant* (i.e., consumed by the reaction), edges from reactions to species indicate that a species is a *product* (i.e., produced by the reaction), and edges with no direction indicate that the species is a *modifier* (i.e., is neither produced or consumed). Finally, bi-directional edges indicate that a reaction is reversible, meaning that it can run in either direction. The number of molecules produced or consumed by a reaction is known as its *stoichiometry*. The edge is labeled with the stoichiometry when it is not one.

Some chemical reactions from the model are below:

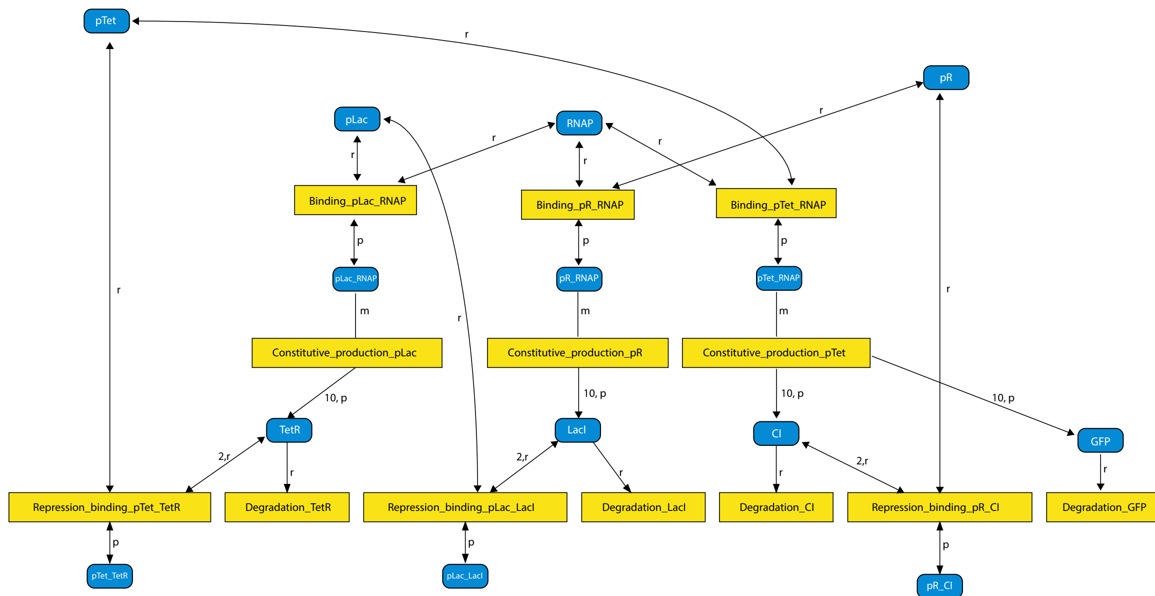
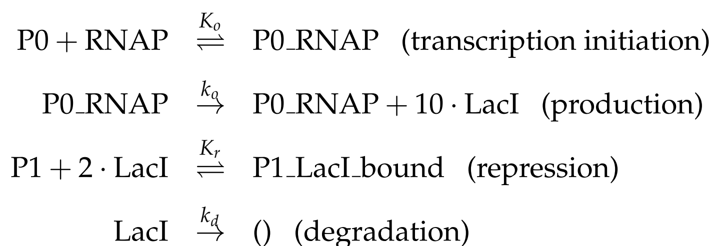


Figure 2.3: This is an illustration of the repressilator circuit as a chemical reaction network. The blue squares are species and the yellow boxes are reactions.



$$\begin{aligned} \text{P0} + \text{RNAP} &\overset{K_o}{\rightleftharpoons} \text{P0_RNAP} \quad \text{(transcription initiation)} \\ \text{P0_RNAP} &\overset{k_o}{\rightarrow} \text{P0_RNAP} + 10 \cdot \text{LacI} \quad \text{(production)} \\ \text{P1} + 2 \cdot \text{LacI} &\overset{K_r}{\rightleftharpoons} \text{P1_LacI_bound} \quad \text{(repression)} \\ \text{LacI} &\overset{k_d}{\rightarrow} () \quad \text{(degradation)} \end{aligned}$$

Note that parameters such as k_o and k_d are known as *rate constants*, and they indicate the speed or likelihood of the reaction. Parameters such as K_r and K_o are known as *equilibrium constants*, and they are ratios of the forward and reverse rate constants (i.e., $K_r = k_{rf}/k_{rr}$).

2.2.2 Constraint-based Models

One of the challenges of modeling genetic circuits as chemical reaction networks is that they require the knowledge of kinetic parameters. Such knowledge is not always available. This is especially problematic for large-scale models, such as genome-scale metabolism models, where some models contain thousands of metabolites and reactions. The most widely used method for modeling genome-scale metabolic networks is constraint-based modeling [56, 69]. This modeling formalism centered around the idea that such networks have network connectivity and physico-chemical constraints [56, 69]. More specifically, a metabolic network is constrained by the network topology imposed by reaction stoichiom-

etry. Also, in a metabolic network, there are chemical reactions that transform substrates into products. Many of these reactions are catalyzed by enzymes. Hence, the substrates or enzymes necessary for a chemical reaction in a metabolic network must be present or produced by another reaction. Without the necessary substrates and enzymes, a chemical reaction is biologically infeasible to occur so metabolic networks are constrained by substrate and enzyme availability. In addition, the biochemical reactions in the metabolic network must obey mass conservation. That is, mass and energy can neither be created nor destroyed in a chemical reaction. In a closed system, the total mass stays the same at all times. Also, such networks must obey thermodynamics laws, which limits the directionality of the reactions. Lastly, a metabolic network is constrained by the capacity of flux rates.

The procedure of modeling genome-scale metabolic networks as constraint-based models is summarized in **Figure 2.4** on the current page. The first step is to reconstruct the metabolic network from biochemical data into a curated network. Then, the network is formulated using a mathematical format known as a *stoichiometric matrix* where metabolites are associated with reactions. Each row corresponds to a particular metabolite and each column corresponds to a particular reaction in the network. A zero entry indicates that a metabolite does not participate in the corresponding reaction. A positive entry indicates that the metabolite is a product of the corresponding reaction. A negative entry indicates that the metabolite is a product of the corresponding reaction. In constraint-based modeling, biological uncertainty inherited from many unknown phenomena in biology is taken into account by computing a solution space. Instead of providing a single solution,

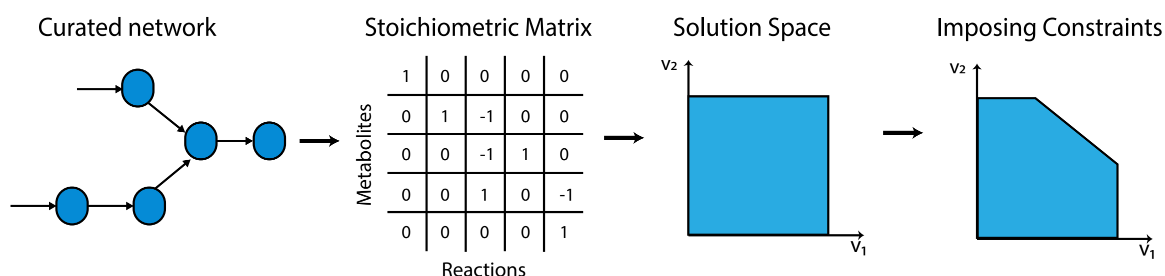


Figure 2.4: This is an illustration of constraint-based models. A curated network is constructed and formulated using a mathematical format known as stoichiometric matrix. This matrix is used to find a solution space for the design at steady-state. The solution space can be narrowed down with constraints. by imposing constraints.

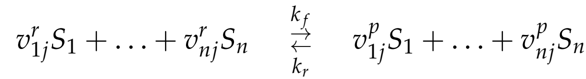
constraint-based models provide a solution space with all possible feasible phenotypic behaviors. When taking into account the physico-chemical constraints mentioned before, some solutions become unfeasible, which narrows down the solution space.

2.3 Analysis Frameworks

Numerical analysis of biological models is used to derive numerical approximations from biological systems. This allows scientists to better understand the behavior of biological systems. While there are several analysis methodologies in literature, this section focuses on three analysis methods that are used in following chapters: deterministic simulation, stochastic simulation, and flux balance analysis.

2.3.1 Deterministic Simulation

A chemical reaction model can be converted into a set of *ordinary differential equations* (ODEs) using the *law of mass action*. This law states that the rate of a reaction is its rate constant times the concentration of the reactants raised to the power of their stoichiometry. More formally, consider a model with n species $\{S_1, \ldots, S_n\}$ and m reactions $\{R_1, \ldots, R_m\}$ where each reaction, R_j, is of the form:



$$v^r_{1j}S_1 + \ldots + v^r_{nj}S_n \underset{k_r}{\overset{k_f}{\rightleftarrows}} v^p_{1j}S_1 + \ldots + v^p_{nj}S_n$$

where v^r_{ij} is the reactant stoichiometry for species S_i in reaction R_j and v^p_{ij} is its product stoichiometry. Therefore, the law of mass action states that the *rate equation*, V_j, for reaction R_j is:

$$V_j = k_f \prod_{i=1}^{n} [S_i]^{v^r_{ij}} - k_r \prod_{i=1}^{n} [S_i]^{v^p_{ij}} \tag{2.1}$$

where $[S_i]$ is the concentration of species S_i. The rate equations for all reactions that produce or consume a species, S_i, can be combined to form an ODE describing the time evolution of the concentration of that species as follows:

$$\frac{d[S_i]}{dt} = \sum_{j=1}^{m} v_{ij} V_j, \;\; 1 \leq i \leq n \tag{2.2}$$

where $v_{ij} = v^p_{ij} - v^r_{ij}$ (i.e., the net change in species S_i due to reaction R_j).

As an example, the ODE for LacI is as follows:

$$\frac{d[\text{LacI}]}{dt} = 10k_o[\text{P0_RNAP}] - k_d[\text{LacI}] - 2(k_{rf}[\text{P1}][\text{LacI}]^2 - k_{rr}[\text{P1_LacI_bound}]) \quad (2.3)$$

ODE simulation results for the repressilator are shown in **Figure 2.5** on this page. It is clear from these results that ODE simulation of this model is not an accurate representation of the repressilator circuit, since the circuit stabilizes rather than oscillates. ODE simulation is deterministic, meaning that multiple simulations starting from the same initial condition always produce the same result. Moreover, ODE methods assume a large count of the entities being analyzed. In electrical engineering, ODE methods are reasonable for simulating electronic circuits, since the number of electrons flowing through the wires is very large. However, ODE methods can be inaccurate for certain genetic circuits, such as the repressilator circuit, because the numbers of molecules of each species in a genetic circuit are typically small discrete values. In addition, since the number of molecules is typically quite small, the system can have large intrinsic noise making ODE methods less accurate.

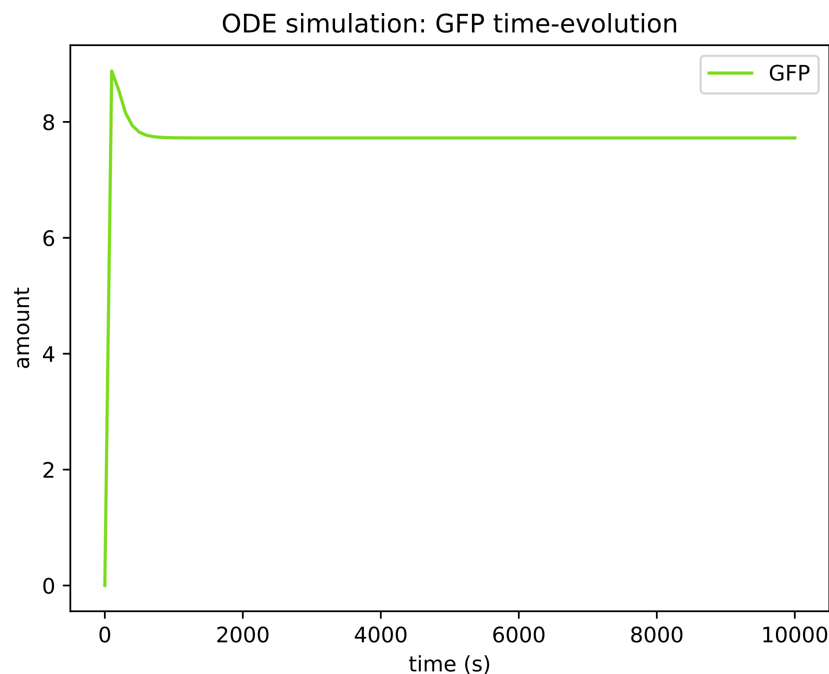


Figure 2.5: This plot shows the time-evolution of the GFP reporter protein when performing dynamic simulation using a deterministic approach based on ODEs.

2.3.2 Stochastic Simulation

While ODE simulation has been used for the analysis of many biological circuits, ODE methods may not be appropriate for certain genetic circuits, such as the repressilator circuit, because the numbers of molecules of each species in a genetic circuit are typically small discrete values [14]. In addition, since the number of molecules is typically quite small, the system can have large intrinsic noise making ODE methods less accurate. Most importantly, the chemical reactions in genetic circuits occur sporadically making them extremely noisy, a behavior not captured by ODE methods. A better method for reasoning about genetic circuits is to use the Gillespie's *stochastic simulation algorithm* (SSA) [57]. There are several variants of the SSA, but the most widely used is the *direct method* which is shown in **Algorithm 2.1** on the current page.

The SSA takes a chemical reaction network model, M, and computes a *time series simulation*, α. The SSA is essentially a Monte Carlo algorithm which treats each reaction as a random event. The simulation first initializes α to an empty sequence, computes the initial time and state, $\langle t, \mathbf{x} \rangle$, from the model, M, and appends this *time point* to α. The state of the network is $\mathbf{x} = \langle x_1, \ldots x_n \rangle$ where x_i is the current amount of species S_i. The next step computes the *reaction propensities*, $\mathbf{a} = \langle a_1, \ldots, a_m \rangle$, where a_j is the propensity for reaction, R_j, and can be approximated using the rate equation as follows:

$$a_j = k_j \prod_{i=0}^{n} (x_i)^{v_{ij}^r} \tag{2.4}$$

where k_j is the rate constant for reaction R_j and v_{ij}^r is the number of reactant molecules of species S_i consumed by the reaction.

Algorithm 2.1: Gillespie's SSA

```
1  Input: Chemical reaction network model, M;
2  Output: Time series simulation, α;
3  α := ⟨⟩;
4  ⟨t, x⟩ := initialize(M);
5  while t < timeLimit do
6  │  α := α · ⟨t, x⟩;
7  │  ⟨a, a0⟩ := computePropensities(M, x);
8  │  τ:= computeNextReactionTime(a0);
9  │  μ := selectNextReaction(a, a0);
10 │  ⟨t, x⟩ := ⟨t + τ, x + v_μ⟩;
```

For example, the propensity for the forward reaction for transcription initiation on promoter P0 is approximately:

$$k_{of} \cdot \mathrm{P0} \cdot \mathrm{RNAP} \tag{2.5}$$

The total propensity, a_0, is the sum of all propensities. The total propensity is used to determine time until the next reaction using the following equation:

$$\tau = \frac{1}{a_0} \ln \frac{1}{r_1}. \tag{2.6}$$

where r_1 is a random number drawn from a uniform distribution from $[0, 1]$. Next, the propensities are used to compute the next reaction, μ, as follows:

$$\mu = \text{smallest integer s. t.} \sum_{j=1}^{\mu} a_j > r_2 a_0 \tag{2.7}$$

where r_2 is a random number drawn from a uniform distribution from $[0, 1]$.

Finally, the time and the state are updated as shown in Algorithm **Algorithm 2.1** on the preceding page, where $\mathbf{v}_\mu$ is a vector representing the change in state due to reaction R_μ. This process repeats until the time, t, exceeds the simulation time limit. Using the SSA method, the repressilator model indeed oscillates as shown in **Figure 2.6** on the next page.

2.3.3 Flux Balance Analysis

Flux balance analysis (FBA) is a method to analyze constraint-based models [70]. According to mass balance, the rate of accumulation of a metabolite can be modeled as:

$$\frac{d\mathrm{X}_i}{dt} = \mathrm{V}_{\text{produce}} - \mathrm{V}_{\text{consume}} \tag{2.8}$$

where X_i is an arbitrary metabolite and $\mathrm{V}_{\text{produce}}$ is the rate of production and $\mathrm{V}_{\text{consume}}$ is the rate of consumption of the metabolite X_i. The complete metabolic network can be represented as:

$$\frac{d\mathbf{X}}{dt} = \mathbf{S} \cdot \mathbf{v} \tag{2.9}$$

where $\mathbf{S}$ is the formulated stoichiometry matrix for the metabolic network and $\mathbf{v}$ is the metabolic flux vector $\langle \mathbf{v}_0, ..., \mathbf{v}_n \rangle$. At steady-state, this becomes:

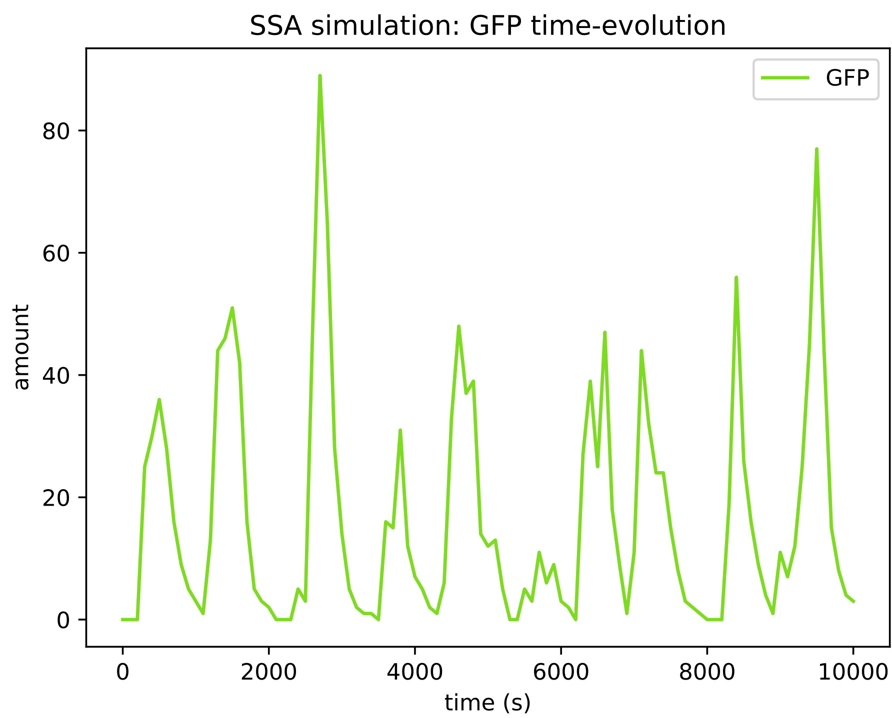


Figure 2.6: This plot shows the time-evolution of the GFP reporter protein when performing stochastic simulation.

$$\mathbf{S} \cdot \mathbf{v} = 0 \tag{2.10}$$

In order to address capacity and thermodynamics constraints, each metabolic flux can be assigned a constraint as an inequality as follows:

$$\alpha_i \leq \mathrm{v}_i \leq \beta_i \tag{2.11}$$

Flux balance analysis is solved in respect to an objective function, such as maximizing cell growth, maximizing metabolite production, minimizing ATP production, and others. The objective function can be formulated as follows:

$$Z = \mathbf{c} \cdot \mathbf{v} \tag{2.12}$$

where **c** is the weight of each flux in the objective function.

The canonical form of a FBA problem is:

$$\begin{aligned} \text{maximize/minimize} \quad & Z \\ \text{s.t.} \quad & \mathbf{S} \cdot \mathbf{v} = 0 \\ & \alpha \leq \mathbf{v} \leq \beta \end{aligned}$$

FBA can be solved using linear programming. Given a set of linear equations and linear inequalities, a value is assigned to each flux variable such that the specified linear objective function is maximized or minimized.

2.4 Data Standards

Standards are key to the success of any engineering field since standards give engineers the ability to share and reproduce models. In electronic circuit design, *hardware description languages* (HDL), such as Verilog and VHDL, are used to describe the behavior of a circuit. HDLs have played an important role in the development of complex circuits. One reason is because HDLs allow engineers to reason about behavior without caring about the logic

implementation for the design. However, a major reason is because HDLs allow engineers to collaborate with each other and reuse existing modules to create more complex designs.

For computational biology, standards can help with the representation of complex models by reusing existing ones. In addition, standards help towards research reproducibility. Thus, there have been efforts towards the development of standards and more research groups are adopting the use of standards.

2.4.1 COMBINE Initiative

The Computational Modeling in Biology Network (COMBINE) is an initiative towards the coordination between communities for the development of open standards used in computational biology [71]. COMBINE organize meetings that bring together several communities working on standardization of computational biology, facilitate interactions between the communities, provides the infrastructure for the specification of each standard and resources for the development of standards, among others.

2.4.2 The Systems Biology Markup Language

The *Systems Biology Markup Language* (SBML) is the leading standard representation for representing biological models [30]. SBML is a standard under the COMBINE umbrella whose main purpose is to provide an unambiguous data model that can be interpreted by different computational tools. For that end, SBML uses the Extensible Markup Language (XML) to encode models with well-defined semantics. Note that SBML is intended to represent how models should be interpreted and not how they should be used. SBML is backed up by a large international community of researchers and software developers who continuously help the standard to evolve. Such involvement resulted in over 280 tools that support SBML. The latest SBML specification is Level 3 Version 2 [72]. A major feature of SBML level 3 in comparison to previous levels is that it has been designed to be modular. That is, SBML level 3 contains core elements, which are elements that compliant tools must support, and package extensions, which are optional. Core elements include constructs for representing the most common modeling formalisms in computational biology, such as chemical reaction networks, discrete-event systems, boolean networks, and others. Package extensions adds support for special purpose models that are more complex and cannot be represented using only core elements. Several packages have been created, including

hierarchical composition package (comp) [63], *flux balance constraints* (fbc) [73], *groups* [74], *layout* [75], *multistate and multicomponent species* (multi) [76], *qualitative models* (qual) [77], and *rendering* (render) [78]. There are two major implementations of the SBML library: a C/C++ library called *LibSBML* [79] and a Java library called *JSBML* [80]. This section further explains the comp and fbc packages, which are the primary ones used by this dissertation. The proposed arrays package is the primary contribution of this dissertation and is discussed in further detail in Chapter 4.

2.4.2.1 Core Elements

The SBML data model has been designed primarily for the modeling of chemical reaction networks. An SBML model of a chemical reaction network contains a list of chemical *species*, each representing a chemical molecule or entity. Species reside within a *compartment*, which represents a well-stirred container with finite size. Species can be transformed into new species through *reactions*. Reactions have a list of *species references* to indicate which species participate in the reaction and how they participate (i.e. reactant, product, or modifier). Each species reference associates a species to a stoichiometry value in the reaction. Reactions have a *kinetic law*, which is a mathematical expression for the reaction rate. Such mathematical expressions can include *parameters*, which are simply named variables often used for kinetic parameters. SBML also includes *function definitions* that can be used for pre-defined named mathematical expressions. They are especially useful for mathematical expressions that are used multiple times.

SBML core is not restricted to chemical reaction networks. In fact, SBML can represent arbitrary mathematical models. *Initial assignments* are used to set the initial value of a variable. *Rules* can be used to model continuous systems. The rate of change of a variable can be expressed using a *rate rule*. *Assignment rules* are used to assign the evaluation of a mathematical expression to a variable. For discontinuous systems, SBML *events* can be used. Events are executed when its *trigger* condition evaluates from false to true. Events can have *delay*, which indicates how long a triggered event needs to wait until execution. When an event fires, its *event assignments* are performed which assign the evaluation of a mathematical expression to a variable. Finally, SBML models may have *constraints*, which are conditions that cannot be violated throughout a simulation run.

2.4.2.2 Hierarchical Composition Package

Constructing hierarchical models is a common practice in many fields, such as in electrical and computer engineering for the design of electronic circuits at different levels (e.g. transistor, gate, and module) and in software engineering, where software engineers often use the principle of separation of concerns. Hierarchy is also helpful for reasoning in biology as well. Namely, biological models can be realized at the molecular, cell, population, tissue, organs, etc. In SBML, models can be constructed hierarchically using the hierarchical model composition package (comp) [63]. This package allows the expression of hierarchy in SBML by allowing a top-level model to be constructed from a collection of sub-models. This package also enables the customization and connection of sub-models using *replacements* and *deletions*. A replacement can be used to state that an element in the top-level model replaces an element in a sub-model. A replacement can, for example, be used to state that a species in the top-level model is to replace a species in two sub-models which effectively connects the two sub-models through this species. A deletion can be used to remove part of a sub-model that is not relevant to this use of the sub-model. A deletion, for example, can be used to remove a reaction that is not needed for this particular instantiation of a sub-model. More details are given in Chapter 3

2.4.2.3 Flux Balance Constraints Package

Constraint-based modeling is a widely adopted technique for representing biological systems. Such models can be represented using the flux balance constraints package (fbc) [73]. Using this package, a model is associated with an *objective*, which is used for the optimization problem formulated by FBA. In addition, reactions are associated with *flux bounds* that are used to constrain the FBA model. More details are given in Chapter 5

2.4.3 The Simulation Experiment Description Markup Language

The *Simulation Experiment Description Markup Language* (SED-ML) [50] is a standard that encodes the necessary information for reproducing a simulation experiment. That is, a SED-ML document is associated with *models* that can be modified with *changes*. *Simulations* describe the simulation settings (e.g. simulation method, algorithm parameters, initial time, etc.). Simulation runs are described with *tasks*, which executes a simulation on a specified model. Since users typically expect to collect data from simulation, SED-ML

describes the data that simulation runs should report and how it should process the data. The generated data can then be used in *outputs*, which indicates how the data should be presented to the user.

2.4.4 COMBINE Archive

There is no universal standard that fits all applications in computational biology. However, there are many special-purpose standards, such as the standards developed under the COMBINE umbrella, that can be used in combination with other standards to encode a wider range of information. For instance, the *Synthetic Biology Open Language* (SBOL) [49, 81] can be used to describe structural and functional information of genetic circuits. The mathematical models of such genetic circuits can be encoded using SBML. The SBML model can then be visualized using the *Systems Biology Graphical Notation* (SBGN) [82]. Finally, the procedures to simulate the SBML model can be encoded using SED-ML.

Users can encode complex information by combining different standards. However, one problem is that each standard is encoded in a different file. If one of the files is missing, then critical information can be lost. Furthermore, standards can be tightly coupled together in a way that they are not meaningful on their own. For these reasons, *COMBINE Archive* has been created for grouping together COMBINE standards [83].

2.5 iBioSim Version 3

`iBioSim` is a *genetic design automation* (GDA) tool for the modeling, analysis, and design of genetic circuits that is being actively developed at the University of Utah [17, 32, 33] (see **Figure 2.7** on the following page). `iBioSim` is enabled by community developed standards that promote the model-based design of genetic circuits and allow the sharing of these designs via data repositories. `iBioSim` emerged in 2003 as a systems biology tool. The first version included `reb2sac` [62], a simulation tool that converts reaction-based networks to stochastic asynchronous circuits for efficient analysis, `GeneNet` [84], a learning tool for inferring the connectivity of genetic circuits from time-series data, and an user-interface (UI) to facilitate the usage of `reb2sac` and `GeneNet`. `iBioSim` started targeting synthetic biology applications after the tool was used to design a Genetic C-element *in silico* [85].

In the first version, the tool used a custom modeling representation called *genetic cir-*

cuit model (GCM) as a high-level abstraction to represent genetic regulatory networks. However, in the second version, the tool adopted standards for reproducibility and sharing of models and designs that included the SBML [30] and the SBOL standards [49, 81]. A schematic editor was implemented for constructing models using a graphical user-interface (GUI). New analysis methods were also implemented, including the incremental stochastic simulation algorithm (iSSA) [86], which works with small time increments and checks statistics at the end of each time step to constrain the initial values of the next time step; *stochastic model checking* [87], which uses continuous-time Markovian analysis to reason about the design's correctness with respect to stochastic properties that capture its critical behaviors; and grid-based models of dynamic cellular populations [88].

The most recent version of `iBioSim` enables a design workflow that leverages models and their analysis to guide the design choices made when constructing genetic circuits as shown in A high-level illustration of the key features of `iBioSim` is shown in **Figure 2.8** on the next page.

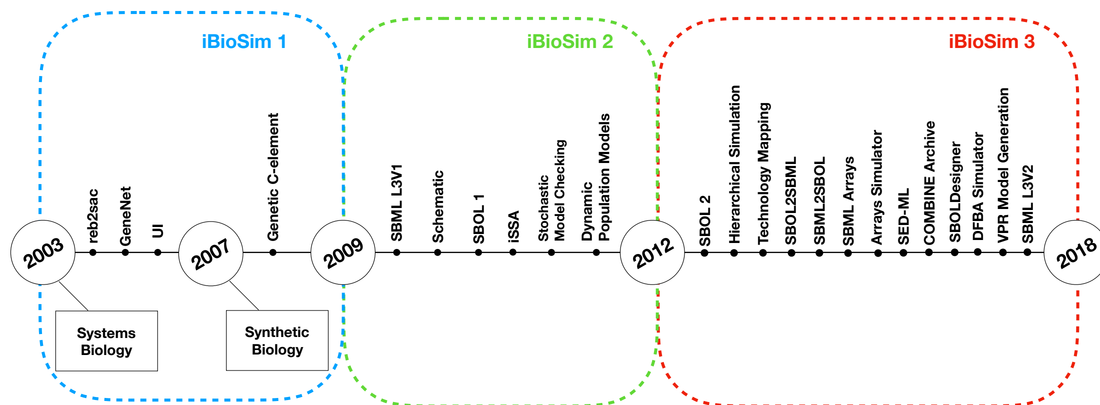


Figure 2.7: This timeline shows the evolution of `iBioSim` and highlights the key features implemented in each version. The latest version, `iBioSim` 3, includes support for new standards, the latest SBML and SBOL versions, and additional SBML packages. Furthermore, the tool supports new features for DNA circuit design, model generation, additional analysis methods, and synthesis methods.

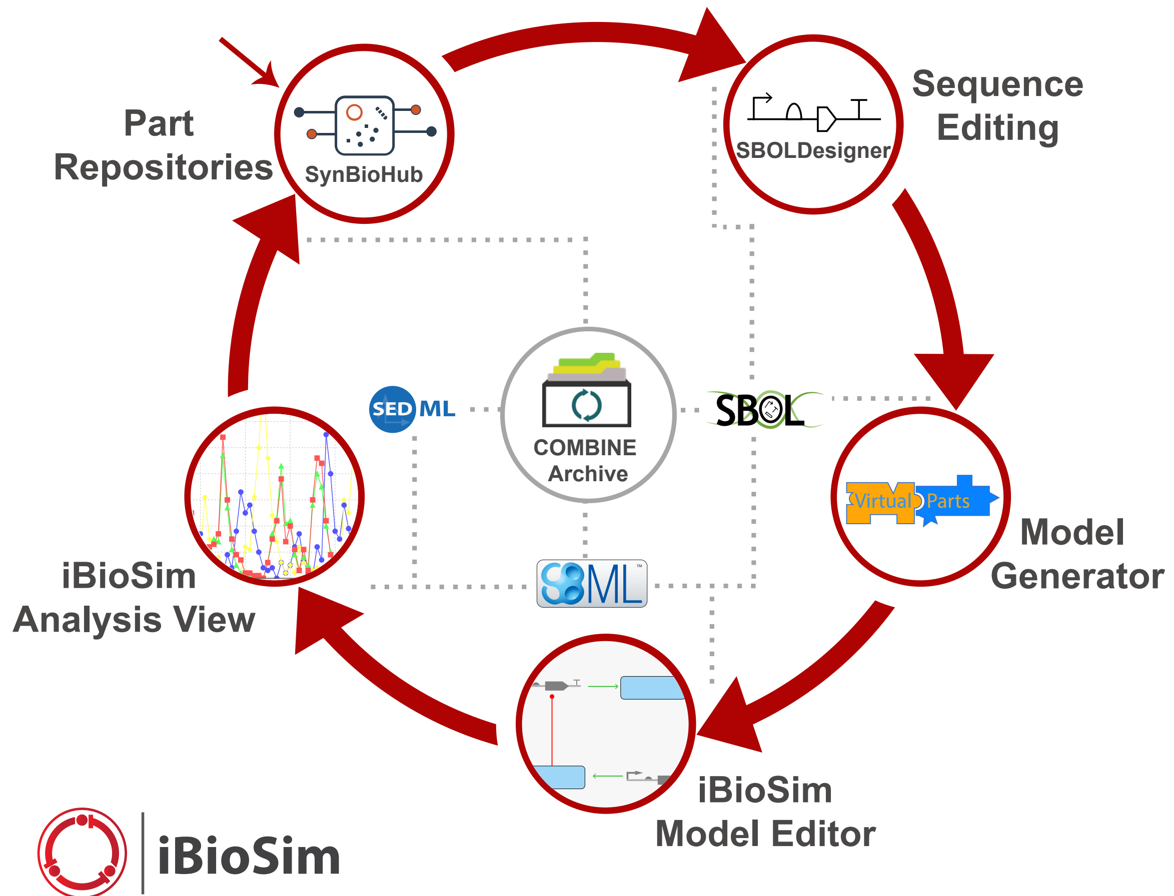


Figure 2.8: This is a high-level diagram of the genetic circuit design workflow supported by `iBioSim`. The red arrows indicate the flow between the different software components and dotted lines indicate the output of each step that is then used by the proceeding software component in the workflow. First, genetic parts encoded using SBOL are fetched from `SynBioHub` using the `SBOLDesigner` plugin to construct the DNA-level design encoded using SBOL. Next, the DNA design is augmented with interaction data using the `Virtual Parts` model generator, and the functional SBOL is converted into an SBML model. The resulting mathematical model can then be refined and parameters configured using `iBioSim`'s model editor. The SBML model can be analyzed in `iBioSim` as described by an associated SED-ML document. The data created for the SBOL parts, the SBML model, and the analysis can be shared and documented by uploading these artifacts to `SynBioHub` as a COMBINE archive.

DNA Circuit Design: The process of genetic circuit design in `iBioSim` begins by using the `SBOLDesigner` [89] tool to select genetic parts from the `SynBioHub` part repository [90]. This DNA-level design is expressed using Version 2 of the SBOL [91]. `SBOLDesigner` is an intuitive sequence editor tool that is incorporated into `iBioSim` as a plugin. The structural layer of genetic designs can be viewed and created hierarchically in `SBOLDesigner`'s canvas. `SynBioHub` is a repository for synthetic biology designs that allows storing and sharing genetic designs represented in SBOL. This feature facilitates model-based design of genetic circuits by providing the means to construct new designs from existing modeled parts.

Model Generation: The `Virtual Parts Repository` (VPR) model generator is used to obtain *interaction* data, as described in [92, 93], from the `SynBioHub` to add functional information to the SBOL description [94]. For example, it adds the proteins that act as *transcription factors* for the *promoters*, as well as their *coding sequences* in the DNA-level design. These *protein components* are coupled with the *DNA components* constructed by `SBOLDesigner` along with their interactions into functional *module definitions*. Next, an SBOL to SBML converter [95] can be applied to translate the structural and functional information of the corresponding SBOL into a quantitative model expressed in the SBML Level 3 Version 2. Since SBOL is used to represent qualitative models, the quantitative information required by SBML is inferred. However, this SBML model can then be further refined and model parameters added using `iBioSim`'s model editor. Any changes made can be mapped back to SBOL using the SBML to SBOL converter [96].

Analysis: `iBioSim` supports simulation of SBML models using a variety of different simulation methods, such as ODEs and stochastic simulation. This list also includes the hierarchical simulation, arrays simulation, and hybrid simulation methods described in the following chapters of this dissertation. Since one of the goals of `iBioSim` is to use standards for the interoperability between tools, the SED-ML standard is integrated into `iBioSim`. Each `iBioSim` project is associated with a single SED-ML file, where each analysis corresponds to a single task that is used to specify how a model should be analyzed (e.g., which simulator to use) and how the results are presented to the user (e.g., how the output plot should look like). The SBOL document, the SBML model, and the SED-ML file along with results of analysis can be collected within a COMBINE Archive and uploaded to

SynBioHub.

Synthesis: While the workflow shown in **Figure 2.8** on page 27 requires manual selections of parts for a genetic design, iBioSim also supports automated methods for part selection leveraging a process called technology mapping [97]. Rather than derive a model from manually composed parts, this process derives a genetic combinational circuit design from a given SBML model by automatically selecting parts to implement the model's specified function. The key challenge that has to be addressed is that the parts selected must not interfere with each other. Namely, there should be no unintended interactions between the proteins produced by each portion of the design.

CHAPTER 3

HIERARCHICAL SIMULATION

Many tools have been designed to model and simulate models at the molecular level. However, it is of interest to many scientists to have the ability to represent models at the population level since there are applications in which population modeling is used [98, 99]. While there are tools for modeling and simulation of cellular populations, the field still lacks a standard-enabled workflow for the modeling and simulation of population-based models. Expressing such models in a standard fashion is crucial, but it becomes meaningless if there is no efficient simulation methods capable of handling such complexity. Traditional methods flatten out the hierarchical constructs of population models, which causes the state space of the model to grow quickly. For that reason, a new stochastic simulation algorithm is developed. This fact motivated the development of the hierarchical simulation method, which is a method that takes advantage of the inherent hierarchical structure of population-based models by reusing parts.

Section 3.1 describes why hierarchy is an important abstraction for biological systems and the motivation for the development of a hierarchical simulator. Section 3.2 describes the hierarchical simulation algorithm. Section 3.3 illustrates the hierarchical method through an example. Section 3.4 presents extensions to the simulator to support additional SBML constructs, such as, rules, events, and constraints. Finally, Section 3.5 describes the results and compares the performance of the hierarchical simulator described in this chapter against simulation of flat models.

3.1 Algorithm Overview

Genetic circuits have been constructed for many applications, such as genetic timers, oscillators, and logic gates, among others. These applications can be developed in single-celled organisms. However, there are applications in which cellular population modeling is required. One example is tissue development [100], which relies on cell-cell communi-

cation for the coordination of different processes within the population.

It is natural to model cellular populations using hierarchy because cells can be defined separately and then instantiated in a top-level model. In SBML, hierarchy is represented using the hierarchical model composition package. The hierarchical model composition package in SBML is better illustrated using an example in **Figure 3.1** on the current page. Assume there is a chemical reaction network as shown in **Figure 3.1a** on this page, where a molecule of A and B are taken as the reactants of a certain reaction R1 that forms a molecule of C. In addition, a molecule of C is used to form a molecule of D through reaction R2. In this model, species A and D are put on a port, where the former is on an input port and the latter is on an output port. This chemical reaction network can be used to construct a hierarchical model as shown in **Figure 3.1b** on the current page. In this model, the top-level model contains two instances, C1 and C2, of the chemical reaction network shown in **Figure 3.1a** on this page. In addition, the top-level model has three species: X, Y, and Z. Species X replaces species A in instance C1, species Y is replaced by species D in C1 and replaces species A in C2, and species Z is replaced by species D in C2. Note that when a species in the top-level model replaces or is replaced by a species in a sub-model, the two species are effectively the same. Furthermore, reaction R2 in sub-model instance C2 is deleted from the respective model.

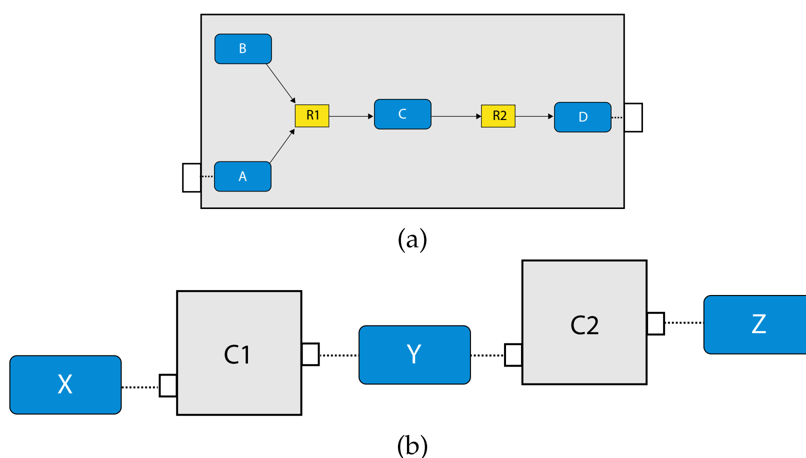


Figure 3.1: This is an illustration of how hierarchy is used for modeling chemical reaction networks. (a) A simple chemical reaction network consisting of species A, B, C, and D. Species A reacts with species B through reaction R1 to form species C. Species C is used to form species D through reaction R2. Species A and D are put on ports. (b) A hierarchical model example using the simple chemical reaction network above as a sub-model.

Dealing with the hierarchy inherent in cellular population models can be difficult because there are many dependencies that need to be handled. Therefore, it is a common practice for many modeling tools to flatten (inline) the hierarchy of a model before simulation. In other words, a typical simulator would instantiate copies of each sub-model and perform replacements and deletions during this flattening process resulting in a potentially much larger model that no longer includes any hierarchical modeling constructs. This approach has several disadvantages. First, the flattening routine causes the size of the model representation to grow quickly, consuming a lot of computational resources. Second, the flattening process itself can be very time consuming.

3.2 Hierarchical Stochastic Simulation Algorithm (hSSA)

The *Hierarchical Stochastic Simulation Algorithm* (hSSA) is a more efficient algorithm for simulating cellular populations. This method avoids the cost of flattening while preserving identical simulation results through several steps. First, in the preamble stage, the simulator locates the sub-models, $\{M_1, \ldots, M_p\}$, used by the top-level model, M_0. The simulator, however, only stores in memory one copy of each unique type of sub-model. The state of the simulator is now a vector of state vectors (i.e., $\mathbf{x} = \langle \mathbf{x}^0, \ldots \mathbf{x}^p \rangle$ where $\mathbf{x}^i$ is the state corresponding to model M_i).

The SSA is modified as shown in **Algorithm 3.1** on the next page to support hierarchical simulation. Structurally, the algorithms are similar. The main difference is the introduction of ν to indicate the model for the reaction to be executed. Since there is only one copy of each unique sub-model stored in memory, the key challenge is that replacements and deletions must be performed on the fly during simulation making each step a bit more involved. In the description of the algorithm, the notation $replaces(S_i^k, S_j^l)$ is used to indicate that species S_i^k in model M_k replaces species S_j^l in model M_l, and the notation $delete(R_j^k)$ indicates that reaction R_j^k is to be deleted from model M_k.

In the hSSA method, replacements must be considered when determining the initial state which is accomplished with **Algorithm 3.2** on the following page. First, the initial state vector is set to the initial value defined within each model. Next, each state in the top model, x_i^0, must be updated to take the value, x_j^k, of the initial state of a species S_j^k when that species is specified to replace the top-level species S_i^0. Finally, the algorithm updates

Algorithm 3.1: Hierarchical SSA

```
1  Input: Hierarchical reaction model, M = ⟨M_0, ..., M_p⟩;
2  Output: Time series simulation, α;
3  α := ⟨⟩;
4  ⟨t, x⟩ := initialize(M);
5  while t < timeLimit do
6      α := α · ⟨t, x⟩;
7      ⟨a, a_0⟩ := computePropensities(M, x) ;
8      τ := computeNextReactionTime(a_0);
9      ⟨ν, μ⟩ := selectNextReaction(a, a_0);
10     ⟨t, x⟩ := ⟨t + τ, x + v_μ⟩;
11     performReplacements(M, x, reactants(R^ν_μ) ∪ products(R^ν_μ));
```

Algorithm 3.2: initialize(M)

```
1  x := ⟨x^0_0, ..., x^p_0⟩;
2  for k := 1 to p do
3      for i := 1 to n^0 do
4          for j := 1 to n^k do
5              if replaces(S^k_j, S^0_i) then
6                  x^0_i := x^k_j;
7  for k := 1 to p do
8      for i := 1 to n^0 do
9          for j := 1 to n^k do
10             if replaces(S^0_i, S^k_j) then
11                 x^k_j := x^0_i;
12 return ⟨t_0, x⟩;
```

any species in a sub-model which is replaced by a species at the top-level. These steps are necessary to ensure that the states of species involved in replacements coincide initially.

Deletions are considered when evaluating reaction propensities. Namely, in **Algorithm 3.3** on the next page, the propensity of a deleted reaction is set to zero, so it does not participate in the simulation. The total propensity calculated, a_0, is the sum of the propensities for all the non-deleted reactions in all models.

Computing the next reaction time is the same as for the original SSA, but the computation of the next reaction is modified as shown in **Algorithm 3.4** on the following page. Namely, the sum must be over all reactions in all models, and return both the model, M_ν, and the reaction, R_μ^ν, in this model to execute.

Finally, the current time is advanced to the next time step and the state of the model M_ν is updated as a result of the reaction R_μ^ν. Reactants and products of the selected reaction that are involved in replacements must have their state updated in order to ensure that the values of these species continue to coincide throughout simulation. Namely, **Algorithm 3.5** on the next page is passed a set of species which have been updated. For each of these species, y_δ^μ, if this species is not from the top-level model (i.e., $\mu \neq 0$), then it must check if this species is involved in a replacement with a top-level species, y_i^0. If it is, this top-level species must be updated, and this algorithm must be called recursively to perform replacements on y_i^0. Otherwise, if this is a top-level species (i.e., $\mu = 0$), then it must check if it is involved in a replacement for any species at a lower-level. If it is, then this species must be updated to take this value.

3.3 Example

To better illustrate how the algorithm works, the hierarchical model shown in **Figure 3.1b** on page 31, which is described in Section 3.1 is going to be used as an example. In hSSA, the input is a collection of models where M_0 represents the top-level model, and there are p sub-models. In this particular case, there are two sub-models where M_1 represents instance C1 and M_2 represents instance C2. Both C1 and C2 are instances of the model shown in **Figure 3.1a** on page 31. The output is a times series simulation α, which is set to be initially empty.

The first step in the initialization process is to set initial time to be equal to zero and

Algorithm 3.3: computePropensities($M, \mathbf{x}$)

1 $a_0 := 0;$
2 **for** $l := 0$ **to** p **do**
3   **for** $j := 0$ **to** m^l **do**
4   **if** $delete(R_j^l)$ **then**
5             $a_j^l := 0$
6   **else**
7             $a_j^l := k_j^l \prod_{i=0}^{n^l} (x_i^l)^{v_{ij}^{rl}}$
8         $a_0 := a_0 + a_j^l$
9 **return** $\langle \mathbf{a}, a_0 \rangle;$

Algorithm 3.4: selectNextReaction($\mathbf{a}, a_0$)

1 $a := 0, r_2 = uniform(0, 1);$
2 **for** $k := 0$ **to** p **do**
3   **for** $j := 0$ **to** m^k **do**
4         $a := a + \alpha_j^k;$
5   **if** $a > r_2 \cdot a_0$ **then**
6   **return** $\langle k, j \rangle;$
7 **return** $\langle p, m^p \rangle;$

Algorithm 3.5: performReplacements($M, \mathbf{x}, Y$)

1 **for** $y_\delta^\nu \in Y$ **do**
2   **if** $\nu \neq 0$ **then**
3   **for** $i := 1$ **to** n^0 **do**
4   **if** *replaces*(y_i^0, y_δ^ν) $\vee$ *replaces*(y_δ^ν, y_i^0) **then**
5                 $y_i^0 := y_\delta^\nu;$
6                 $\mathbf{x}$:= performReplacements($M, \mathbf{x}, \{y_i^0\}$);
7   **else**
8   **for** $k := 1$ **to** p **do**
9   **for** $i := 1$ **to** n^k **do**
10   **if** *replaces*(y_δ^0, y_i^k) $\vee$ *replaces*(y_i^k, y_δ^0) **then**
11                     $y_i^k := y_\delta^0;$
12 **return** $\mathbf{x}$;

the state vector **x** to contain the initial amount of each species in each model as shown in row *Initial* in **Table 3.1** on the following page. The green entries represent changes in the state vector. Since the state vector is populated with the initial state of all species in the model, all of the entries are marked green. Note that the state vectors in C1 and C2 are equal since they refer to the same model definition. Once the state vectors are populated with the initial amount of each species, the simulator handles replacements. The *Before* row marks the species involved in replacements. First, the simulator handles the case where a species in a sub-model replaces a species in the top-level model. Once this step is completed, the simulator is going to handle the case where a top-level species replaces species in sub-models. In the example, the value of species X percolates down to species A in instance C1. The same holds for the case where species Y replaces A in sub-model C2. The *After* row in **Table 3.1** on the next page shows the final values once all replacements are handled.

After initialization is done, the simulator enters the loop. First, the simulator records the current state of the simulation. Then, the propensities for each reaction are calculated along with the total propensity as shown in **Table 3.2** on the following page. The next reaction and the next reaction time are computed afterwards.

From the table shown in **Table 3.2** on the next page, it is possible to notice that the only possible reaction to be selected is reaction R1 in C1 given that it is the only reaction that has a propensity greater than zero. **Table 3.3** on the following page shows the computed time for the next reaction to occur, as well as, the next reaction to be fired. The last step is to update the state of the simulation. Time is advanced to the next time step and the state vector **x** is updated based on the stoichiometry of the species involved in the selected reaction as shown in **Table 3.4** on page 38.

These steps are repeated until the current time exceeds the time limit. Assuming the current time is still lower than the time limit, another iteration is performed. First, the current state of the simulation is recorded. Then, the propensities are computed as shown in **Table 3.5** on page 38. The next step is to compute the next reaction time, which, in this case, is 0.2. The next reaction selected is R2 in sub-model C1. Once the next reaction is selected, the state of the simulation is updated. Time is advanced to the next time step and the state vector **x** is updated after firing reaction R2 in C1. **Table 3.6** on page 38 shows the

Table 3.1: This table shows the initial value of the state vector of the top-level model under Top and the initial value of the state vector of the models C1 and C2 given in **Figure 3.1b** on page 31, and how replacements affect the state vector of each model. In this particular model, species Y in the top-level model is replaced by species D in C1 and the value of Y is updated accordingly. Similarly, species Z in the top-level model is replaced by species D in C2. In addition, species X is replacing species A in C1 and species Y is replacing A in C2.

	Species Amounts													
	Top				C1					C2				
	t	X	Y	Z	t	A	B	C	D	t	A	B	C	D
Initial	0	5	10	10	0	10	10	0	0	0	10	10	0	0
Before	0	5	10	10	0	10	10	0	0	0	10	10	0	0
After	0	5	0	0	0	5	10	0	0	0	0	10	0	0

Table 3.2: Propensity for each reaction and the total propensity, which is the sum of all reaction propensities.

Reaction Propensities				
C1		C2		Total
a_1	a_2	a_1	a_2	a_0
5	0	0	0	5

Table 3.3: The next reaction time is computed and the next reaction time is selected, which is a random variable drawn from an exponential distribution where the mean is the inverse of the total propensity. The next reaction to fire is selected based on the reaction propensities. That is, the next reaction is random with probability proportional to the contribution of this reaction's propensity to the total propensity.

Next Reaction		
τ	ν	μ
0.1	C1	R1

Table 3.4: Amount for all species in the hierarchical model after the first iteration. Reaction R1 is selected, where a molecule of A and a molecule of B are consumed for the production of a molecule of C. Since species A in C1 is involved in a replacement, the new value of A needs to be percolated up to X in the top-level model. Now, the value of X is 4.

Species Amounts													
Top				C1					C2				
t	X	Y	Z	t	A	B	C	D	t	A	B	C	D
0	5	0	0	0	5	10	0	0	0	0	10	0	0
0.1	5	0	0	0.1	4	9	1	0	0.1	0	10	0	0
0.1	4	0	0	0.1	4	9	1	0	0.1	0	10	0	0

Table 3.5: Propensity for each reaction for the second iteration, as well as, the total propensity. Note that reaction R2 in sub-model C1 can be fired now, since a molecule of C is produced in the last iteration.

Reaction Propensities				
C1		C2		Total
a_1	a_2	a_1	a_2	a_0
3.6	1	0	0	4.6

Table 3.6: Amount for all species in the hierarchical model after the second iteration. The total amount of C is 0 and the amount of D is 1 in sub-model C1. The state update for species D is effectively affecting the state of Y in the top-level model, and consequently, species A in C2 since Y replaces A. In the update function, the value of D in C1 is percolated up to species Y in the top-level and the value of Y is percolated down to species A in C2.

Species Amounts													
Top				C1					C2				
t	X	Y	Z	t	A	B	C	D	t	A	B	C	D
0	5	0	0	0	5	10	0	0	0	0	10	0	0
0.1	4	0	0	0.1	4	9	1	0	0.1	0	10	0	0
0.3	4	1	0	0.3	4	9	0	1	0.3	1	10	0	0

state after firing reaction R2 in C1 and handling replacements in the second iteration.

After recording the state of the simulation, the propensities are calculated as shown in **Table 3.7** on the following page. Up until this point, C2 is unable to fire any reaction. However, species A in C2 has a molecule now which enables reaction R1 to fire. The next reaction time that is 0.2. The next reaction selected in this iteration is reaction R1 in C2. Once again, the state of the simulation is updated by advancing time to the next time step and the reaction is fired. The new state is shown in **Table 3.8** on the next page.

Something interesting happens in the fourth iteration. After recording the state of the simulation, the propensities are calculated. Even though reaction R2 in C2 could, in theory, be fired since this reaction requires only a molecule of C, the propensity is zero as shown in **Table 3.9** on the following page. This is because the reaction is deleted, causing the reaction propensity to be always zero. That is, this reaction can never be fired. One final note, although in this example duplicate copies of local and top-level variables connected through replacements are shown, as a further memory saving optimization, our implementation only keeps one copy of these variables.

3.4 Extensions to hSSA to Support SBML

While the algorithm presented in Section 3.2 is limited to SBML models composed of only species and reactions, the actual implementation of our hierarchical simulator supports nearly all SBML Level 3 Version 2 core constructs, such as assignment and rate rules, events, and constraints. The modifications necessary to support these are similar to those for reactions. Namely, deleted elements are dropped from sub-models, math expressions are computed on local states, and care must be taken to ensure that top-level model and local sub-model states for variables involved in replacements must always coincide throughout simulation. **Algorithm 3.6** on page 42 shows how these features can be incorporated into hSSA, and the rest of this section describes the modifications in more detail.

In general SBML, there are five types of objects that can take a value: compartments, species, parameters, species references, and reaction. **Algorithm 3.6** on page 42 extends the state vector, $\mathbf{x}$, to take values of all of these types. Elements can have an initial assignment or be involved in an assignment rule that changes the value at the starting point of simu-

Table 3.7: Propensity for each reaction and the total propensity for the third iteration.

Reaction Propensities				
C1		C2		Total
a_1	a_2	a_1	a_2	a_0
3.6	0	1	0	4.6

Table 3.8: Amount for all species in the hierarchical model after the third iteration. In this iteration, reaction R1 in C2 is selected and, in this reaction, a molecule of both species A and B is consumed for the production of a molecule of C. Since the amount of A is changed, the value of species Y in the top-level model and species D in C1 must be updated accordingly.

Species Amounts													
Top				C1					C2				
t	X	Y	Z	t	A	B	C	D	t	A	B	C	D
0	5	0	0	0	5	10	0	0	0	0	10	0	0
0.1	4	0	0	0.1	4	9	1	0	0.1	0	10	0	0
0.3	4	1	0	0.3	4	9	0	1	0.3	1	10	0	0
0.5	4	0	0	0.5	4	9	0	0	0.5	0	9	1	0

Table 3.9: Propensity for each reaction and total propensity for the fourth iteration that illustrates deletion in hierarchical models. When a reaction is deleted from the model, the propensity is always zero.

Reaction Propensities				
C1		C2		Total
a_1	a_2	a_1	a_2	a_0
3.6	0	0	0	3.6

lation. Thus, **Algorithm 3.2** on page 33 considers whether a variable's value is determined by one of these. If so, the math is evaluated and the initial value of the object is updated accordingly. Additional SBML constructs that are not described are fast reactions, delay, algebraic rules, and rate rules.

The extended hSSA supports assignment rules, where a variable's value is associated with a math function. The algorithm for performing assignment rules is shown in **Algorithm 3.7** on the following page. This algorithm goes through each assignment rule, AR in each model. In this function, if the assignment rule AR in the model i for object j exists, and the assignment rule is not deleted, then the math associated with the rule is evaluated and the state vector $\mathbf{x}$ gets updated. Since the variable associated with the rule can participate in a replacement, replacements for this particular variable must be performed. Since assignment rules can affect the math of other rules, they need to be evaluated until there is no change in the evaluations.

Another extension to the hSSA algorithm is the support for constraints, which are terminating conditions to the simulation. In each model, there are c constraints in a set of constraints, C. Simulation ends if any constraint in C evaluates to false. At the beginning of each iteration, hSSA evaluates all of the constraints in each model that are not deleted using the function illustrated in **Algorithm 3.8** on the next page.

SBML models include a powerful discrete-event formalism, which adds much of the complexity to **Algorithm 3.6** on the following page. To support events, **Algorithm 3.9** on page 43, **Algorithm 3.10** on page 43 are needed to handle events. Two sets are introduced in these algorithms: E_U and E_T. The untriggered events are stored in the set E_U and the triggered events are stored in the set E_T. These sets are initialized using **Algorithm 3.9** on page 43. Each model i has e^i events, E^i_j. Each event is analyzed during the initialization process. If an event is deleted, the event is not evaluated since deletion on events prevents them from ever being fired. However, non-deleted events require their initial condition or trigger condition to be evaluated, where the initial condition of a certain event E^i_j is evaluated using $triggerInitial(E^i_j)$ and the trigger condition is evaluated using $trigger(E^i_j)$. All events that are initially false or the trigger condition is evaluated to false are inserted into E_U.

After initializing the model and the event sets, the hSSA needs to handle events using

Algorithm 3.6: Extended Hierarchical SSA

1 **Input:** Hierarchical reaction model, $M = \langle M_0, \ldots, M_p \rangle$;
2 **Output:** Time series simulation, α;
3 $\alpha := \langle \rangle$;
4 $\langle t, \mathbf{x} \rangle :=$ initialize(M);
5 $\langle E_U, E_T \rangle :=$ initializeEvents(M);
6 **while** $t < \mathit{timeLimit} \wedge$ checkConstraints$(M, \mathbf{x})$ **do**
7 $\quad \alpha := \alpha \cdot \langle t, \mathbf{x} \rangle$;
8 $\quad \langle \mathbf{a}, a_0 \rangle :=$ computePropensities($M, \mathbf{x}$);
9 $\quad \tau :=$ computeNextReactionTime(a_0);
10 $\quad t_R := t + \tau$;
11 $\quad \langle t_E, E_U, E_T \rangle :=$ handleEvents($M, t, \mathbf{x}, E_U, E_T$);
12 $\quad$ **if** $t_R < t_E$ **then**
13 $\quad\quad \langle \nu, \mu \rangle :=$ selectNextReaction($\mathbf{a}, a_0$);
14 $\quad\quad \langle t, \mathbf{x} \rangle := \langle t_R, \mathbf{x} + \mathbf{v}_\mu \rangle$;
15 $\quad\quad \mathbf{x} :=$ performReplacements($M, \mathbf{x}, \mathit{reactants}(R^\nu_\mu) \cup \mathit{products}(R^\nu_\mu)$);
16 $\quad\quad \mathbf{x} :=$ performAssignmentRules($M, \mathbf{x}$);
17 $\quad$ **else**
18 $\quad\quad \langle t, \mathbf{x}, E_U, E_T \rangle :=$ fireEvents($M, t_E, \mathbf{x}, E_U, E_T$);

Algorithm 3.7: performAssignmentRules($M, \mathbf{x}$)

1 **repeat**
2 $\quad \mathbf{x}' := \mathbf{x}$;
3 $\quad$ **for** $i := 0$ **to** p **do**
4 $\quad\quad$ **for** $j := 0$ **to** n^i **do**
5 $\quad\quad\quad$ **if** $(\mathit{exists}(AR^i_j) \wedge \neg\, \mathit{delete}(AR^i_j))$ **then**
6 $\quad\quad\quad\quad \mathbf{x} :=$ performAssignmentRule($AR^i_j, \mathbf{x}$);
7 $\quad\quad\quad\quad \mathbf{x} :=$ performReplacements($M, \mathbf{x}, \{\mathit{variable}(AR^i_j)\}$);
8 **until** $\mathbf{x} = \mathbf{x}'$;
9 **return** $\mathbf{x}$;

Algorithm 3.8: checkConstraints($M, \mathbf{x}$)

1 **for** $i := 0$ **to** p **do**
2 $\quad$ **for** $j := 0$ **to** c^i **do**
3 $\quad\quad$ **if** $\neg \mathit{delete}(C^i_j)$ **then**
4 $\quad\quad\quad$ **if** $\neg C^i_j(\mathbf{x})$ **then**
5 $\quad\quad\quad\quad$ **return** false;
6 **return** true;

Algorithm 3.9: initializeEvents(M)

1 $E_U := \emptyset$;
2 $E_T := \emptyset$;
3 **for** $i := 0$ **to** p **do**
4 **for** $j := 0$ **to** e^i **do**
5   **if** $\neg\, delete(E^i_j) \wedge (\neg\, trigger(E^i_j) \vee \neg\, triggerInitial(E^i_j)))$ **then**
6    $E_U := E_U \cup \{\langle i, j\rangle\}$;
7 **return** $\langle E_U, E_T\rangle$;

Algorithm 3.10: handleEvents($M, t, \mathbf{x}, E_U, E_T$)

1 $t_E := \infty$;
2 **for** $i := 0$ **to** p **do**
3 **for** $j := 0$ **to** e^i **do**
4   **if** $delete(E^j_i)$ **then**
5   **continue**;
6   **if** $\{\langle i, j\rangle\} \cap E_U \neq \emptyset$ **then**
7   **if** $trigger(E^i_j)$ **then**
8     $t_F := t +$ delay(E^i_j);
9   **if** $t_F < t_E$ **then**
10      $t_E := t_F$;
11     $E_T := E_T \cup \{\langle t_F, i, j, \mathbf{x}\rangle\}$;
12     $E_U := E_U - \{\langle i, j\rangle\}$;
13   **else**
14   **if** $(\neg\, trigger(E^i_j) \wedge \neg\, persistent(E^i_j))$ **then**
15     $E_T :=$ removeEvent(E_T, i, j);
16     $E_U := E_U \cup \{\langle i, j\rangle\}$;
17   **else**
18     $t_F :=$ getNextEventTime(E_T, i, j) ;
19   **if** $t_F < t_E$ **then**
20      $t_E := t_F$;
21 **return** $\langle t_E, E_U, E_T\rangle$;

Algorithm 3.10 on the preceding page. The algorithm keeps track of t_E, the time of the next event scheduled to fire. **Algorithm 3.10** on the previous page loops through each event, if event (E_j^i) is deleted, then the algorithm does nothing. Otherwise, the algorithm checks if the event becomes enabled and ready to fire. An event is scheduled to fire when its trigger condition is evaluated to true and the trigger condition previously evaluated to false. Thus, the function checks if event (E_j^i) exists in E_U, since the set contains events that previously evaluated to false. If event (E_j^i) is in E_U, then the trigger condition is evaluated. If the event is triggered, then the time t_F in which the event is going to be fired is calculated, where t_F is the current time, t, plus the delay associated with the particular event, where the delay is evaluated using $delay(E_j^i)$. If this event is scheduled to take place before the earliest event previously scheduled, then the next event time t_E is updated and set to be equal to t_F. The event is added to the triggered events set E_T along with the time the event is supposed to fire and the current state of the simulation. The state vector $\mathbf{x}$ is needed because events can use the values from trigger time. The event must also be removed from E_U. On the other hand, if the event (E_j^i) is not in E_U, then the event needs to be evaluated again and check if the event is still allowed to fire. That is, if the event trigger is evaluated to false and the event is not persistent, given by $persistent(E_j^i)$, then all instances of this event in the set E_T are removed from the set and the event is added to the set E_U. If the event is enabled, then the firing time of this event is retrieved using $getNextEventTime(E_T, i, j)$. If this event is scheduled to happen before the current scheduled event, then the time of the next event gets updated to this event's firing time to reflect the fact that this event takes precedence over the other evaluated events.

After handling the events, the algorithm needs to decide whether the next action to fire is a reaction or an event. In order to do so, the algorithm needs to keep track of two additional times: t_R and t_E. The former indicates the time of the next reaction and the latter indicates the time of the next event, and whichever is scheduled to happen first takes precedence over the other. If there is a reaction preceding the events, then the algorithm performs the same way as in **Algorithm 3.1** on page 33. The only difference is that the extended hSSA supports assignment rules, where a variable's value is associated with a math function. After firing the reaction, replacements must be performed, followed by assignment rules that need to be evaluated due to the change in the state of the variables

caused by the reaction.

If, on the other hand, there is an event scheduled to take place before the next reaction, all events preceding the next reaction are fired using **Algorithm 3.11** on the current page. In this algorithm, the current time is advanced to the next event time. Then, all events that are enabled are retrieved using *getEvents*, which returns a set of all events that are scheduled to fire at t_E, and this set is assigned to set F, which is a set local to the *fireEvents* function that keeps track of the events that are ready to fire. The next event to fire is selected from F using the function *getHighestPriority*, which selects event μ in model ν based on the priorities of the events scheduled to fire. This function also returns the state vector $\mathbf{x}'$, which is the state of the simulation when the event was triggered. For each object in model ν, there is a check if the selected event has an event assignment for object x_j^ν. If it does, then the math associated with the event assignment is evaluated and $\mathbf{x}$ gets updated accordingly. Since the variable involved in the event assignment can be involved in a replacement, replacements must be performed to maintain consistency of the objects. Assignment rules are performed afterwards, since the update in $\mathbf{x}$ can cause a change in the assignment rules' math function. After all the assignments rules are performed, events need to be handled again since an event assignment or assignment rule can trigger an a new event.

Algorithm 3.11: fireEvents($M, \mathbf{x}, t_E, E_U, E_T$)

```
1  t := t_E;
2  repeat
3  |  F := getEvents(E_T, t);
4  |  ⟨t_E, ν, μ, x'⟩:= getHighestPriority(M, F);
5  |  for j := 0 to n^ν do
6  |  |  if (exists(EA(ν, μ, j))) then
7  |  |  |  x := performEventAssignment(M, ν, μ, j, x');
8  |  |  |  x := performReplacements(M, x, {x_j^ν});
9  |  |  |  x := performAssignmentRules(M, x);
10 |  ⟨t_E, E_U, E_T⟩ := handleEvents(M, t, x, E_U, E_T);
11 until F ≠ ∅;
12 return ⟨t, x, E_U, E_T⟩;
```

3.5 Results

While the complexity of the algorithm from a theoretical standpoint has not changed, the hSSA provides substantial improvements in performance relative to flat simulation methods. The performance analysis of the hierarchical simulator is conducted as follows: the same simulator is executed with a model with hierarchical constructs and a flattened version of the model. Two different tests have been performed. The first test is a hierarchical model without replacements and deletions. The second test is customized to have replacements. Both tests used flattened models that have been pre-processed before simulation. The time that takes to flatten out the hierarchy is summarized in the **Table 3.10** on this page.

Tests are performed using an Intel(R) Core(TM) i5 CPU 3.50 GHz and 8GB RAM. The first test consists of a hierarchical model that is populated with repressilator sub-models without replacements or deletions and the results are shown in **Figure 3.2** on page 48. The test is performed using 1, 50, 100, 150, 200, 250, 500, 750, and 1000 sub-models. Simulation is executed for 10,000 time units with time-step of 100 time units. The total runtime results are shown in **Figure 3.2a** on page 48. The increase in runtime for hierarchical simulation is clearly more scalable than for flat simulation. Analysis of memory consumption has also been performed. **Figure 3.2b** on page 48 shows the results for the model with a population

Table 3.10: A summary of the time that takes to flatten out different sizes of the population model of repressilator circuits.

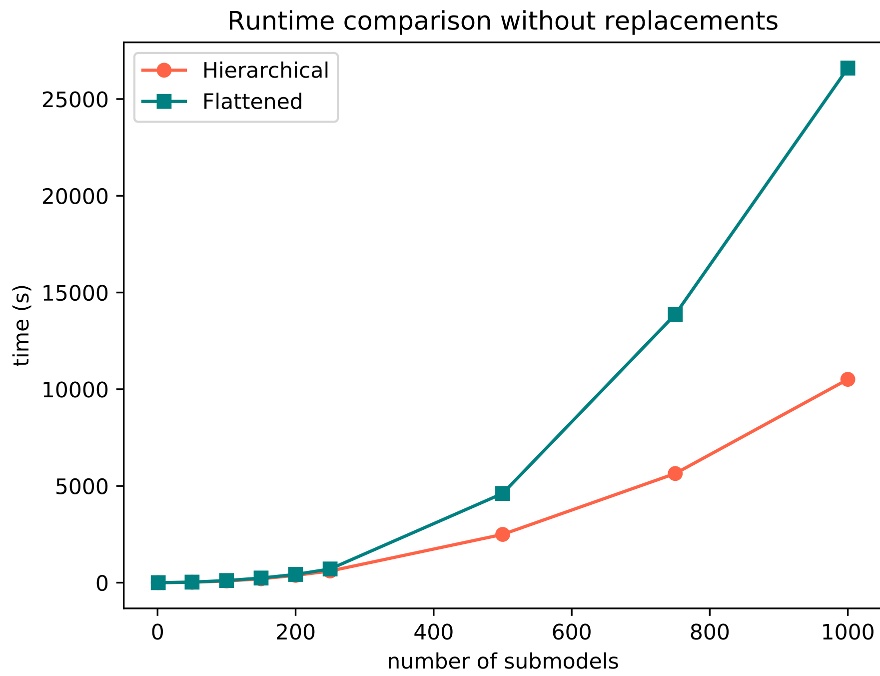
Flattening Time in Seconds		
Size	Without Replacements	With Replacements
1	0.025	0.025
50	0.388	0.382
100	0.691	0.718
150	1.168	1.304
200	1.445	1.71
250	2.23	2.802
500	6.683	7.963
750	14.995	24.493
1000	22.057	40.248

of repressilator circuits in sub-models that have no interaction between each other. The model suggests that the hierarchical simulation method takes less space over the long run compared to the flattening approach.

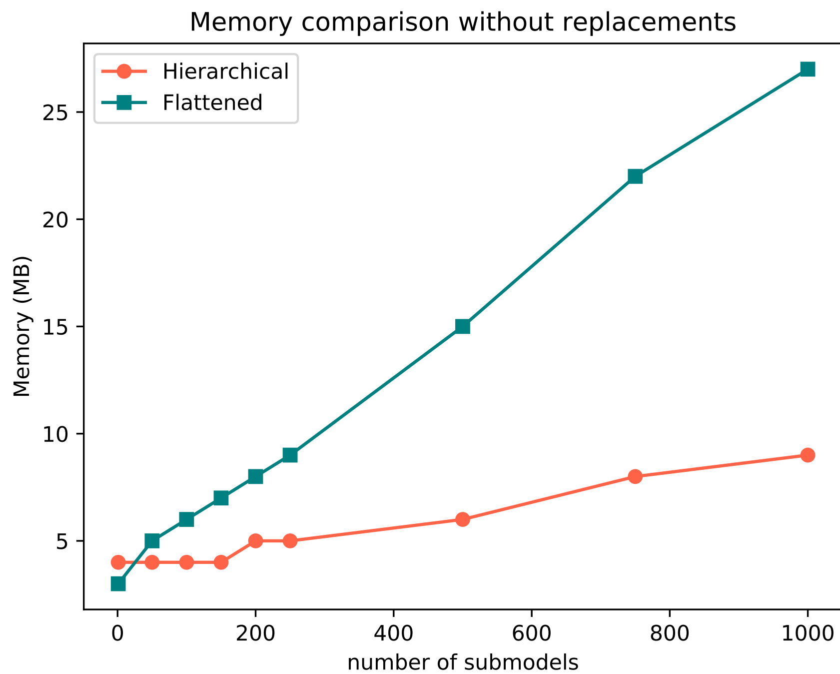
Analysis of memory consumption has also been performed. **Figure 3.2b** on the following page shows the results for the model with a population of repressilator circuits in sub-models that have no interaction between each other. The model suggests that the hierarchical simulation method takes less space over the long run compared to the flattening approach.

A second test is performed using a top-level model populated with repressilator circuits in a top-level species is added for each sub-model that replaces the GFP species in each sub-model as shown in **Figure 3.3** on page 49. An assignment rule is added to the model to compute the sum of all GFP amounts at every time point. Performance tests are performed using the same configurations as the experiment without replacements and deletions, and the total runtime results are shown in **Figure 3.3a** on page 49. These results show that the performance of the hierarchical simulator still scales much better than an SSA simulator that uses flattening even with the added complexity from replacements. The model customized connections of sub-models using replacements, hSSA still takes less space as shown in the right plot of plot of Figure **Figure 3.3b** on page 49.

The algorithm is not a direct translation of the hierarchical simulation implemented in `iBioSim`. This implementation translates SBML models into a custom data structure that is internal to `iBioSim`. Mathematical equations are translated into abstract syntax tree (AST). Variables in the model are named nodes. Since sub-models can be instantiated from the same model definition, ASTs can be reused. The only difference is the state of the variables in each sub-model. Hence, named variables in an AST become vectors in the hierarchical simulator. This concept is illustrated in **Figure 3.4** on page 51. As an example, assume there is a model with four sub-models instantiated from the same sub-model as shown in **Figure 3.4a** on page 51. The mathematical equations need to be duplicated when the model is flattened out. Duplicated trees are avoided in the hierarchical simulator because it is aware that sub-models share the same structure. Such structural information about the model given by hierarchy helps the translation of SBML models to the internal data structure in `iBioSim` because there's no need to process models that have been already

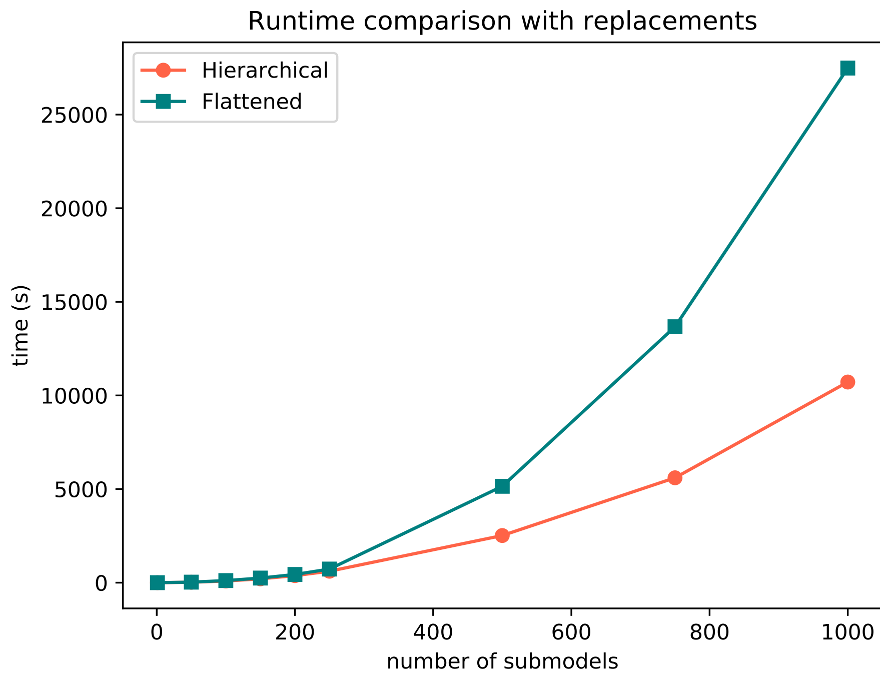


(a)

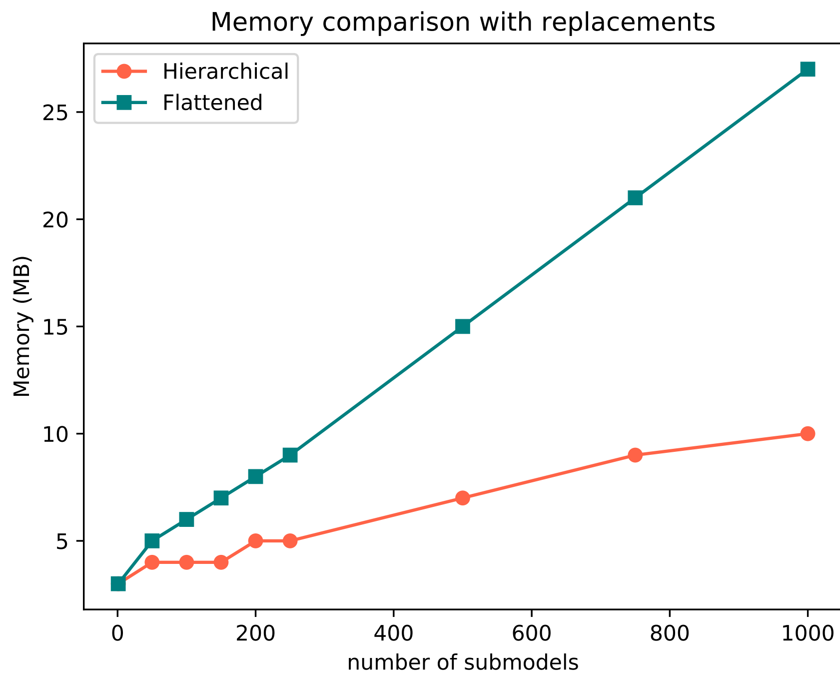


(b)

Figure 3.2: These figures illustrate the performance evaluation of the hierarchical simulation method in comparison to a flattened method for the repressilator example without replacements and deletions. (a) Comparison of runtime of SSA using flattening and the hierarchical approaches. (b) Memory comparison of the hierarchical simulator for models.



(a)



(b)

Figure 3.3: These figures illustrate the performance evaluation of the hierarchical simulation method in comparison to a flattened method for the repressilator example that includes replacements. (a) Comparison of runtime of SSA using flattening and the hierarchical approaches. (b) Memory comparison of the hierarchical simulator for models.

been processed in the setup phase of the simulation.

The extra overhead when handling replacements is avoided by updating pointers to the state object. For example, assume that the variable X in sub-model C_1 from the example shown in **Figure 3.4a** on the following page replaces the variable X in C_3. In this case, the value of X in C_3 refers to the value of X in C_1 and this is achieved by updating the reference to the state value of X in C_3 to point to the value of X in C_1 as shown in **Figure 3.4b** on the next page.

3.6 Summary

This chapter presents an efficient hierarchical simulation method for SBML models. Hierarchy is an useful abstraction that is applied to many engineering principles. Such principle also helps when reasoning about biological systems. Most SBML-compliant tools, if not all, flatten out the hierarchy before simulation. While this facilitates the implementation of simulators, models lose important structural information. As results have shown, simulation is more efficient when models are not flattened out. The algorithm presented in this chapter works for any arbitrary hierarchical SBML model.

In order to test the hierarchical simulator, an ODE simulator has been implemented. The ODE simulator shares core functionality with the presented stochastic simulation algorithm. The only difference being how the variables in the model are dynamically updated over time. The hierarchical ODE simulator has been tested using the SBML Test-Suite [101]. The hierarchical simulator does not support every feature of SBML. Test cases with the following tags have been filtered out: *CSymbolDelay*, *FastReaction*, *FastReaction*, *AlgebraicRule*, *RandomEventExecution*, tags related to any type of *ConversionFactors*, and tags related to the *fbc* package.

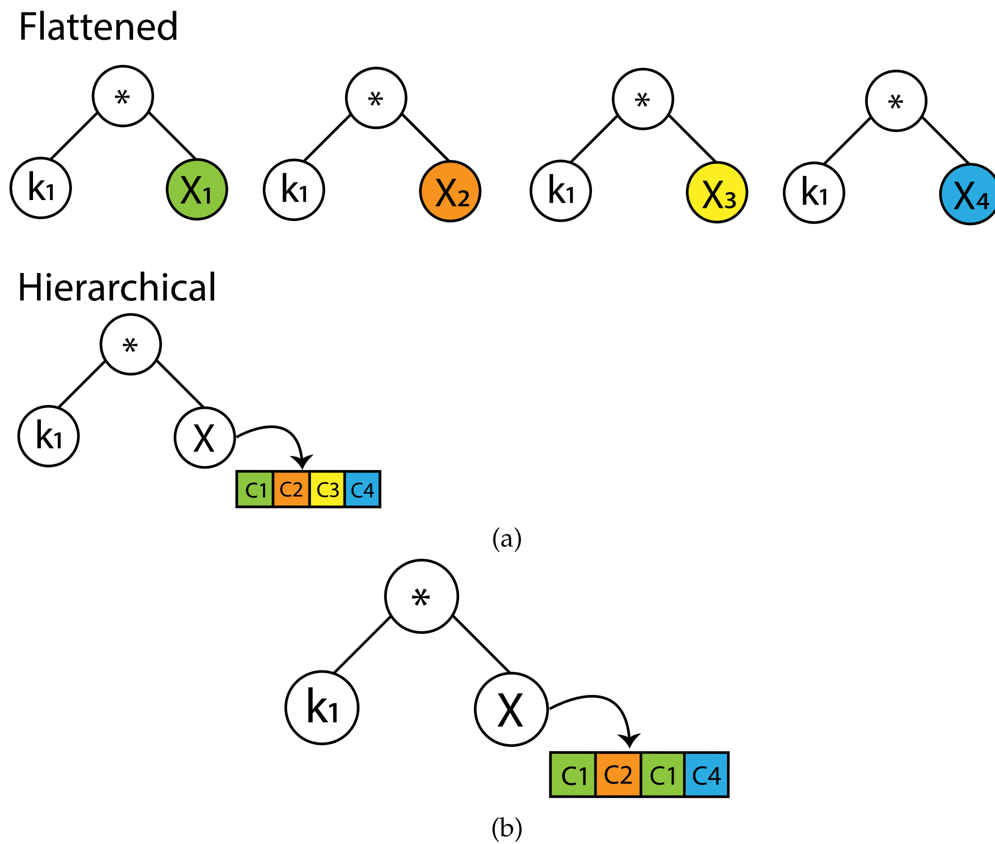


Figure 3.4: An illustration on how a hierarchical structure can be used to improve a simulator. (a) When a hierarchical model is flattened out, important structural information is lost. Common structures need to be duplicated for each sub-model. However, when the simulator is aware of the hierarchy, then common structure between sub-models can be reused. In this case, assume there is a mathematical equation in a model corresponding to $k_1 * X$. When flattening the model, a variable for X is created for each sub-model. In the hierarchical model, the mathematical equation is shared for all sub-models. The only difference is that a state variable is created for each sub-model. (b) In the hierarchical simulator, replacements are handled by updating references to the state object. If X in C_1 replaces X in sub-model C_3, then the state vector in C_3 simply references the state object in C_1.

CHAPTER 4

MODELING AND SIMULATION OF ARRAYS IN SBML

The SBML standard has been widely used for modeling biological systems. Although SBML has been successful in representing a wide variety of biochemical models, the core standard lacks the structure for representing large complex regular systems in a standard way, such as whole-cell and cellular population models. These models require a large number of variables to represent certain aspects of these types of models, such as the chromosome in the whole-cell model and the many identical cell models in a cellular population. While SBML core is not designed to handle these types of models efficiently, the proposed SBML arrays package can represent such regular structures more easily. However, in order to take full advantage of the package, analysis needs to be aware of the arrays structure. When expanding the array constructs within a model, some of the advantages of using arrays are lost. This chapter describes a more efficient way to simulate arrayed models. Section 4.1 describes how the SBML data model has been extended to support arrays. Section 4.2 describes an intuitive way to create models using arrays. Section 4.3 describes how to simulate arrayed models. This section shows how a model can be flattened for simulation and how to extend the SSA direct method to support arrays. The advantages and disadvantages of both methodologies are described. Section 4.4 evaluates the performance of the extended arrays simulation method described in this chapter and compares the performance with simulation of flattened models. Lastly, Section 4.5 gives an overview of the main impacts of using arrays in SBML.

4.1 Arrays Extension in SBML

An SBML package extension called the *arrays* package has been proposed to allow the expression of regular constructs in SBML models more efficiently. The SBML arrays package is shown in **Figure 4.1** on the following page as an *Unified Modeling Language*

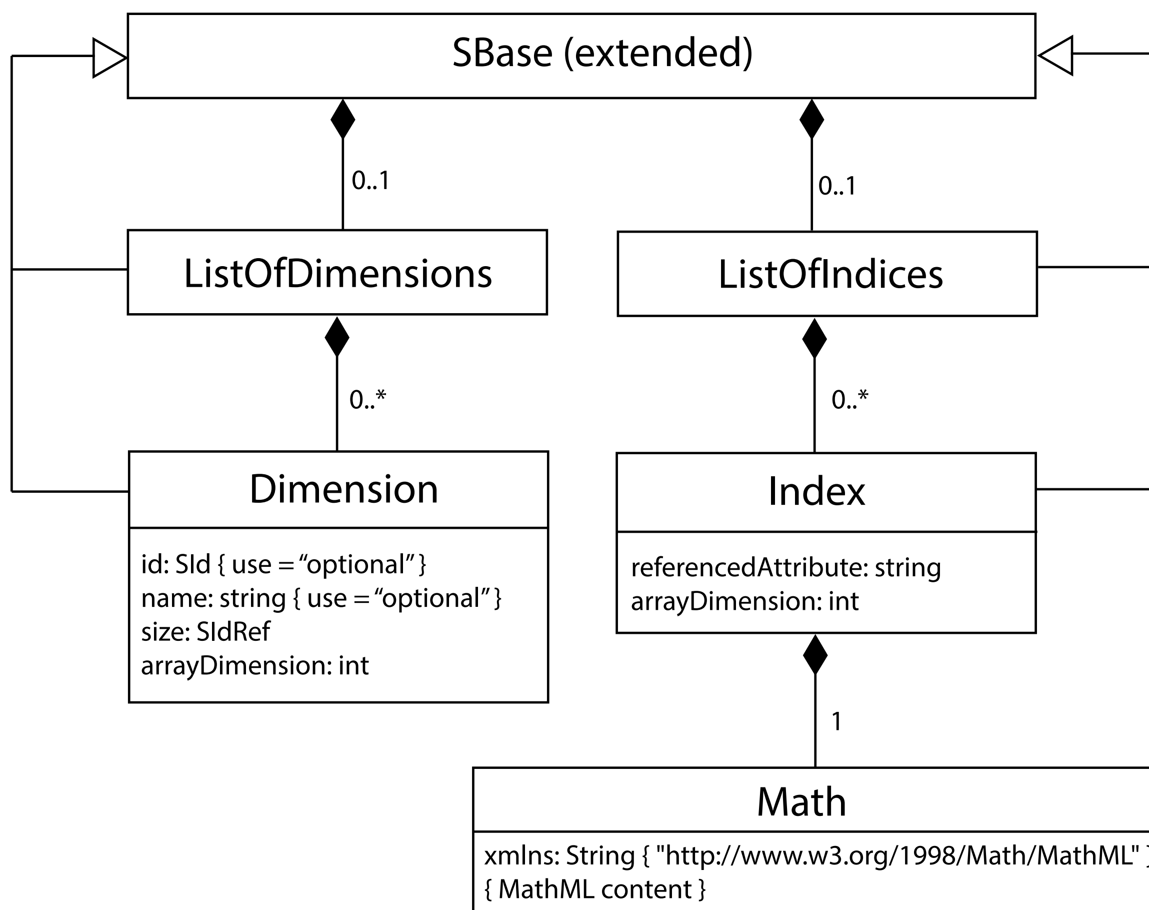


Figure 4.1: *Unified Modeling Language* (UML) diagram for data structure modifications required for the SBML Arrays package. Namely, all SBML objects inherit from the *SBase* class. This class is extended to include a *ListOfDimensions* and a *ListOfIndices*. If an object is arrayed, it has a *Dimension* object for each of its dimensions to specify its size via a constant *Parameter*. If an object has attributes that are arrayed, it can have an *Index* to indicate the specific object being referenced. This *Index* is specified using a MathML expression.

(UML) diagram. Every SBML object inherits from an *SBase* object. The *arrays* extension introduces *Dimensions* and *Indices* to SBML objects and these are explained in detail below. Further details of the package can be found in the latest specification, which can be found at: `http://sbml.org/Documents/Specifications/SBML_Level_3/Packages/Arrays_and_Sets_(arrays)`.

4.1.1 Dimension Class

A SBML object is an array when the object is given a *Dimension*. Each *Dimension* can have a *name* and *id* as optional fields and *size* and *arrayDimension* as required fields. *Dimensions* are locally scoped so different SBML objects can have *Dimensions* with the same

id. The *id* of a *Dimension* can appear in mathematical equations inside the SBML object that has this particular *Dimension*. An SBML object cannot refer to the *Dimension* of a different SBML object. The *size* field points to a non-negative constant *Parameter*. The *arrayDimension* is an integer to indicate a notion of dimensionality. That is, a one-dimensional array has a single *Dimension* with *arrayDimension* equal to 0 and a two-dimensional array has two *Dimensions* where one of them has *arrayDimension* equal to 0 and the other one has *arrayDimension* equal to 1.

4.1.2 Index Class

An object that has attributes that are arrayed need *Indices* to select single elements from an array. *Index* objects include a mathematical equation that evaluates to an integer value. This integer value corresponds to a position within an array. In order to specify the exact element being selected, *Index* objects have the *arrayDimension* and *referencedAttribute* fields. The *arrayDimension* is used to specify which dimension is being indexed and the *referencedAttribute* is used to specify the element that is being indexed (e.g. compartment, variable, and species, among others). Note that objects in an array share the same attributes, including annotations. However, annotations can be structured to make use of the index and dimension fields in a way similar to how the arrays package uses them.

4.2 Creating Models using Arrays

The *arrays* package allows an SBML model to include regular constructs more efficiently. `iBioSim` has been extended to support such array constructs. This tool provides a user interface to construct SBML models using arrays for all SBML core constructs: *Compartments*, *Species*, *Reactions*, *Parameters*, *Rules*, *Events*, *Constraints*, etc. For example, **Figure 4.2** on the next page demonstrates the idea on how to create models with arrays using the user interface in `iBioSim`. This example illustrates an array of repressilator circuits. When constructing this model, each SBML object has an optional dimension, which is indicated in the figure by square brackets enclosing the id for a constant parameter, e.g., “[” size “]”, with a non-negative integer value. In this particular case, the proteins LacI, TetR, and CI, and their respective promoters are arrays of size *n*. The index math is specified within the attributes box of each object.

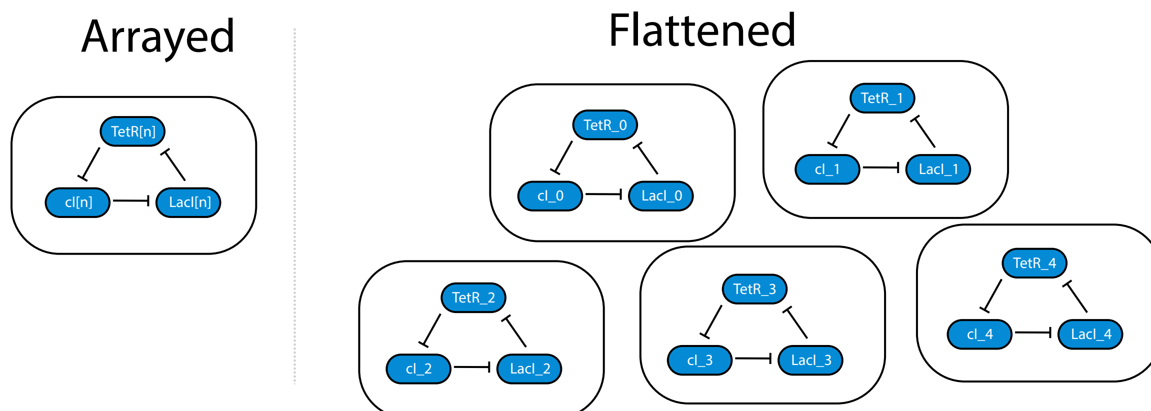


Figure 4.2: The idea on how to represent an array of repressilator circuits in `iBioSim`. The blue rectangles represent the chemical species for the proteins LacI, TetR, and CI. The black arcs represent repression of the promoter by the corresponding species. The blue arrows from promoter to species represent genetic production of the protein from the corresponding promoter. The bracketed entry indicates that each object in this model is actually an array of objects of size n.

Without arrays, the same population of repressilator circuits would have to be instantiated explicitly multiple times as shown in the flattened case. Assuming that the value of n in **Figure 4.2** on this page is 5, then the corresponding flat model would have 5 copies of the repressilator circuit.

4.3 Arrays Simulation

There are two ways to simulate models with arrays. The first way is to flatten a model before simulation. This procedure simply inlines the array elements within the model. The arrays package is just syntatic sugar for SBML models. That is, semantically equivalent models can be constructed without this extension. Therefore, an SBML document using arrays can be flattened into a new SBML document without arrays. This flattening procedure has been implemented in the Java-based library of SBML called JSBML [80].

4.3.1 Flattening of Array Constructs

The flattening algorithm is shown in **Algorithm 4.1** on the next page. The flattening routine starts off by cloning the document that is being flattened to preserve the original document. Then, the algorithm loops through each SBML element in the cloned document (e.g., *Species*, *Parameters*, *Reactions*, etc) and handles the arrayed elements appropriately.

Every object in SBML is derived from an *SBase* object. For a given *SBase* object, the

Algorithm 4.1: Arrays Flattening(D)

1 **Input:** SBML Document D;
2 **Output:** SBML Document D_{new};
3 $D_{\text{new}} := \text{clone}(D)$;
4 **foreach** SBase S **in** $D_{\textbf{new}}$ **do**
5   remove(D_{new}, S);
6     $D_{\text{new}} :=$ expandDimensions(D_{new}, S, dimensionCount(S) $- 1$);
7 Evaluate all selector and replace elements with new ids;

object is deleted from the document and **Algorithm 4.2** on the following page is called. This algorithm is used to inline arrayed objects. Starting from the highest dimension to the lowest, the algorithm dereferences the elements in the array until a scalar element is retrieved. Namely, if the current dimension *dim* is non-negative, then for each element in the array, a clone of the current *SBase* is created. The *id* of the clone is updated to ensure that new elements are created with unique ids. The *metaId* is updated in a similar fashion. Then, math associated with the current *SBase* is updated, since dimension *ids* are being replaced by their corresponding integer value. This is needed because dimensions are being deleted while new elements are being added to the document. Lastly, the algorithm makes a recursive call so lower dimensions can be expanded. If the current dimension is negative, it means the current *SBase* is a scalar and there is no dimension left to be expanded. In this case, the algorithm goes through each attribute that references an arrayed object and updates the value of the attribute by computing the respective indices and dereferencing the corresponding element in the array. Once this is completed, the element is added to the document.

The main advantage of this routine is that it eases the integration of the arrays package into existing analysis tools. A flattened model can be analyzed using any tool that support SBML core constructs. This includes a variety of different methods that are used to analyze ordinary chemical reaction networks, such as the SSA described in Section 2.3.2.

However, the flattening approach has some limitations: it restricts arrays objects to be statically computable (i.e., constant sizes), loses valuable structural information, and causes model size to grow substantially. Memory consumption of the JSBML data object for both the arrayed version of the repressilator and the flattened version are shown in **Table 4.1** on page 59 as an experiment to check the scaling limitation of flattening. Note

Algorithm 4.2: expandDimensions (D, S, dim)

```
1  Input: SBML Document D, SBase S, Dimension dim;
2  Output: SBML Document D;
   /* Handle dimensions recursively.  Repeat the process as long as
      there is a dimension that needs to be handled                    */
3  if dim ≥ 0 then
4  |  for i := 0; i < dimSize(S, dim); i := i + 1 do
   |  |  /* Create n copies of the SBML object, where n corresponds to
   |  |     the size of the current dimension being dealt with          */
5  |  |  S_new := clone(S);
   |  |  /* Update the id by appending the index of the current
   |  |     dimension to it to ensure there is no duplicated ids        */
6  |  |  updateId(S_new, getId(S_new) + "_" + dim);
7  |  |  updateMetaId(S_new, getMetaId(S_new) + "_" + dim);
   |  |  /* If the object is associated with math that is using the
   |  |     dimension id, then replace the dimension id with its
   |  |     corresponding integer value                                 */
8  |  |  replace(S_new, getDimensionId(S_new, dim));
   |  |  /* Repeat the process for a new dimension                      */
9  |  |  return expandDimensions(D_new, S_new, dim − 1);
10 else
   |  /* The base case is when dealing with scalar object with no
   |     dimensions.  In this case, each attribute referencing another
   |     SBML object needs to be handled                                */
11 |  foreach attribute with index for S do
   |  |  /* There is an index for each dimension of a given arrayed
   |  |     object being referenced                                     */
12 |  |  for i := indexCount(S, attribute); i ≥ 0; i := i − 1 do
   |  |  |  /* Get the math associated with the index                   */
13 |  |  |  indexMath := getIndexMath(S, attribute, i);
   |  |  |  /* Update the referenced object to reflect the updated id
   |  |  |     after expanding all of the arrays in the document.  In
   |  |  |     this case, the index value needs to be evaluated         */
14 |  |  |  setAttribute(S,attribute, getAttribute(attribute) + "_" +
   |  |  |   evaluateIndex(indexMath));
   |  /* Add the object to the SBML document after expanding the arrays
   |     and updating the references                                    */
15 |  addSBase(D, S);
16 |  return D;
```

that regardless of the size of the population, the SBML data object of the model consumes the same amount of memory whereas for the flat version, memory consumption grows quickly. When size is one, the flattened document becomes smaller than the one using arrays because all *Dimension* and *Index* objects are stripped away after flattening.

4.3.2 Arrays Simulation Algorithm

A better approach is to simulate on the arrayed model without changing the model [102]. An extension to SSA, where arrays are handled on-the-fly has been implemented and is described below. The SSA takes a chemical reaction network model, M, and computes a *time series simulation*, α. The simulation begins by initializing α to an empty sequence. In **Algorithm 4.3** on the following page, the initial state of the model is computed, where t denotes the current simulation time and $\mathbf{x} = \langle x_1, \ldots x_n \rangle$ denotes the state of all species. In order to handle arrays, modifications to the algorithm are introduced. The first modification is the addition of vector **r**. This vector is a multi-dimensional vector where each element corresponds to a reaction. Although the simulation algorithm implemented in this dissertation supports multi-dimensional arrays, the algorithms presented in this section assume reactions are inlined to an one-dimensional array. In addition to the reaction vector, the state vector **x** is modified and it can now take either scalar values or multi-dimensional vectors.

The simulation proceeds until the current simulation time exceeds a given time limit. For each simulation step, the current state of the simulation is appended to α. Then, the *reaction propensities*, **a**, are computed as shown in **Algorithm 4.4** on the next page. Each reaction propensity, a_j, is the propensity for reaction, R_j, which can be approximated by taking the product of the rate constant times each of the reactants raised to the power of their respective *stoichiometry*. Lastly, the total propensity is computed, which is simply the sum of all reaction propensities. In the modified algorithm, there are p reactions, where p corresponds to the number of reactions inlined. For each reaction R_j, the corresponding index values, **ind**, are retrieved. The indices are used to reference the corresponding species participating in the reaction. If the vector **ind** is empty, then the reaction is a scalar and only contains scalar species participating in the reaction. Once again, the total propensity is computed in the end.

Table 4.1: A summary of the SBML document sizes corresponding to the population of repressilator circuits before and after flattening the document.

Document Size For the Repressilator After Flattening		
Population Size	Document Size (Arrays)	Document Size (Flattened)
1	36 KB	15 KB
50	36 KB	612 KB
100	36 KB	1.2 MB
150	36 KB	1.8 MB
200	36 KB	2.4 MB
250	36 KB	3 MB
500	36 KB	6 MB
750	36 KB	9 MB
1000	36 KB	12 MB

Algorithm 4.3: initialize(M) // arrays

1 **Input:** Chemical reaction network model M;
2 **Output:** Simulation time t, simulation state vector $\mathbf{x}$, ordered list of reactions $\mathbf{r}$;
3 $t := 0$;
4 $\mathbf{x} := \langle\rangle$;
```
/* Make an ordered list for the reactions in the model */
```
5 $\mathbf{r} := enumerateReactions(M)$;
6 **foreach** m_i **in** M **do**
7 | **if** $isArray(m_i)$ **then**
8 | | $\mathbf{x_i} := expandArray(M, m_i)$;
9 | | $\mathbf{x} := \mathbf{x} \cdot \mathbf{x_i}$;
10 | **else**
11 | | $\mathbf{x} := \mathbf{x} \cdot getValue(m_i)$;
12 **return** $\langle t, \mathbf{x}, \mathbf{r} \rangle$;

Algorithm 4.4: computePropensities(M,**x**,**r**) // arrays

1 **Input:** Chemical reaction network model M, simulation state vector $\mathbf{x}$, ordered list of reactions $\mathbf{r}$;
2 **Output:** List of reaction propensities $\mathbf{a}$, total propensity a_0;
3 $p := size(\mathbf{r})$;
4 $\mathbf{a} = \langle a_1, \ldots, a_p \rangle$;
5 **for** $j = 1; j \leq p); j := j + 1$ **do**
6 | $\mathbf{ind} := getIndices(r_j)$;
7 | $a_j = k_j \prod_{i=0}^{n} getValue(x_i, \mathbf{ind})^{v_{ij}^r}$;
8 $a_0 = \sum_{j=1}^{size(\mathbf{r})} a_j$;
9 **return** $\langle \mathbf{a}, a_0 \rangle$;

The total propensity is used to determine the time until the next reaction. The next reaction time is computed using **Algorithm 4.5** on the following page. The next reaction is an exponential random variable with mean proportional to the inverse of the total propensity. This function remains unmodified for the arrayed simulation.

The next step is to select a reaction to fire using **Algorithm 4.6** on the next page. To compute the next reaction, a running sum of propensities is computed. The selected reaction is the one where the running sum exceeds the total propensity times a random number from an uniform distribution with value [0,1]. Note that the algorithm remains the same for the arrayed simulation since the reactions have been inlined before computing the propensities.

Finally, the simulation time and state of the model are updated as shown in **Algorithm 4.7** on the following page, where $\mathbf{v}_\mu$ is a vector representing the change in state due to reaction R_μ. This process repeats until the time, t, exceeds the simulation time limit. For the arrays simulation, there is a small change as shown in the algorithm. In the extended algorithm, the inlined reaction vector is used to determine how the state should be updated using the $update$ function since arrayed reactions can be selected, which can possibly refer to arrayed species. When updating the states, the fact that the state vector can have scalar and vector entries needs to be taken into account.

As shown above, the structure of the algorithm stays the same in the arrays simulation approach. However, the simulator can make smarter choices since the arrays give hints on the structure of the objects. Only one copy of each construct is necessary with the exception of variables. Variables, such as *Species*, *Parameters*, and *Compartments*, among others, need a record of the state of each member of the array. However, attributes such as constant fields, boundary condition for *Species*, and number of space dimensions for *Compartments*, among others, are stored only once since all of the arrayed objects can refer to the same parent object to look up attribute values. In addition, optimizations that are implementation-specific can be applied. Rather than using maps to store values, arrays can be used, which are more efficient. Other constructs need a record of the size of the array. When performing arrayed *Reactions*, *Events*, *Rules*, and other constructs that change the state of the simulation, the simulator iterates through each of the components of the array and performs the necessary updates. Objects that reference other arrayed objects

Algorithm 4.5: computeNextReactionTime(a_0)

```
1 Input: Total propensity a_0;
2 Output: Next time step τ;
3 x, ordered list of reactions r;
4 r_1 := getRandom();
5 τ := (1/a_0) ln (1/r_1);
6 return τ;
```

Algorithm 4.6: selectNextReaction(a_0, $\mathbf{a}$)

```
1 Input: List of reaction propensities a, total propensity a_0;
2 Output: Next reaction index μ;
3 μ := 1;
4 sum := a_μ;
5 r_2 := getRandom();
6 while sum < r_2 a_0 do
7 |   μ := μ + 1;
8 |   sum := sum + a_μ;
9 return μ;
```

Algorithm 4.7: updateState(a_0, $\mathbf{x}$, $\mathbf{r}$, μ, t, τ)

```
1 Input: total propensity a_0, simulation state vector x, ordered list of reactions r, next
  reaction index μ, simulation time t, next time step τ;
2 Output: Simulation time t, simulation state vector x;
3 v := update(r, μ);
4 return ⟨t + τ, x + v⟩;
```

need to calculate the index for each object being referenced.

4.4 Results

In order to illustrate the benefits of simulating arrayed models without flattening, comparison between the two approaches is conducted by simulating models with arrays and their corresponding flattened versions.

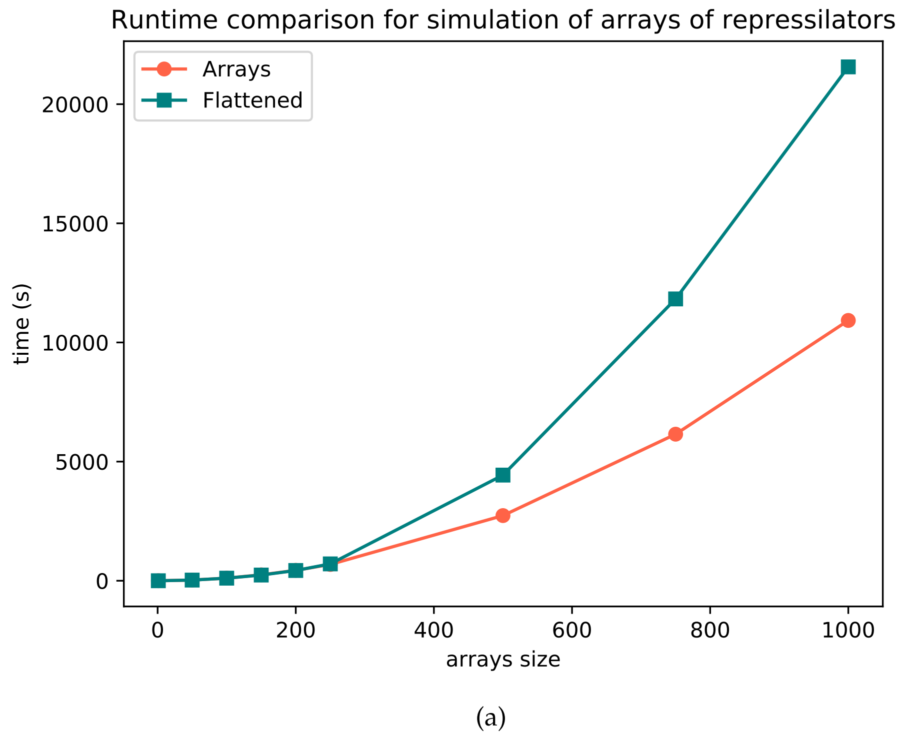
4.4.1 Repressilator

The first experiment consists of comparing the performance of simulation for cellular population of different sizes, where each cell contains the repressilator. Tests are performed using an Intel(R) Core(TM) i5 CPU 3.50 GHz and 8GB RAM. The test is performed using the following array sizes: 1, 50, 100, 150, 200, 250, 500, 750, and 1000. Simulation is executed for 10,000 time units with time-step of 100 time units. The results are summarized in **Figure 4.3** on the next page. The runtime comparison is shown in **Figure 4.3a** on the following page. For this particular example, the arrays simulator is faster because arrays allow the aggregation of reactions that helps prune the search of the next reaction to fire. That is, when finding the next reaction to fire, array reactions are considered as one reaction where its propensity is the sum of the propensities of all reactions in the array. If the next reaction to fire is an array, then the algorithm can simply search which element in the array should be selected.

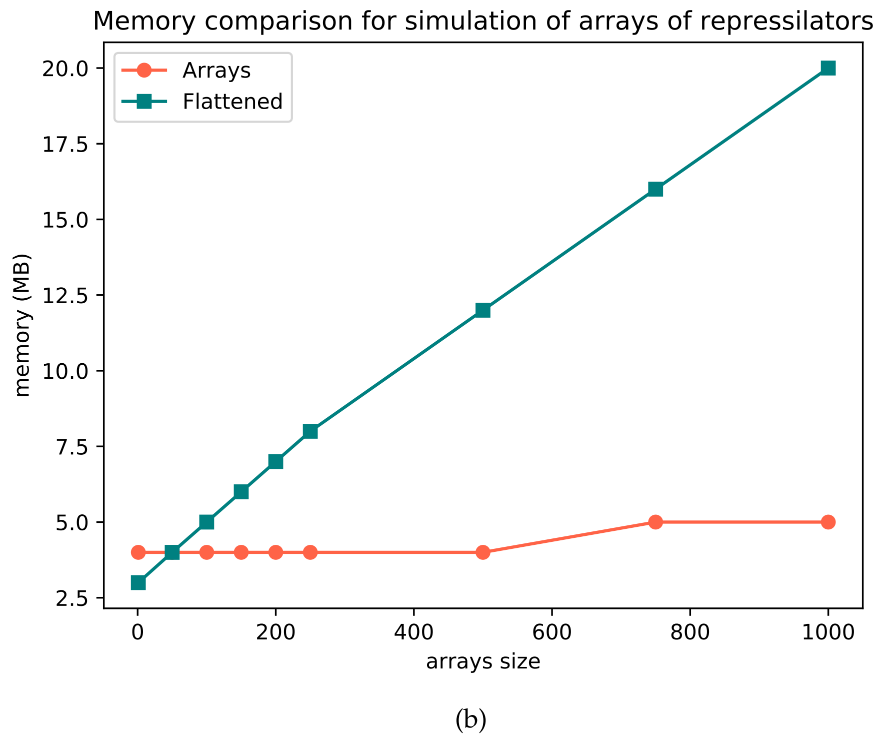
Using knowledge of the structure of the arrays, the simulator is able to condense memory usage. The method discussed in this paper reduces the memory usage substantially as shown in **Figure 4.3b** on the next page. While the flattened approach has a linear increase in the memory usage, the arrays simulator memory usage is nearly constant.

4.4.2 Genetic Toggle Switch

A population of cells with a genetic toggle switch [64] is used to evaluate the performance of the arrays simulator. An illustration of the genetic toggle switch is shown in **Figure 4.4** on page 64. In the genetic toggle switch design, there are two proteins, LacI and TetR, where they act as repressors for one another. This scheme leads to bistability. Namely, LacI and TetR can either be in a high or low state. In order to switch from a high to low state, small molecules are injected. The injected small molecule reacts with the



(a)



(b)

Figure 4.3: These figures illustrate the comparison of the arrays simulator for a population of cells that include a repressilator genetic circuit. (a) Runtime comparison. (b) Memory comparison.

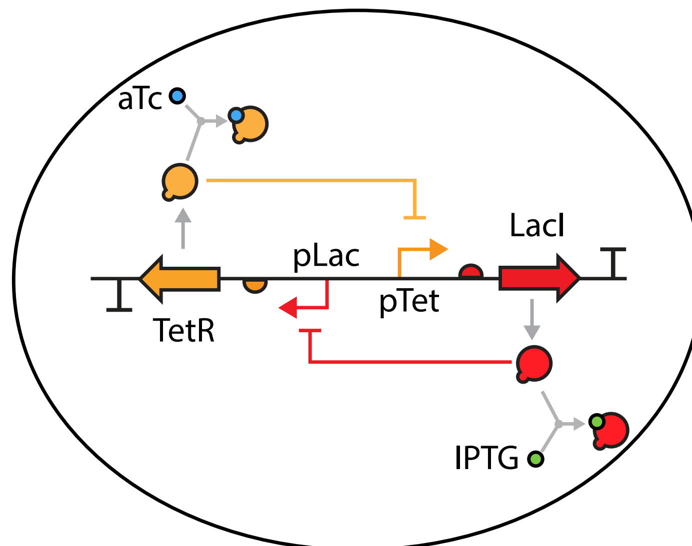


Figure 4.4: Each toggle switch includes the TetR and LacI proteins that mutually repress each other on different promoters. The model also includes complex formation reactions, indicated by dashed arrows, to represent the reactions that combine aTc with TetR and IPTG with LacI to sequester them from being able to act as repressors.

corresponding protein to form a complex, which no longer represses the production of the other protein. In this case, LacI and IPTG form a complex and TetR and aTc form another complex.

A population of cells containing the genetic toggle switch circuit is modeled as a two-dimensional array of size 10x10. **Figure 4.5** on the next page shows the result when simulating this model. Each blue line represents the LacI protein in each cell and each green line represents the TetR protein in each cell. Cells that start in the low state (i.e., LacI is high and TetR is low), remain in the high state. However, there is a small probability of a cell switching to the high state (i.e., LacI is low and TetR is high). This is indicated by the red lines in the plot where two cells switch state erroneously.

The model for the population of cells containing the genetic toggle switch circuit is simulated with the arrays simulator and compared with the flattening approach. In order to evaluate performance, the model is simulated with varying array sizes. Tests are performed using an Intel(R) Core(TM) i5 CPU 3.50 GHz and 8GB RAM. The test is performed using the following array dimension sizes: 8, 10, 12, 16, 20. Note that because this is a two-dimension array of size NxN, the model with array dimension size of 10 includes 100 copies of the genetic toggle switch. Simulation is executed for 2,000 time units with

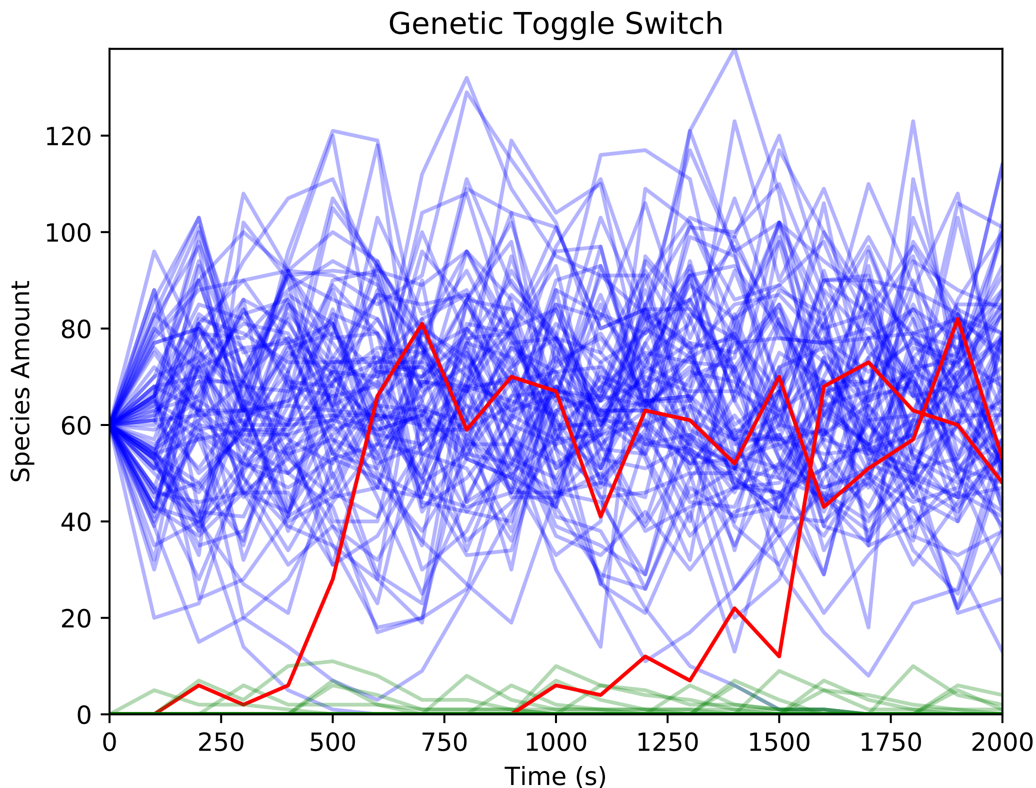
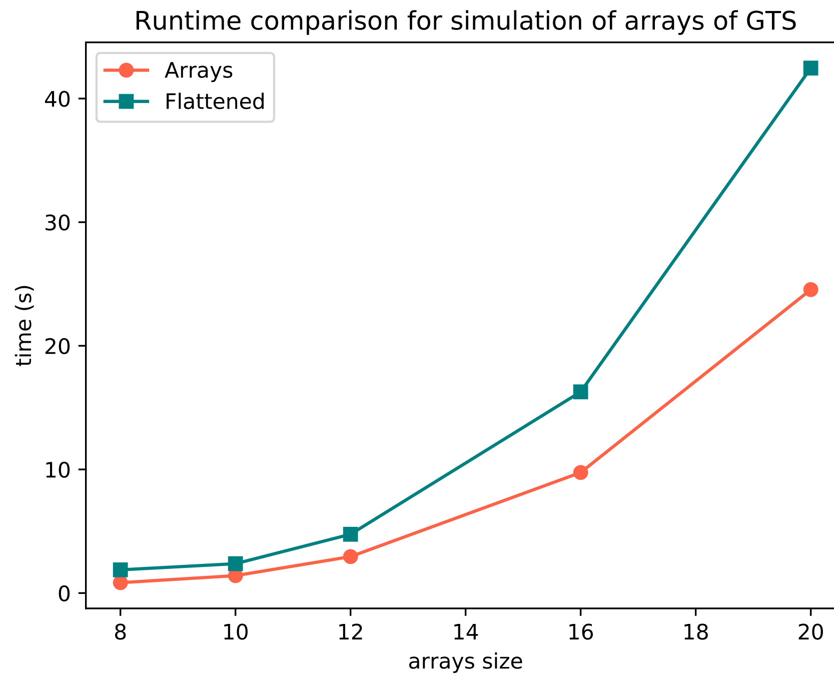


Figure 4.5: Simulation results for a population of cells that include a genetic toggle switch circuit that are not communicating. In most cases, the cells remain in the low state (i.e., LacI is high and TetR is low), but sometimes a cell changes to the high state erroneously (i.e., LacI is low and TetR is high.)

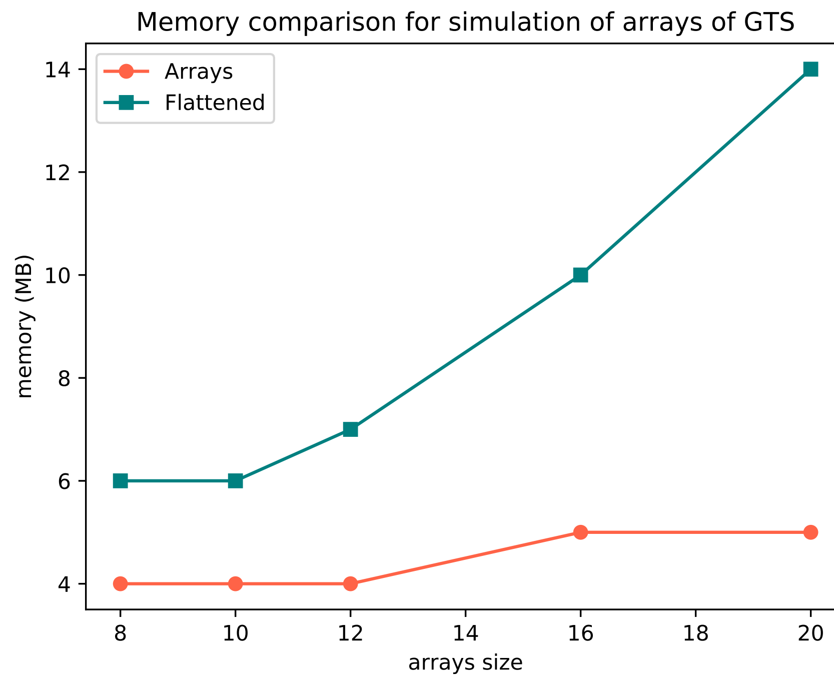
time-step of 100 time units. As shown in **Figure 4.6** on the following page, the arrays simulator is more efficient in runtime and memory. Namely, **Figure 4.6a** on the next page shows that the both methods have similar complexity but the arrays simulator is slightly faster. However, the main advantage of the arrays simulator is memory as shown in **Figure 4.6b** on the following page. The arrays simulator has nearly a constant space complexity.

4.4.3 Genetic Toggle Switch with Cell Communication

In order to create more robust genetic toggle switches (i.e. if a genetic toggle switch starts in a high state, it should stay in a high state), the genetic toggle switch model is modified by adding quorum sensing [103]. Quorum sensing is a mechanism used for cell communication as a response to fluctuation in cell-density. This is accomplished by the



(a)



(b)

Figure 4.6: These figures illustrate the comparison of the arrays simulator for a population of cells that include a genetic toggle switch circuit without cell communication. Simulation for both cases use the next reaction method rather than the direct method. Results are shown for both a flattened and arrayed model. (a) Runtime comparison. (b) Memory comparison.

sensing and secretion of signal molecules known as autoinducers [104]. Coupling genetic toggle switch and quorum sensing allows for more reliable behavior because cells would only switch state when there is consensus among neighboring cells. The genetic toggle switch coupled with quorum sensing design is shown in **Figure 4.7** on the next page. This circuit is an extension to the design shown in **Figure 4.4** on page 64. A new gene that encodes for *RhlI* is placed downstream of the *pLac* promoter. The synthase *RhlI* is used in the synthesis of an autoinducer called *C4-HSL*. Another DNA strand is added to the design, which contains a promoter induced by *C4-HSL* that encodes for *TetR*. Similarly, a different gene is placed downstream of the *pTet* promoter. This gene encodes for *CinI*. *CinI* is a synthase used for the synthesis of the *C14-HSL* autoinducer. Another DNA strand is added to the design, which contains a promoter induced by *C14-HSL* that encodes for *LacI*. Both *C4-HSL* and *C14-HSL* can diffuse through cell membrane.

In order to construct models of cell-to-cell communication, a grid-based model can be used. An example of a grid-based model is shown in **Figure 4.8** on the next page. In a grid model, relational positions are given to each entity. Each grid location is associated with a compartment that has a spatial location. Each location may include a membrane-enclosed cell. Within a cell, there may exist species that can diffuse through membranes. Namely, diffusible species can move between a cell and the environment for each grid location. Species in the environment can also move between adjacent grid locations. Such a model can be easily represented using SBML arrays. Diffusible species are created as a NxN array and placed within a NxN compartment, where each one of these compartments represents a cell. Each species is placed inside its corresponding compartment. Namely, the species at position 0,0 is placed in compartment 0,0, the species at position 0,1 is placed in compartment 0,1, and etc. In addition, a NxN compartment is created for the representation of the cell exterior. A NxN reversible reaction for membrane diffusion is created to transport species within a cell to the cell exterior at the corresponding grid location. In order to move species in the environment, two reaction arrays are added. A NxM reversible reaction is used to horizontally move species between two grid locations, where M is N-1. For example, reaction at position d_1, d_0 in the two-dimensional array is added to move species between d_1, d_0 and $d_1, d_0 + 1$. Because these are reversible reactions, there is also a reaction to move between $d_1, d_0 - 1$ and d_1, d_0. In order to move species

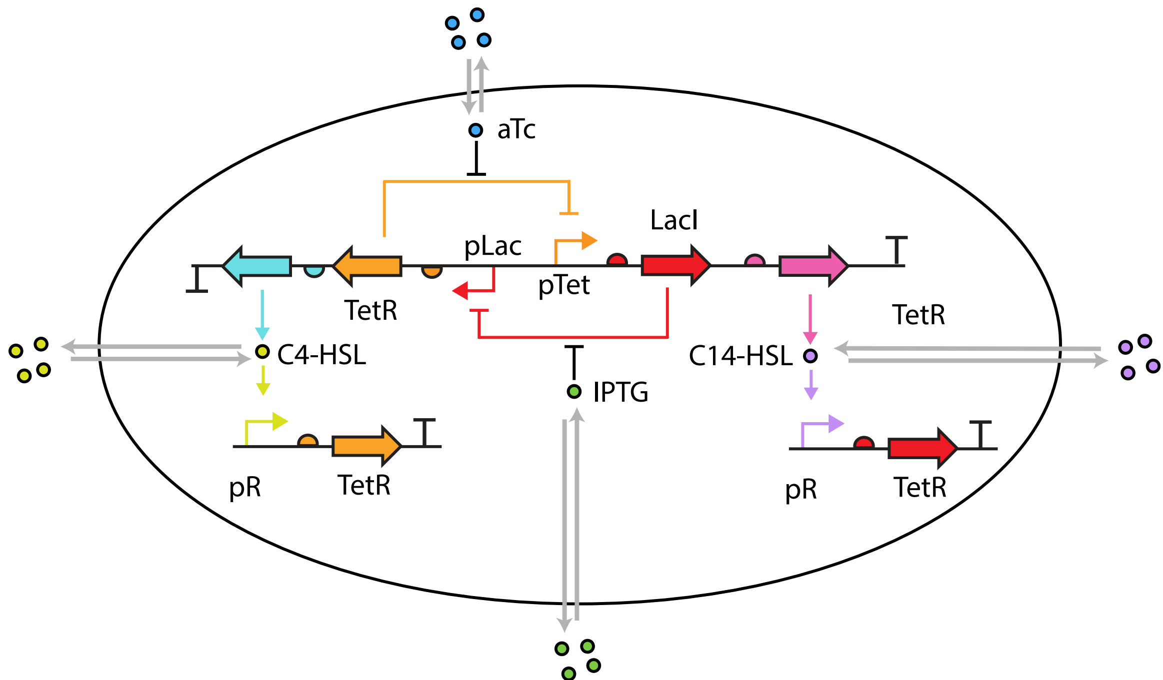


Figure 4.7: An example of a grid-based model and how it can be model using arrays.

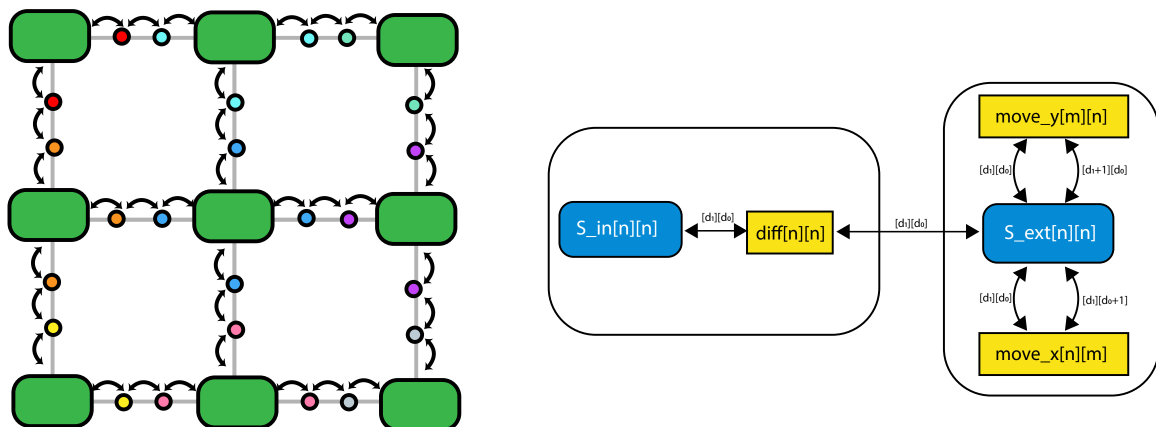


Figure 4.8: An example of a grid-based model and how it can be model using arrays.

vertically, an array of MxN reactions is added. However, in this case, species at d_1, d_0 moves to $d_1 + 1, d_0$.

The design shown in **Figure 4.7** on the preceding page is modeled as a grid-based model. This model contains a population of cells that contain the extended genetic toggle switch. Each cell is contained within a membrane-enclosed compartment that is placed in a grid location. there are reactions for moving a species from one grid location to a neighbor location. In each grid position, there may exist a cell corresponding to the genetic toggle switch circuit, where the circuits are enclosed by their own compartment. These compartments clearly distinguish the interior from the exterior of the cells. Diffusion is represented as a reaction, which is a reversible reaction that takes one species from the exterior and moves it to the interior of a cell and vice-versa. Diffusion between the cells is also modeled using additional reactions.

This model is much more complex than the model of the population of cells containing the genetic toggle switch. Using SSA direct method to simulate this model is highly unpractical because there are numerous fast reactions (e.g. diffusion and degradation) that need to fire frequently. Both the arrays method and flattened method timed out (after 5hrs) when simulating the grid-based models described above for a 6x6 population size.

4.4.4 Towards Scalable Modeling and Simulation

In order to effectively simulate this model, it is necessary to improve the performance of the arrays simulation. For each time-step, the direct method would compute the propensity of each reaction in the model. However, this is unnecessary since a single reaction is fired in each time-step. Such an approach is not scalable when simulating a model with many fast reactions. A better way to deal with such models is to use the Gibson and Bruck next reaction method [61]. **Figure 4.9** on the next page illustrates the idea of this method. The main idea behind the next reaction method is the use of a dependency graph. When a reaction fires, only the reactions that have at least one reactant affected by the fired reaction needs to be have the propensity recalculated. In this example, notice that when reaction R_1 fires, the number of molecules for S, R, and SR changes. Since reaction R_2 has SR as a reactant, the propensity of R_2 changes when SR changes. Since no other reaction is affected by S, R, or SR, the only propensities that need to be recomputed are for R_1 and R_2.

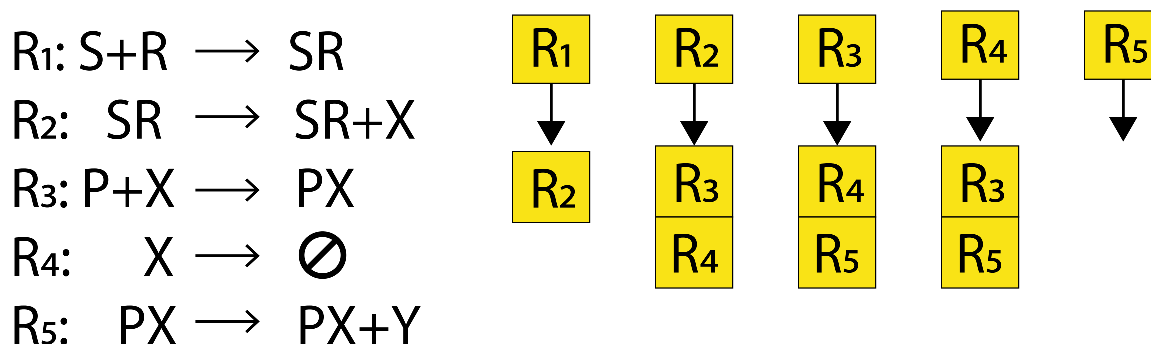
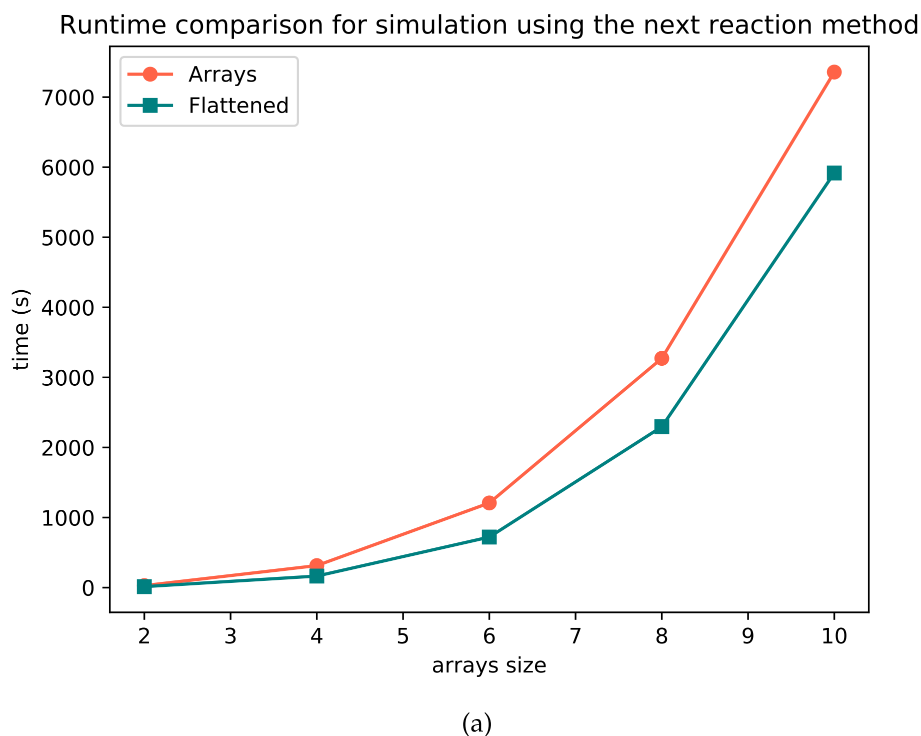


Figure 4.9: An illustration of how a dependency graph can be generated from a reaction network. As an example, assume there is a network with five reactions, $R_1, ..., R_5$. A reaction X depends on a reaction Y if reaction Y modifies at least one of the reactants of X when fired. In this example, R_2 depends on R_1 since R_1 modifies the species SR, which is a reactant for R_2. Hence, when R_1 fires, the propensity of R_2 should be recalculated.

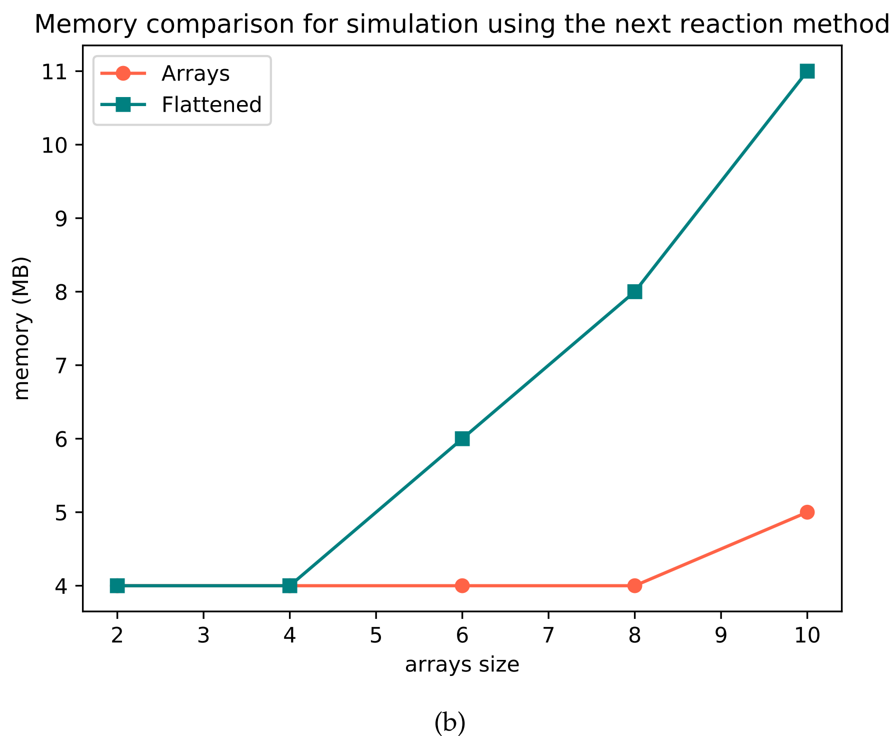
As shown in **Figure 4.10** on the following page, the grid model for the design in **Figure 4.7** on page 68 can be simulated for a population of 2x2, 4x4, 6x6, 8x8, and 10x10. The arrays simulation is slower than the flattened version because there is an extra overhead for indexing arrayed elements through selectors and index objects as shown in **Figure 4.10a** on the following page. However, the arrays simulator scales better in terms of memory as shown in **Figure 4.10b** on the next page.

While this is a significant improvement, the runtime of simulation is still highly affected by the fast reactions. In order to address this issue, model abstraction can be used. In particular, *stoichiometry amplification* is used for the diffusion reactions. Using this abstraction, the stoichiometry of the species participating in the reaction is multiplied by an amplification factor and the propensity is divided by the same amplification factor [14]. Using stoichiometry amplification and the next reaction method, the model can be simulated much faster. The arrays simulator actually performed better than the flattened version as shown in **Figure 4.11** on page 72. **Figure 4.11a** on page 72 shows that the arrays simulator is slightly faster than the approach using flattening. **Figure 4.11b** on page 72 shows a significant improvement in memory for the arrays simulator.

By simulating the models with large array sizes and performing multiple runs, the probability of a cell going into a bad state can be approximated. The results are summarized in **Table 4.2** on page 73. These results indicate that a probability of a bad state is improved by more than two-fold when the cells are able to communicate with each other.

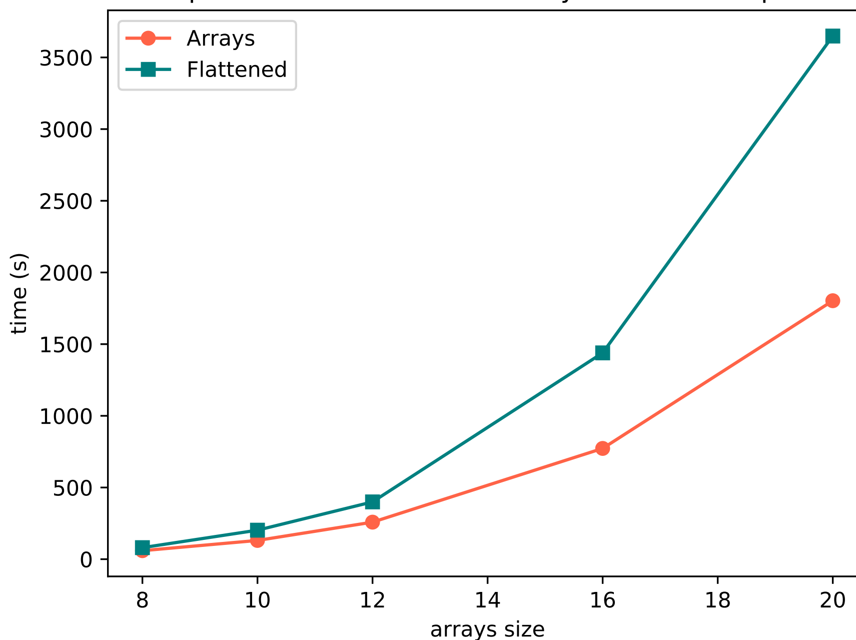


(a)

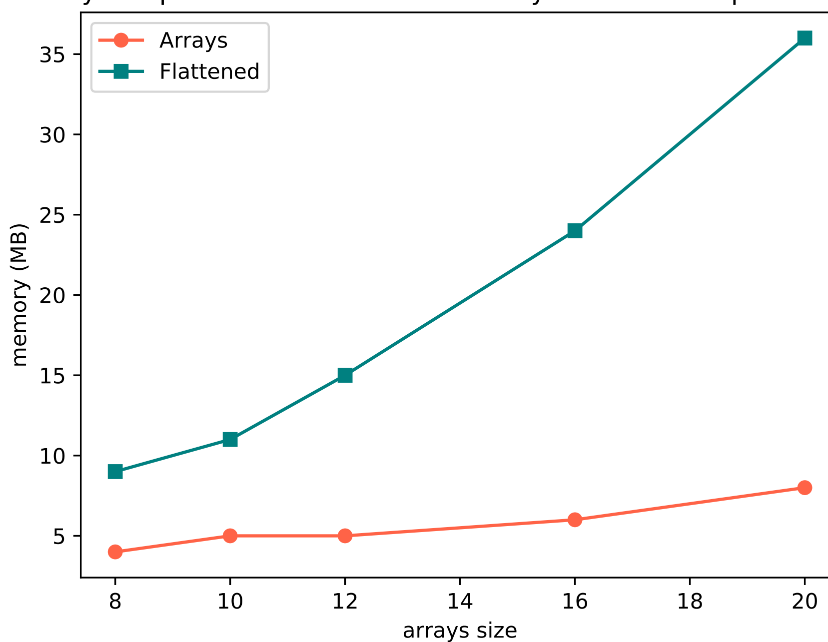


(b)

Figure 4.10: These figures illustrate the comparison of the arrays simulator for a population of cells that include a genetic toggle switch circuit coupled with quorum sensing. Simulation for both cases use the next reaction method rather than the direct method. Results are shown for both a flattened and arrayed model. (a) Runtime comparison. (b) Memory comparison.



(a)



(b)

Figure 4.11: These figures illustrate the comparison of the arrays simulator for a population of cells that include a genetic toggle switch circuit coupled with quorum sensing. Models have been modified to use stoichiometry amplification. Results are shown for both a flattened and arrayed model. (a) Runtime comparison. (b) Memory comparison.

Table 4.2: A comparison of a cell switching to the wrong state when cells communicate and when they do not communicate.

Probability of a cell going into a bad state			
	Number of Cells	Number of Failures	Probability
With Diffusion	100	0	0 %
With Diffusion	400	4	1 %
No Diffusion	100	2	2 %
No Diffusion	400	8	2%

4.5 Summary

The arrays extension is just an abstraction to the data model to represent regular structures more efficiently. All models created using the arrays package can be created with core constructs alone. However, not only would this be a tedious process, but also it would not scale, since the model would grow quickly for large population sizes. This can hinder reproducibility because large models get too large to a point where it becomes more difficult to distribute and also becomes more difficult for other to understand the model.

With the arrays package, it is simple to extend a two-dimensional grid-based model to a three-dimensional grid-based model. All of the model elements shown in **Figure 4.8** on page 68 inherits one additional dimension as shown in **Figure 4.12** on the current page. Aside from adding an extra dimension to the array objects in **Figure 4.8** on page 68, one additional three-dimensional reaction array is added to move species along the third dimension.

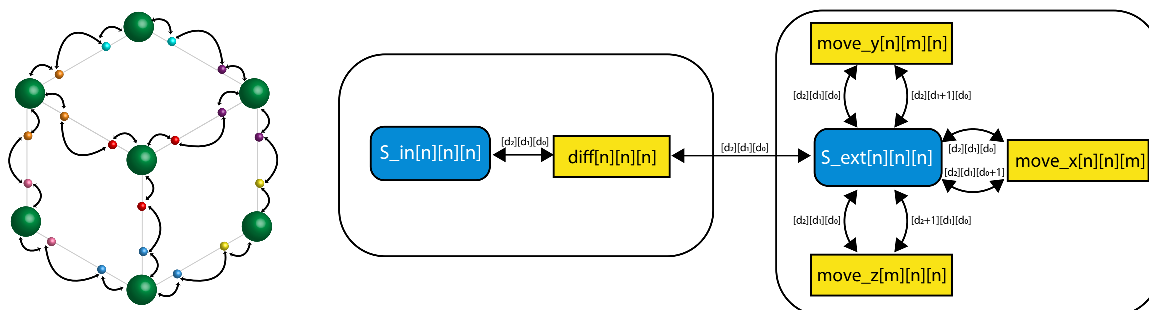


Figure 4.12: An example of a grid-based model and how it can be model using arrays.

The method presented in this section prevents the unnecessary duplication of data caused by flattening. This simulation method enables the simulation of larger models that otherwise would not fit in memory by handling array structures more efficiently. However, the method adds some overhead when retrieving the value of a certain variable within an array due to the necessity of computing indices when accessing arrayed elements. As shown in the results, using array constructs improves runtime as compared to simulating a flattened model. This is because arrays help with the selection of the next reaction to fire. That is, the simulator aggregates arrayed reactions in the reaction selection phase of the simulation and then selects which element within the selected array to fire. One of the issues of arrays is an extra overhead with indexing arrayed elements, and this can affect the overall performance of simulation.

CHAPTER 5

TOWARDS REPRODUCIBLE HYBRID MODELING

A key challenge in the modeling realm is ensuring that modeling efforts are reproducible and easily exchanged between research groups. When models are reproducible and exchangeable, results can be validated and models can be reused to build more complex ones. To achieve these goals, standard model representation formats for the model exchange, such as the SBML or CellML [105, 106], have been established. Both SBML and CellML have been successfully applied to the encoding of models using a single modeling framework, but the support of multiple frameworks adds new challenges. The first few sections in this chapter gives a high-level overview on key points that highlights the contributions of this chapter. Namely, Section 5.1 gives an overview about the different forms of modeling formalisms and how it is important to have a mechanism to couple different frameworks for complex models. Section 5.2 discusses how FBA can be extended for the dynamic analysis of biological systems through the use of dynamic FBA. Section 5.3 discusses the benefits of model reproducibility and limitations with current methods towards reproducibility. Then, Section 5.4 describes the scheme for a standard-compliant model encoding and how such models can be simulated. Section 5.5 shows the results of this work and how the models encoded using the proposed scheme can be successfully exchanged between two tools for different models. Finally, Section 5.6 summarizes the main accomplishments of the work described in this chapter.

5.1 Multi-framework Computational Models

Various simulation and analysis methods have been developed in systems biology, and depending on the biological question that the model is attempting to answer, different methods are preferred. Kinetic time-course simulation based on ODEs or stochastic methods is often employed to observe the dynamics of the entities in a model over time.

Other simulation frameworks are Boolean models [107–109], Bayesian models [54] and constraint-based approaches [69], among others.

Metabolic networks, in particular, are often challenging to model dynamically using ODE or stochastic approaches because kinetic parameters needed for such models are often unavailable [69, 110]. Hence, steady-state approaches that do not need kinetic information are employed to model metabolism, so called *flux balance analysis* (FBA) [111, 112] based on constraint-based optimization. This method only requires the connectivity of the reactions and metabolites along with the respective stoichiometry, an objective function (e.g. cell growth), and additional constraints like flux bounds. The idea is to constrain the model based on the stoichiometry of the reactions and optimize the objective function while satisfying the flux constraints. This approach computes the flux distribution at steady-state that optimizes the objective function and that satisfies the set of constraints imposed by the model. The advantages of using such method include its efficiency and not requiring any kinetic information.

Biological research questions often require the coupling of different model formalisms. One such recent example is the whole-cell model for the *Mycoplasma genitalium* [24] that is encoded using a mixture of Boolean networks, stochastic processes, differential equations, and FBA.

5.2 Dynamic Flux Balance Analysis

One disadvantage of FBA is that it cannot express the dynamics of the metabolites since it does not change species' amounts or concentrations. FBA only provides information about the optimal flux distribution for the given optimization problem. Due to this limitation, the field of *dynamic FBA* (DFBA) [110] has emerged, which couples the stationary flux distribution resulting from FBA with the kinetic update of the metabolites taken up or consumed by the FBA network. In DFBA, the FBA sub-model is coupled to a kinetic model (ODE) via a multi-framework approach.

Besides the whole-cell model which uses DFBA as a core module, many DFBA models have been constructed for different metabolic pathways. DFBA has been applied in small-scale models [110, 113, 114], medium-scale models [115–117], and up to genome-scale applications [118, 119]. For a recent overview, see Table 1 in [120].

The coupling between FBA and kinetic model parts can be implemented via three main approaches, i.e., *static optimization approach* (SOA), *dynamic optimization approach* (DOA), or *direct approach* (DA) [121]. DOA approaches discretize the simulation time and optimize simultaneously over the entire time period by solving a nonlinear programming problem (NLP). The DA approach directly includes the LP solver in the right-hand side of the ODEs. The SOA approach solves the LP at each time step using a Euler forward method assuming constant fluxes over the time step [121]. Most of the published DFBA models use the SOA approach, which is relatively simple to implement and not as computationally demanding.

5.3 Exchangeability & Reproducibility of Models

Despite the multitude of published DFBA models, currently no standard for the exchange of such models exists. Existing models are hard-coded in programming code, e.g., the whole-cell model in MATLAB. Hereby, the mathematical models in their respective formalisms are embedded in the script along with the connections between the kinetic and flux balance parts of the models. As a consequence, it is not possible to exchange existing DFBA models between different software tools. Thus, they cannot be reproduced or validated. This is especially problematic in the case of DFBA models because often multiple optima for the objective function can exist for the FBA model part (and the various time steps). Thus, the resulting DFBA solutions is not unique. The solution varies depending on the actual simulation implementation, i.e., how an implementation or solver selects one of the possible solutions. In addition, solutions can depend on the selected step size in SOA if the step size is not small enough.

While it is possible to replicate the same scripts in different programming languages, it is unpractical to do so as replication is error prone, requires unnecessary work, needs conversions that can lead to data loss, and most importantly does not solve the underlying problem of exchangeability of such models. For these reasons, replication of scripts makes achieving reproducibility difficult and often infeasible. The necessity of an exchange format for DFBA resulted from efforts trying to encode and reproduce the DFBA sub-model of the whole-cell model using standards during the whole-cell workshop [68].

5.4 Methods

A scheme is proposed for the encoding of DFBA models in SBML. SBML core is used in combination with the *hierarchical model composition* (comp) package [63] and the *flux balance constraints* (fbc) package [73] for describing the multi-framework DFBA models. The comp package is used to construct hierarchical models. It provides the means to build models from sub-models and define the interfaces between them. The fbc package is used to encode the FBA sub-model (typically consisting of the metabolic network). This sub-model provides the flux bounds for the reactions and an objective function, which are the information necessary to perform FBA. In addition, SED-ML is used to describe how each SBML model should be simulated, i.e., provide reproducible example simulation experiments by encoding which simulation algorithm to use, specifying the corresponding parameters, and defining the time course simulations for the DFBA. A COMBINE Archive is used for the exchange of the encoded models, simulation descriptions, and reference solutions. While the scheme discussed in this chapter is applied to DFBA models, the scheme is applicable to any combination of modeling frameworks.

One of the challenges in current SBML models is the limitation on the expression of models using different formalisms. Although there are several tools that support ODE simulation and FBA, they all support them independently. In order to overcome this challenge, this chapter introduces a scheme that allows the coupling of ODE and FBA models. This scheme provides exchangeability and reproducibility by encoding and simulating DFBA models in both `iBioSim` [33] and `sbmlutils` [122].

5.4.1 Stationary optimization approach (SOA)

A stationary optimization approach for DFBA was implemented as a simulation algorithm in `iBioSim` and `sbmlutils` following the simulation scheme depicted in **Figure 5.1** on the next page. The first step is the initialization of the model. All of the species and parameters in the model are initialized, where each variable's initial value is computed. After the initialization step, the FBA sub-model is executed. During the FBA step, reaction fluxes are computed using the initial flux bound values where the flux bounds for the reactions come from the top-level using SBML comp *replacements*. In SBML, as described earlier, replacements of parameters and species indicate the top-level entities are the same

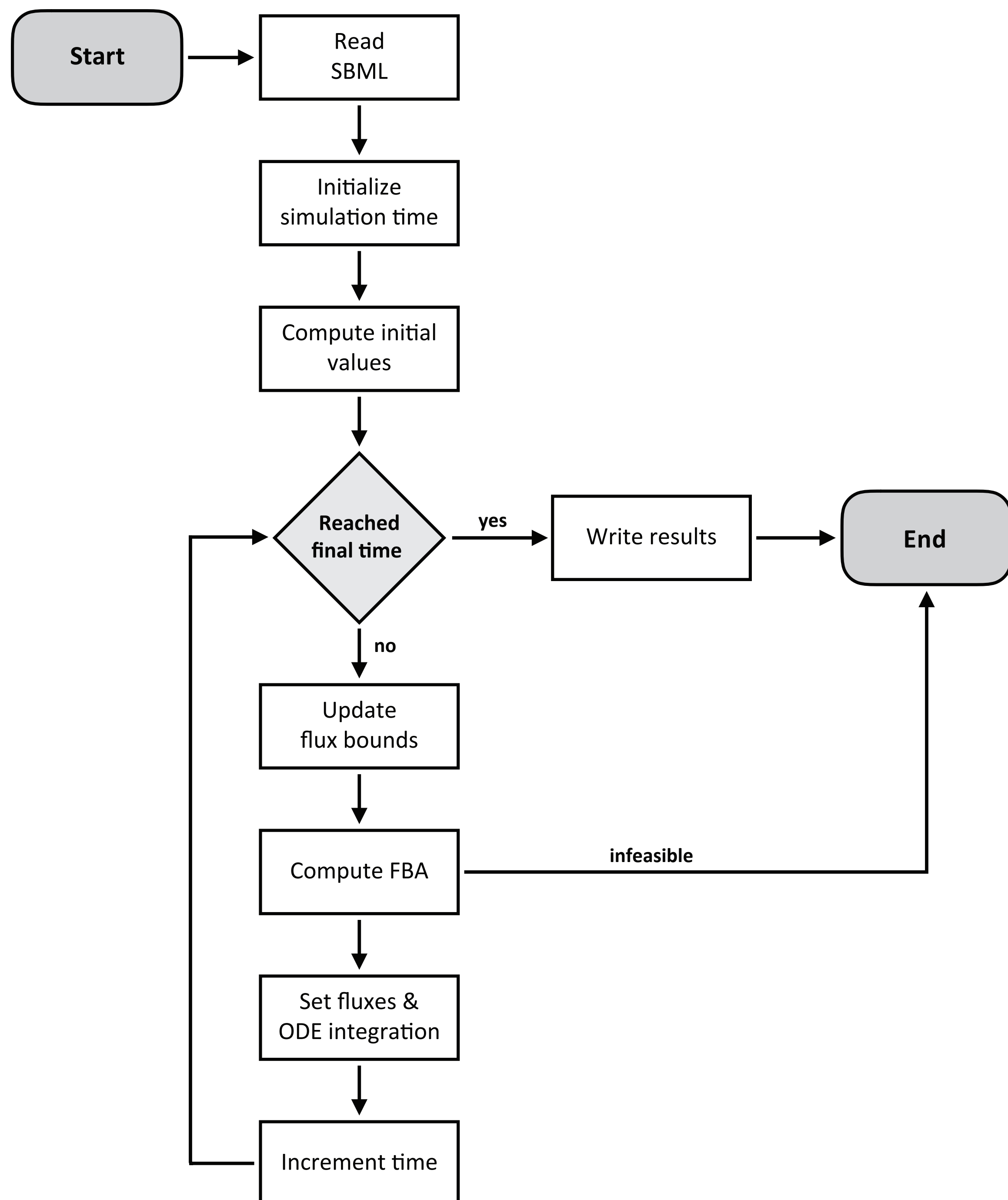


Figure 5.1: Overview of the implemented SOA algorithm for DFBA. After initialization of the model, the FBA and kinetic simulations are run in an iterative manner until the simulation end point. In every step, FBA is used to compute the reaction rates of the FBA network. Subsequently, based on the computed FBA rates, the values of the species are updated dynamically. In the SOA approach, FBA fluxes are assumed to be constant within a time step. For a detailed description see the methods section.

entity as the one being replaced. Once the fluxes are computed, they are assigned to parameters using assignment rules on the top-level. These parameters are assigned reaction rates computed as functions of the fluxes.

After computing reaction fluxes, the update step is performed concurrently with the dynamic step by computing the time-evolution of every species in the UPDATE and KINETIC sub-models. Species that affect any flux bound in the FBA sub-model are updated in the top-level. The new bounds are used in the FBA sub-model for the next time step. Simulation time is incremented at the end. If the time limit is reached, then simulation is complete. Otherwise, all of the steps above are repeated.

The SOA simulation algorithm has been implemented in `iBioSim` and `sbmlutils`. The `iBioSim` tool uses the structure of [123] for simulation. The `sbmlutils` tool uses *libRoadRunner* [124] for the kinetic simulation and *COBRApy* [125] to solve the FBA problem. Both `iBioSim` and `sbmlutils` take an SBML file that describes a DFBA model and a SED-ML file that describes the simulation experiment. In the proposed approach, SED-ML is mainly used to indicate which simulation algorithm to use, the time points in which tools should print out the values of the variables, the initial time and the time limit. The SED-ML files provide a minimal simulation experiment to check reproducibility between implementations. The value of each time increment for SOA is defined as a parameter with id *dt* in the SBML model, which can be overwritten by the SED-ML file for the actual simulation. Ontology terms for the description of DFBA simulation algorithms have been introduced in the Kinetic Simulation Algorithm Ontology (KISAO) [126] and are used in the SED-ML descriptions, i.e., `KISAO:0000500` (SOA-DFBA).

5.4.2 Model Reproducibility

In order to test interoperability based on the proposed scheme, models were built in both `iBioSim` and `sbmlutils`. Models built in `iBioSim` were imported into `sbmlutils` and vice-versa to check whether models could be interpreted by both tools consistently. This was done in an iterative manner and resulting issues were solved by clarifying the encoding scheme, e.g., by adding additional rules which resolved ambiguities.

Reproducibility of DFBA models is challenging because there may exist several possible outcomes that satisfy the objective function and constraints of the FBA models. De-

pending on how the FBA solver selects one of the multiple solutions, different trajectories can result from the DFBA simulation. The issue of multiple solutions can be solved by guaranteeing uniqueness of the solution in every time step based on *Flux Variability Analysis* (FVA) [127]. FVA gives the possible minimal and maximal fluxes for each reaction in each step of the simulation. If all minimal fluxes are equal to all maximal fluxes for a time point, then a solution is unique in the time point. If all time points are unique, then the solution is unique. As a practical note, if the solution is not unique, the addition of additional constraints to the FBA problem can make the solution unique.

Model reproducibility was tested by comparing the numerical SOA results between simulation results of the two tools for models with unique solutions. Results are assumed as numerically equivalent if the absolute difference for every time point t_k for all dynamical FBA species in the model c_k is smaller than the tolerance $\epsilon = 10^{-5}$, i.e.,

$$abs(c_i(t_k)_{sbmlutils} - c_i(t_k)_{ibiosim}) \leq \epsilon \; \forall c_i, t_k \tag{5.1}$$

In SOA-DFBA, it is important that the time step dt is small enough so that the solution converges to the correct solution. Solutions vary if selected step sizes are too large (e.g. changing the step size in the `toy_wholecell` model described later from 1.0 to 0.1 resulted in differences in steady state concentrations of up to 10%). Consequently, different step sizes were tested for the models and step size of the simulations were selected, so that smaller step sizes did not change the simulation results.

5.5 Results

The major result of this work is the creation of the first scheme for encoding DFBA in SBML. This section presents the scheme and its application to multiple DFBA models and demonstrates that the proposed multi-framework computational models can be exchanged and reproduced between tools.

5.5.1 Scheme for Dynamic Flux Balance Analysis

This chapter proposes for the first time a scheme to encode hybrid models, such as DFBA models, in SBML. The DFBA models presented in this chapter were created in the proposed scheme either using a graphical user interface in `iBioSim` or a script-based

approach in `sbmlutils`. For a given model, the *TOP*, *FBA*, *BOUNDS*, and *UPDATE* sub-models, as depicted in **Figure 5.2** on the following page, were packaged with respective simulation files using SED-ML in COMBINE Archives for the exchange between tools.

In this section, a high-level overview of the underlying concepts used in the scheme is provided, followed by application of the scheme to encode DFBA models. The DFBA model is constructed hierarchically using the SBML comp package, separating the hybrid model in different building blocks based on the respective functionality and modeling frameworks (**Figure 5.2** on the next page). The top-level model is hereby composed of four sub-models: (i) a kinetic sub-model that computes flux bounds based on the dynamic metabolite availability and ensures that the FBA problem is constrained by the available metabolite resources (BOUNDS sub-model); (ii) a FBA sub-model that encodes metabolism as a FBA problem (FBA sub-model); and (iii) a kinetic sub-model that updates the amounts and concentrations of the dynamic metabolites changed via the FBA sub-model via consumption or production (UPDATE sub-model); (iv) an optional kinetic sub-model that represents a dynamic part with all kinetics other than the metabolic pathway, such as DNA transcription, DNA translation, and protein degradation, among others (KINETIC sub-model). Alternatively, arbitrary kinetics can be part of the top model.

The top-level model ties together the three different sub-models using SBML comp *replacements* and *replacedBy* constructs with the interface between the sub-models defined via *ports*. Ports define which components within a sub-models are exposed and can be connected to components in the top-level.

In order to describe the different formalisms of each sub-model, the *Systems Biology Ontology* (SBO) is used [128]. The SBO defines controlled vocabulary terms used in the systems biology field. The SBO terms are arranged in a taxonomic hierarchy using a tree structure. This allows the grouping of terms that are related to one another. The modeling formalisms of the individual sub-models are described using terms on the *modeling framework* branch, where FBA models are described using the *flux balance framework* term, stochastic processes are described using the *non-spatial discrete framework* term, and differential equations are described using the *non-spatial discrete framework* term. Although the terms for stochastic processes and differential equations can be used for describing either stochastic or deterministic simulation methods, these terms were selected because

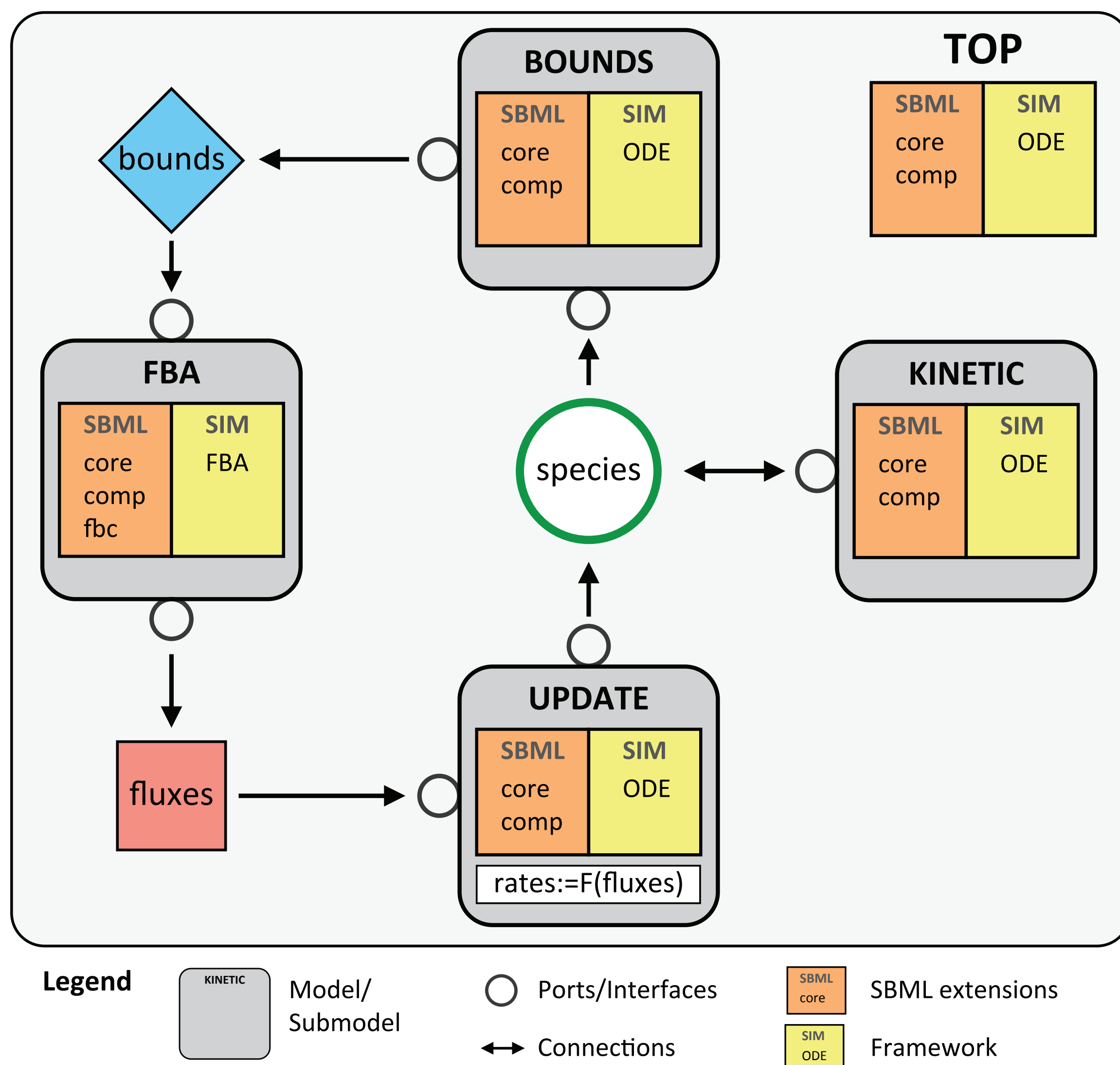


Figure 5.2: Overview of scheme for encoding DFBA models in SBML. The hierarchical SBML model is composed of a top-level model with four sub-models: FBA, BOUNDS, UPDATE, and KINETIC. The individual sub-models are connected via ports. The respective SBML packages used are listed in the models, as well as the simulation framework used. The BOUNDS sub-model calculates the upper and lower flux bounds based on metabolite availability. The FBA sub-model computes the reaction fluxes of the metabolic fbc model using the bounds as constraints. The UPDATE sub-model calculates the dynamic update of the dynamic metabolites affected by the FBA model. The rates of change are functions of the FBA fluxes. The KINETIC sub-model includes all of the other processes in the model, which may affect or be affected by entities in metabolism. The top-level model ties together the different sub-models using SBML comp *replacements* and *replacedBy* constructs.

they are the ones that best describes these two formalisms.

In addition to the modeling formalism, other key components are annotated in the sub-models via SBO terms in the scheme, e.g., the upper and lower flux bounds and the exchange reactions in the FBA sub-model defining which metabolites can be consumed or produced in the FBA part of the DFBA, or the dynamic species in the top model changed by the FBA sub-model. By the means of these annotations, the interface between the hybrid sub-models can be clearly defined.

All of the interconnections between the sub-models are encoded in SBML rather than using an external approach like for instance via SED-ML. The connections between model components are crucial information of the model and should be part of the model encoding. SED-ML is only used to encode which simulation to run with the model. As a consequence, this scheme requires only a single hierarchical SBML model and a single SED-ML file.

5.5.2 Minimal Example (`toy_wholecell`)

In order to illustrate the proposed scheme, a simplified example of a whole-cell model was created with a model overview depicted in **Figure 5.3** on the following page. The figure was created with `cy3sbml` using the SBML models [129]. This model is constructed hierarchically where a top-level model is created to instantiate different sub-models (BOUNDS, UPDATE, and FBA shown in **Figure 5.2** on the previous page) and makes the necessary connections between them. The figure illustrates the structure of each sub-model and how each sub-model ties in with each other in a flat version of the model once all of the connections are established.

In the example, the FBA sub-model imports species `A` and converts it via a linear chain of reactions to species `C`. The exchange reactions `EX_A` and `EX_C` contain the rate of consumption and production of the respective species. The TOP model contains assignment rules which assign the fluxes to the parameters `pEX_A` and `pEX_C`, which are used by the UPDATE model to update the dynamic species `A` and `C` via the update reactions `update_A` and `update_C`. The BOUNDS model calculates the bounds of all FBA exchange reactions, which are constrained by the availability of the dynamic species, as well as bounds changed by kinetic expressions. In the example, the upper bound `ub_R1` of reaction `R1` is changed via a

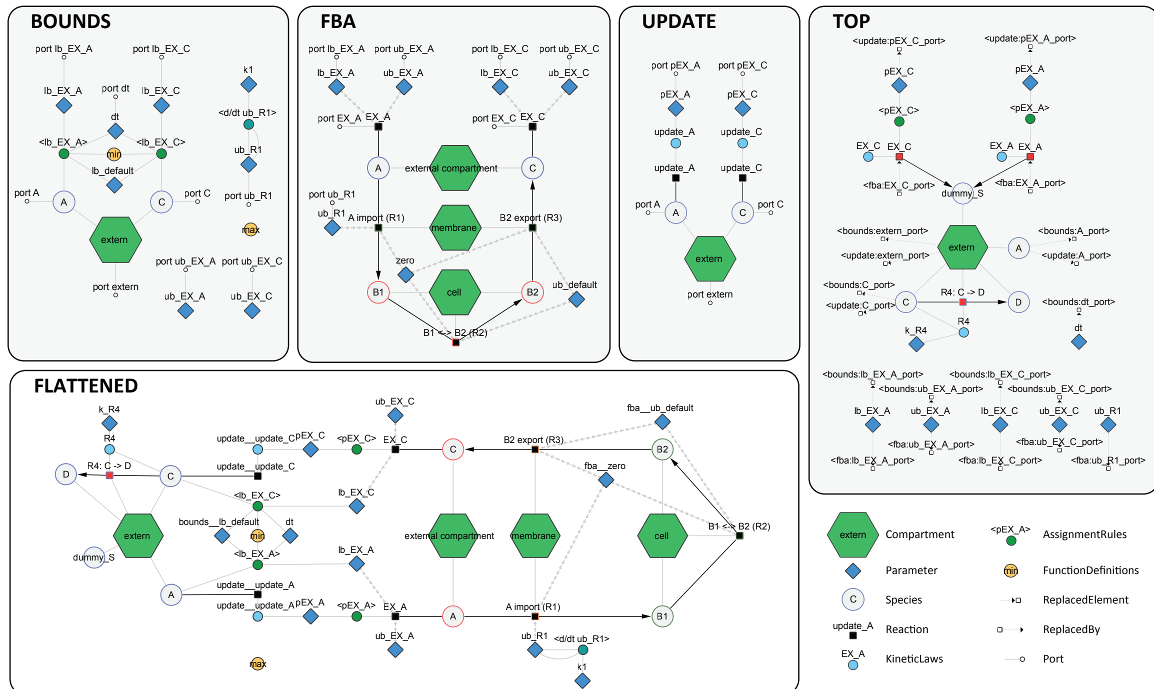


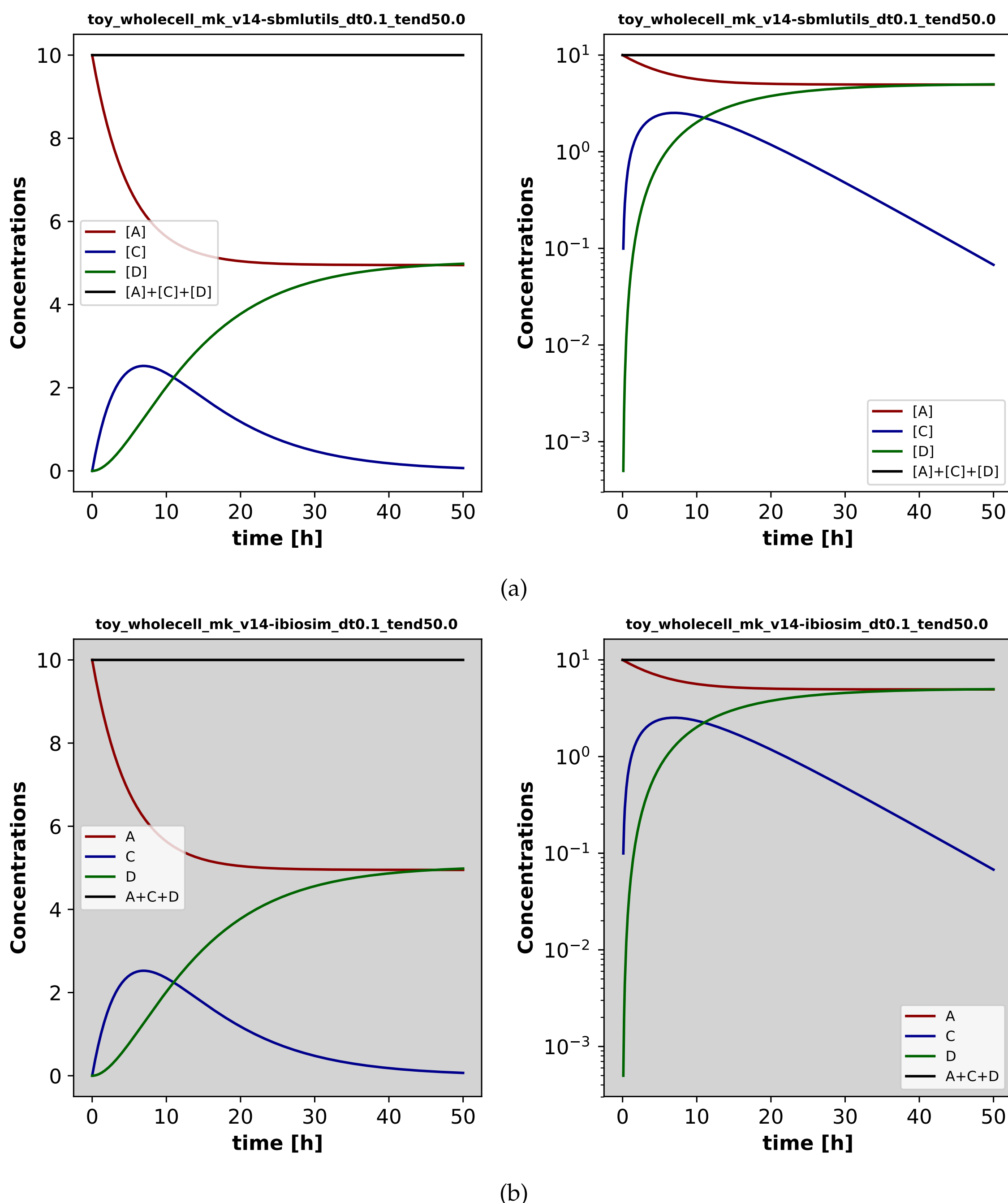
Figure 5.3: Detailed scheme of the minimal example model (`toy_wholecell`). The figure shows the components in the BOUNDS, FBA and UPDATE sub-models. Links between sub-model components are based on ports which are connected elements via TOP model replacements (replacedElements and replacedBy). The flattened SBML comp model (FLATTENED) shows the resolved connections between the different sub-models after these replacements have been performed. The flattened model cannot be simulated because the separation of frameworks is lost in the flattening process.

rate rule. Additional kinetics are encoded in the TOP model, i.e., a kinetic conversion of `C` to `C` (these could also be in a separate KINETIC sub-model).

In order to validate the exchangeability and reproducibility of the model, simulations in `iBioSim` and `sbmlutils` were performed using the simulation algorithm described in **Figure 5.1** on page 79 with results depicted in **Figure 5.4** on the next page. Both implementations resulted in numerically equivalent results (see 5.4.2). Importantly, the proposed encoding scheme allowed the replication of the numerical results to converge against the correct solution even for bigger step size than **Figure 5.4** on the following page, thereby allowing to test the effects of varying step sizes in a reproducible manner.

5.5.3 Diauxic growth in *E. coli* (`diauxic_growth`)

The next example is an encoding and reproduction of results from a published DFBA model of diauxic growth of *E. coli* [113] consisting of four reactions between four metabo-

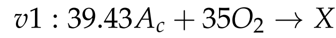


(a)

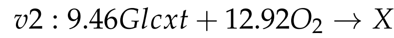
(b)

Figure 5.4: DFBA Simulation results for the `toy_wholecell` model in two different tools. This demonstrates that models can be exchanged by different tools using standards and the results can be reproduced when using the same simulation algorithm. Species `A` is converted to `C` via the FBA subnetwork over time. `C` is converted to `D` via the kinetic parts in the top model. Species `A` is not consumed completely because the import of `A` in the FBA subnetwork via `R1` is shut down via a rate rule for the upper flux bound, and a steady state is reached. The model was simulated for 50[h] with a time step `dt` of 0.1[h]. (a) Simulation results with `sbmlutils`. (b) Simulation results with `iBioSim`.

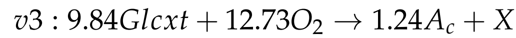
lites, i.e., glucose ($Glcxt$), oxygen (O_2), acetate (A_c) and biomass (X). The model can grow either aerobically on acetate ($v1$), aerobically on glucose ($v2$ or $v3$) or anaerobically convert glucose to acetate:



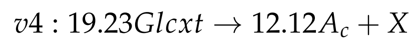
$$v1 : 39.43A_c + 35O_2 \rightarrow X$$



$$v2 : 9.46Glcxt + 12.92O_2 \rightarrow X$$



$$v3 : 9.84Glcxt + 12.73O_2 \rightarrow 1.24A_c + X$$



$$v4 : 19.23Glcxt \rightarrow 12.12A_c + X$$

The kinetic part of the model is described by the following differential equations:

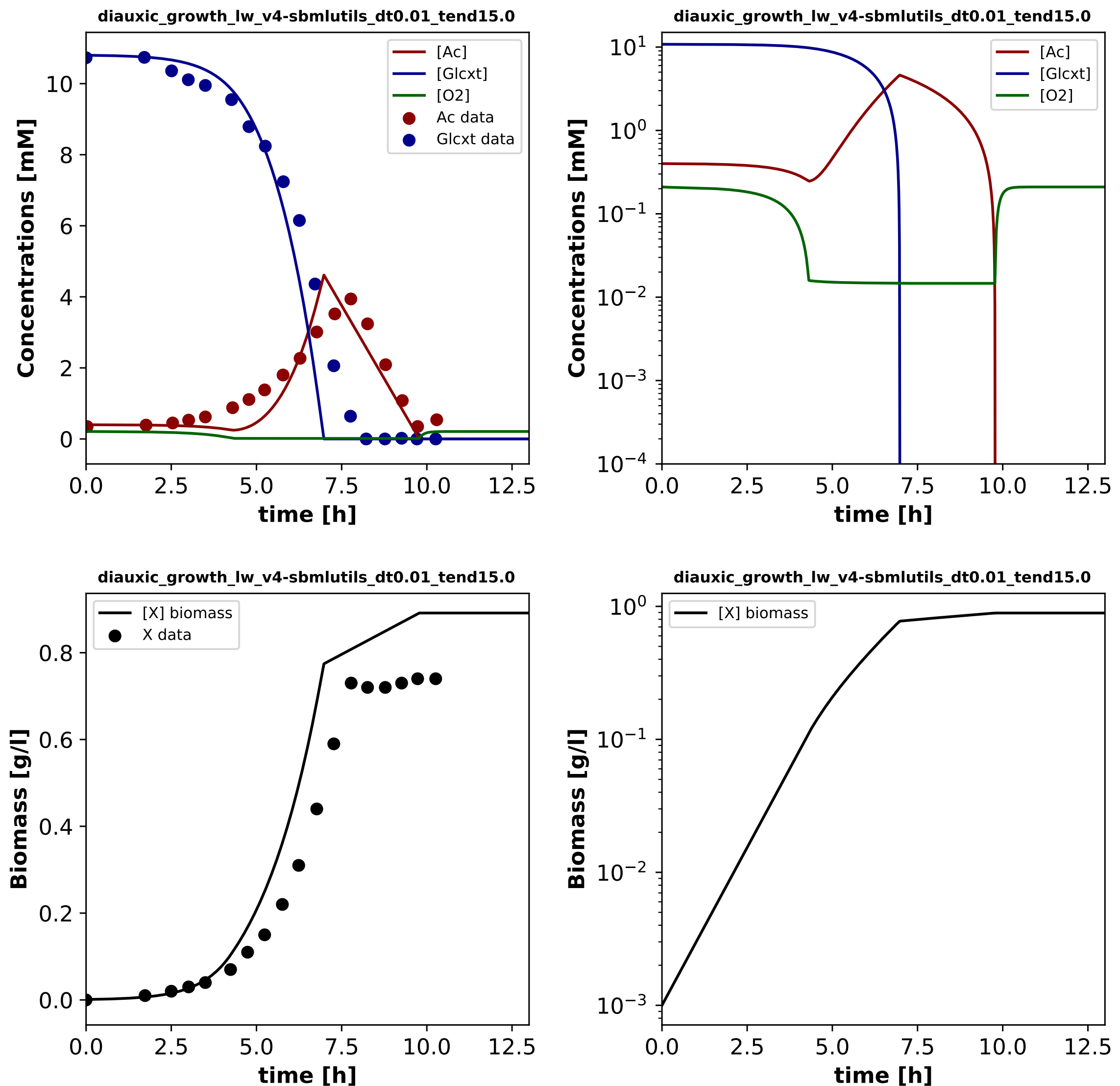
$$\begin{aligned}
\frac{dGlcxt}{dt} &= A^{Glcxt}\nu X \\
\frac{dA_c}{dt} &= A^{A_c}\nu X \\
\frac{dO_2}{dt} &= A^{O_2}\nu X + k_L a(0.21 - O_2) \\
\frac{dX}{dt} &= (v1 + v2 + v3 + v4)X
\end{aligned}$$

where A^{Glcxt}, A^{A_c}, A^{O_2} are the respective rows of each variable in the stoichiometry matrix and $k_L a$ is the mass transfer coefficient of oxygen. For a detailed description see [113].

The results in **Figure 5.5** on page 89 depict an exponential growth phase using glucose anaerobically until running out of glucose, which at this point the cell grows linearly due to oxygen. When both oxygen and glucose run out, the cell growth stagnates. Experimental data from [110] is plotted alongside the simulation results. The model is able to capture the behavior observed in the experimental data. The results are equivalent to the models in [113]. Results show that the proposed scheme is able to encode published DFBA models, resulting in a reproducible and exchangeable model representation between tools.

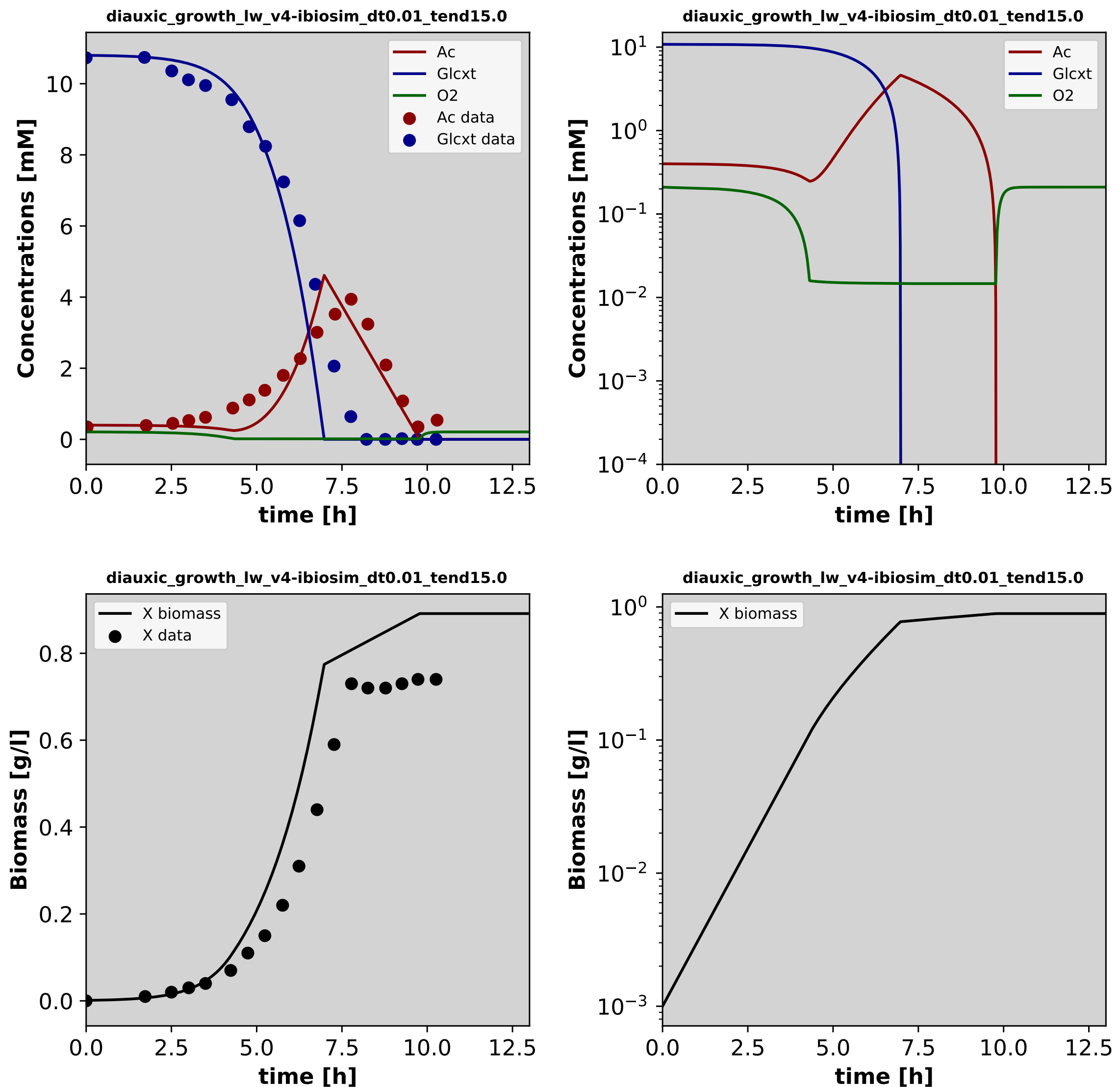
5.5.4 *E. coli* core (`ecoli`)

To demonstrate the feasibility of the proposed scheme for real-world examples of DFBAs, a larger metabolic network for the core metabolism of *E. coli* [130] was encoded in



(a)

Figure 5.5: This plot shows the results for the model representing diauxic growth in *E. coli*. The model is able to reproduce the general behavior captured from experiment data. There is an exponential cell growth while glucose is present in the model, but when the cell runs out of glucose, growth slows down and is affected mostly by oxygen. However, when the cell runs out of glucose and oxygen, growth diminishes significantly. The model was simulated for 15[h] with a time step `dt` of 0.01[h]. (a) Simulation results for the diauxic growth of *E. Coli* in `sbmlutils`. (b) Simulation results for the diauxic growth of *E. Coli* simulated in `iBioSim`.



(b)

Figure 5.5: (continued) This plot shows the results for the model representing diauxic growth in *E. coli*. The model is able to reproduce the general behavior captured from experiment data. There is an exponential cell growth while glucose is present in the model, but when the cell runs out of glucose, growth slows down and is affected mostly by oxygen. However, when the cell runs out of glucose and oxygen, growth diminishes significantly. The model was simulated for 15[h] with a time step `dt` of 0.01[h]. (a) Simulation results for the diauxic growth of *E. Coli* in `sbmlutils`. (b) Simulation results for the diauxic growth of *E. Coli* simulated in `iBioSim`.

the proposed scheme and simulated as shown in **Figure 5.6** on the following page. The FBA sub-model was downloaded from BiGG [131] (core metabolism of Escherichia coli str. K-12 substr. MG1655) and transformed to an DFBA model in an automatic fashion using `sbmlutils`. BiGG models encode the exchangeable species via annotated exchange reactions which allows and automatic inference of the dynamic species. The only additional information required to run DFBA simulations are initial concentrations for the species. The automatic encoding of larger scale examples demonstrates the scalability of the proposed encoding approach.

While `sbmlutils` is able to find a solution for the model, `iBioSim` cannot as it runs into an unfeasible solution in the middle of simulation. This captures the well-known problem of DFBA with multiple solutions. The FBA problem is not constrained enough to result in a unique solution and depending on which solution the simulator picks, different solutions and thereby trajectories arise. Despite the existence of multiple solutions, tools and LP solvers typically pick solutions deterministically. Hence, single tools can reproduce their own results, but results can be irreproducible between different implementations. Without the use of standards, this could never be demonstrated because variations in results could be due to discrepancies in the model, and not in the tool.

5.6 Summary

Modularity of models, the ability to encode multi-framework models, and reproducibility of models is indispensable for encoding more complex models in computational biology. In this work, an approach that allows a clear separation of the different modeling frameworks via comp sub-models and defining the interfaces between the sub-models is presented. The proposed scheme implements for the first time an exchangeable and reproducible multi-framework scheme purely in SBML. This scheme for encoding DFBA models in a standard way has been implemented in two different tools, demonstrating the exchangeability and reproducibility of our approach on various example models like diauxic growth in *E.coli*. `iBioSim` and `sbmlUtils` are freely available for download and offer the necessary infrastructure for anyone to develop DFBA models using the proposed scheme.

All of the materials presented in this chapter are publicly available at the following link:

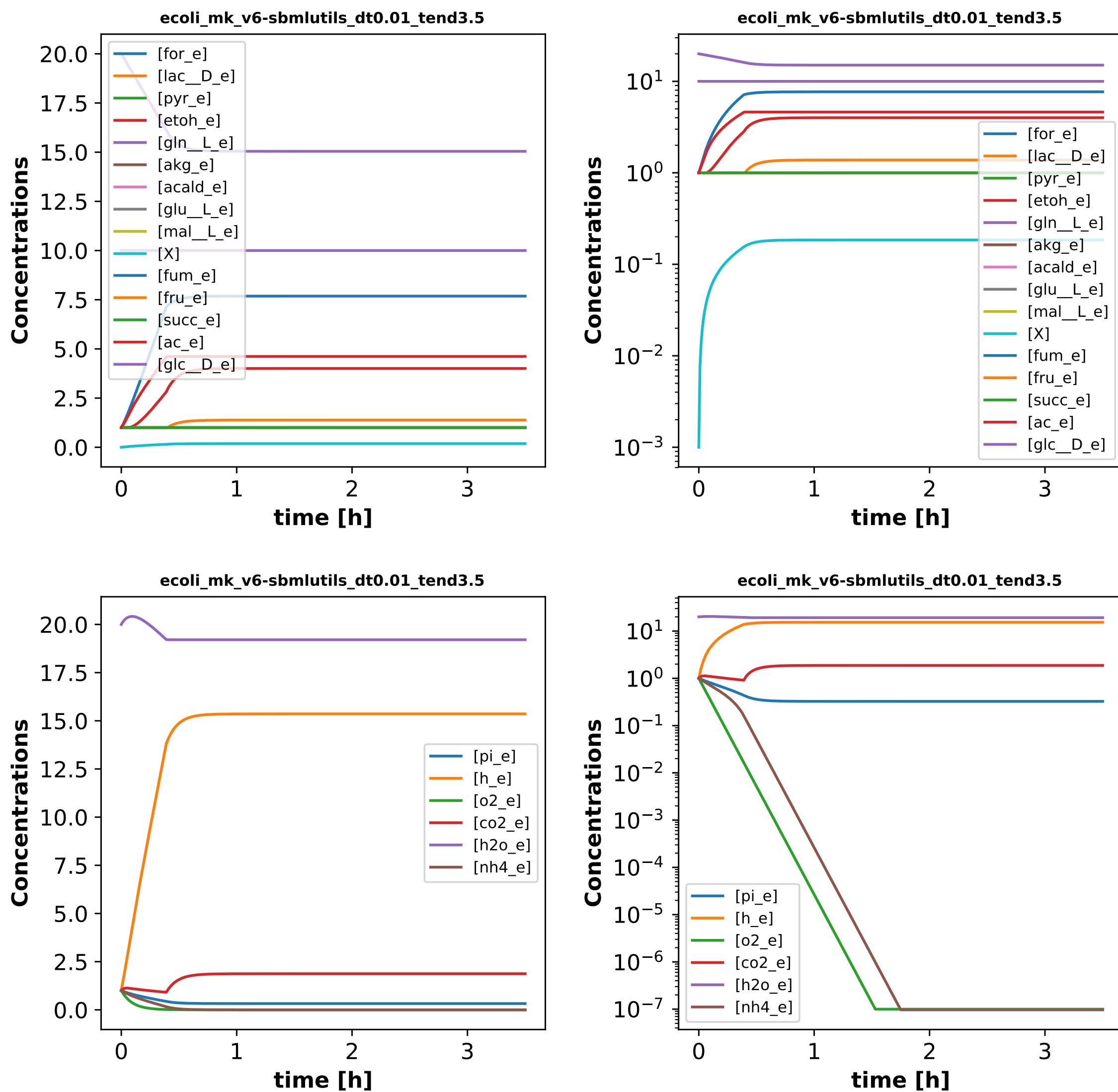


Figure 5.6: DFBA simulation results for core metabolism of *E. coli* with sbmlutils. The proposed approach can be used in larger models, such as the *E. coli* model described in the paper. The model is growing aerobically on glucose in the initial phase and reaches a steady state after oxygen is consumed. The model was simulated for 3.5[h] with a time step `dt` of 0.01[h].

https://github.com/matthiaskoenig/dfba/. This link includes the developed scheme consisting of rules, guidelines and additional information, as well as models and simulation results. Proposals, errata, and updates to the scheme are managed via the respective issue tracker and releases.

CHAPTER 6

TOWARDS REPRODUCIBLE DISEASE MODELS USING THE SYSTEMS BIOLOGY MARKUP LANGUAGE

SBML provides a standard data representation for biological models that can help towards reproducibility. Although SBML primarily targets biological models, the standard is capable of expanding to other fields. Many fields that lack a well-established infrastructure for reproducibility can potentially benefit from SBML. One example is the disease modeling field, which is a field that has encountered serious issues related to reproducibility. This chapter demonstrates how SBML and the arrays package can be applied to disease modeling. Section 6.1 discusses why using standards for disease models is important, problems related to reproducibility encountered in the field, and how SBML can be used to represent microsimulation disease models. Section 6.2 demonstrates through several abstract examples how microsimulation disease models can be encoded using the SBML Arrays package enabling reproducible disease modeling. One of the main benefits of standards in modeling is the exchange and reuse of models. Section 6.3 shows how the examples can be reproduced in two different tools, and finally Section 6.4 concludes the chapter by highlighting the importance of this work.

6.1 Overview

Disease modelers have been modeling progression of diseases for several decades using tools such as Markov Models or microsimulation. However, they need to address a serious phenomenon; many models they create are not reproducible. Moreover, there is no proper practice that ensures reproducible models since modelers rely on loose guidelines that change periodically, rather than well-defined machine-readable standards. SBML is one such standard that allows exchange of models amongst different software tools. Recently, the SBML Arrays package has been developed, which extends the standard

towards the modeling and simulation of population models.

Disease models attempt to explain phenomena observed by clinical trials and follow-up of patient populations through time. Such phenomena include complications of chronic diseases such as diabetes [132] and cardiovascular diseases [133], infectious diseases such as Ebola [134] and HIV [135], or even mental health conditions [136]. Beyond complications, models can also include economic aspects, such as costs or quality of life related to health utility scores. Models describe those phenomena using mathematical and statistical equations or other programmatic constructs.

In the past, differential equations have been used, which are still very dominant in the infectious disease domain [134]. However, other disease models have used state transition mechanisms. Markov cohort models have been prevalent in the past [136], but modern disease models tend to use *microsimulation* [132], where simulation considers each individual in the population separately. Some infectious disease models are also moving in the direction of individual-based simulation [137].

Individualization of computation makes models more flexible, but also more complex to understand. Therefore, clarity in model publication and transparency are essential. However, modeling practices in the field lack support for reproducibility. Publication of models' source code is rare – [138–141]. The norm is still publication of descriptive-only models in papers in which they appear, and only rarely do authors attempt to publish in a way that their work can be reproduced. However, publishing models within a paper does not allow full reproducibility as numeric examples provided in papers have insufficient precision and are prone to misinterpretation (see, for example [132]).

The *Mount Hood Diabetes Challenge* highlighted this reproducibility problem. The challenge revolved around reproducing models from two published papers. Multiple modeling teams around the world attempted to reproduce these published models, and they were unsuccessful. This is conclusive proof that a new method for model reproducibility is needed since models that cannot be reproduced are perceived to be non-credible.

To date, disease modelers have been trying to improve their model publication methods by publishing guidelines. Yet a better solution is to provide standard-compliant modeling tools that allow model exchange. This is exactly what SBML and associated languages such as *Pharmacometrics Markup Language* (PharmML) [51] and its counterpart

Model Description Language (MDL) [142] are designed for. While SBML is a standard representation that primarily targets chemical reaction networks, it has strong discrete event support in its core constructs, which allows the representation of a wide range of models other than chemical reaction networks in the form of Boolean networks [143], Petri nets [144], and Markov chains [87], among others. This is a continuation of the first attempt to reproduce disease models in such modeling languages that started with [145]. This work demonstrated how a disease model can be reproduced in three languages: SBML, PharmML, and MIcro Simulation Tool (MIST) [146]. When the SBML models in [145] were created, SBML capabilities lacked the means to support microsimulation. SBML has evolved with the recent introduction of the proposed SBML Arrays package. This package makes the expression of more complex models using microsimulation possible [147]. This chapter demonstrates this through a few abstract disease modeling examples that can now be implemented using SBML Arrays described in Chapter 4.

6.2 Disease Modeling Examples

To illustrate the requirements for disease models, the following sections present several abstract examples that are successfully implemented using SBML coupled with the arrays package. The first three examples are the same examples given in [145] but they have been modified with the addition of microsimulation components that were not originally modeled in the discrete-time Markov models. Two more examples are added that are impossible to model without SBML Arrays. Important nuances are discussed for each example.

6.2.1 Example 1: Simple Example

The first simple example is depicted in **Figure 6.1** on the next page. This example starts with a population of 100 individuals. Every individual starts in the *Alive* state. Each individual has 0.05 probability of moving to the Dead state. This example can be modeled as a cohort model as demonstrated before in [145], where the number of individuals in each state is counted for each time step. However, the model has been modified and implements it using microsimulation where each individual is processed through the model using Monte Carlo simulation with the probability defined. SBML Arrays defines an array of

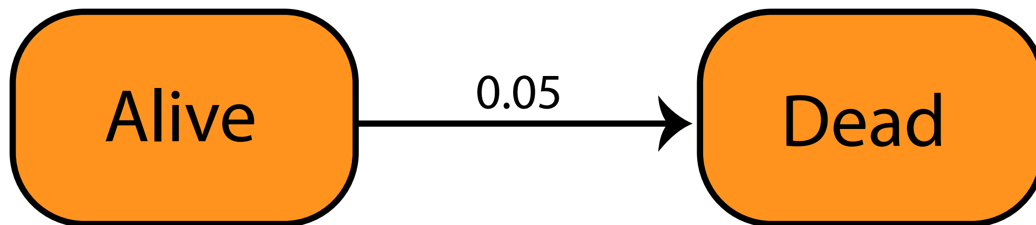


Figure 6.1: State transition diagram of a simple Markov model. The model uses two disease states: *Alive* and *Dead*, where the *Dead* state terminates simulation.
Initial conditions: 100 people start in Alive and none in Dead.
Output: Number of people in each state for years 1-10.

individuals, where each individual can be either *Alive* or *Dead*. Unlike cohort models where simulation continues for each time step until the end, microsimulation models can stop for individuals who reach a terminal state. In all simulations in this chapter, the terminal state would be represented by the *Dead* state. This mechanism is used in disease models to shorten simulation time and to indicate non-existence of a record for a human in years after death, effectively diminishing cohort size.

This entire model is implemented as SBML events as can be seen in **Table 6.1** on the following page. The *InstructionNumber* parameter is used to control the firing sequence of events and specific events competing in time. This competition of events is an important SBML element and is not related to the model being implemented. Therefore, the addition of the *InstructionNumber* parameter forces discrete times for the sequence of occurrences. Also note that model time and SBML implementation time are different. In this example, there is a header of events enumerated as #0 ,#1,#2 that start each time step in the simulation. Event #0 advances the *Time* parameter. Event #1 provides a point where the user can take a snapshot of the data to represent the state of the system in the time step. Event #3 is used for termination. The last three events represent transitions. Namely, Event #4 generates a random number and stores it in the *Random* variable. Event #4 tests if the drawn random number matches the transition criteria and, if so, updates the states and increases the instruction count to progress the simulation. Event #5 is a counter event for event #4 that is triggered if event #4 does not happen. It is essential to advance the simulation by setting the *InstructionNumber*. Unless set, the simulation would not continue, since there would not be another event for the individual, which is how event #3 terminates simulation. Alternatively, termination can happen if the simulation

Table 6.1: SBML events for Example 1.

	Trigger	Assignments
0	InstructionNumber[d0] $= 0$	Time[d0] $:=$ Time[d0] $+ 1$ InstructionNumber[d0] $:= 0.1$
1	InstructionNumber[d0] $= 0.1$	InstructionNumber[d0] $:= 0.2$
2	InstructionNumber[d0] $= 0.2 \wedge$ Dead[d0] $= 0 \wedge$ Time[d0] < 10	InstructionNumber[d0] $:= 1$
3	InstructionNumber[d0] $= 1$	Random[d0] $=$ uniform$(0, 1)$ InstructionNumber[d0] $:= 1.5$
4	InstructionNumber[d0] = 1.5 $\wedge$ Alive[d0] $= 1 \wedge$ Random[d0] $\geq 0 \wedge$ Random[d0] $< (0 + 0.05)$	Alive[d0] $:= 0$ Dead[d0] $:= 1$ InstructionNumber[d0] $:= 0$
5	InstructionNumber[d0] $= 1.5 \wedge$ Alive[d0] $= 1 \wedge$ Random[d0] $\geq (0 + 0.05) \wedge$ Random[d0] < 1	InstructionNumber[d0] $:= 0$

time limit is reached. Future examples follow this structure. The models shown here are generated using the SBML Arrays package that automatically generates any arbitrary number of individual copies, which is 100 for the examples shown.

6.2.2 Example 2: Three State Markov Model

The next example (depicted in **Figure 6.2** on the current page) is a simple extension of the first one. This example demonstrates how new states and transitions are added by introducing more parameters and events. Rather than starting with a population of individuals in the *Alive* state, this example starts with individuals in the *Healthy* state. Healthy individuals can transition to the *Sick* state each year with 2% probability and they can transition to the *Dead* state with a 1% probability. Individuals in the *Sick* state can transition back to a *Healthy* state with 10% probability and to the *Dead* state with 5% probability.

Table 6.2 on the following page represents the events implemented to run this simulation. Events #0-#2 form a simulation header. Events #3-#6 represent transitions originating from the *Healthy* state while #7-#10 represent transitions originating from the *Sick* state. Each state has three events since there are two transitions emanating from each event and therefore three competing options have to be checked: taking the first transition, taking the second transition, or not taking any transition. The model generates a random variable and stores it in the *Random* parameter. The three events afterward check the three different possible options using the *Random* parameter and transition thresholds. This structure is used in all the remaining examples.

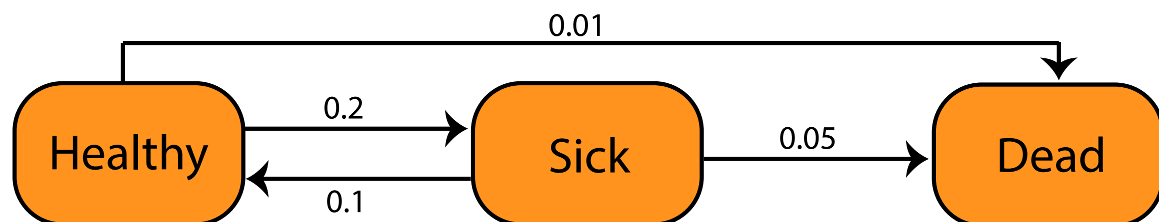


Figure 6.2: State transition diagram of a three state Markov model. There are 3 disease states: *Healthy*, *Sick*, and *Dead*, where the *Dead* state is terminal
Initial conditions: *Healthy* = 100, *Sick* = 0 and *Dead* = 0.
Output: Number of people in each state for years 1-10.

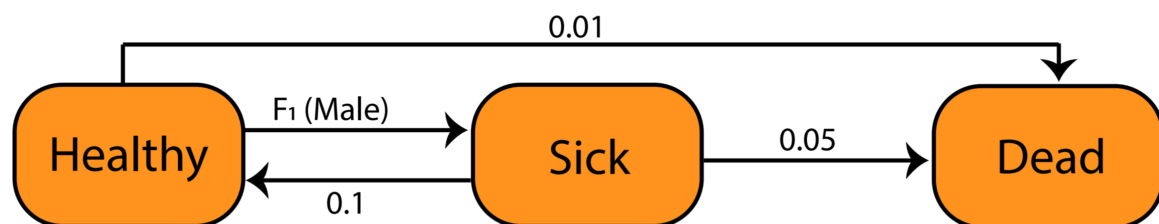
Table 6.2: SBML events for Example 2.

	Trigger	Assignments
0	InstructionNumber[d0] $= 0$	Time[d0] $:=$ Time[d0] $+ 1$ InstructionNumber[d0] $:= 0.1$
1	InstructionNumber[d0] $= 0.1$	InstructionNumber[d0] $:= 0.2$
2	InstructionNumber[d0] $= 0.2 \wedge$ Dead[d0] $:= 0 \wedge$ Time[d0] < 10	InstructionNumber[d0] $:= 1$
3	InstructionNumber[d0] $= 1$	Random[d0] $:=$ uniform$(0, 1)$ InstructionNumber[d0] $:= 1.5$
4	InstructionNumber[d0] = 1.5 $\wedge$ Healthy[d0] = 1 $= 1 \wedge$ Random[d0] $\geq 0 \wedge$ Random[d0] $< (0 + 0.01)$	Healthy[d0] $:= 0$ Dead[d0] $:= 1$ InstructionNumber[d0] $:= 0$
5	InstructionNumber[d0] = 1.5 $\wedge$ Healthy[d0] = 1 $= 1 \wedge$ Random[d0] $\geq (0 + 0.01) \wedge$ Random[d0] $< (0 + 0.01) + 0.2$	Healthy[d0] $:= 0$ Sick[d0] $:= 1$ InstructionNumber[d0] $:= 0$
6	InstructionNumber[d0] $= 1.5 \wedge$ Healthy[d0] $= 1 \wedge$ Random[d0] $\geq (0 + 0.01) + 0.2 \wedge$ Random[d0] < 1	InstructionNumber[d0] $:= 0$
7	InstructionNumber[d0] $= 1$	Random[d0] $:=$ uniform$(0, 1)$ InstructionNumber[d0] $:= 1.5$
8	InstructionNumber[d0] = 1.5 $\wedge$ Sick[d0] = 1 $= 1 \wedge$ Random[d0] $\geq 0 \wedge$ Random[d0] $< (0 + 0.1)$	Sick[d0] $:= 0$ Healthy[d0] $:= 1$ InstructionNumber[d0] $:= 0$
9	InstructionNumber[d0] = 1.5 $\wedge$ Sick[d0] = 1 $= 1 \wedge$ Random[d0] $\geq (0 + 0.1) \wedge$ Random[d0] $< (0 + 0.1) + 0.3$	Sick[d0] $:= 0$ Dead[d0] $:= 1$ InstructionNumber[d0] $:= 0$
10	InstructionNumber[d0] $= 1.5 \wedge$ Sick[d0] $= 1 \wedge$ Random[d0] $\geq (0 + 0.1) + 0.3 \wedge$ Random[d0] < 1	InstructionNumber[d0] $:= 0$

6.2.3 Example 3: Stratified Markov Model

The example depicted in **Figure 6.3** on the current page is similar to the previous example. That is, this example starts with individuals in the *Healthy* state. Healthy individuals can transition to the *Dead* state with 1% probability. Individuals in the *Sick* state can transition back to a *Healthy* state with 0.1 probability and to the *Dead* state with 5% probability. However, unlike the first two examples, this example starts introducing microsimulation concepts. Every individual is given a gender. This parameter is given by the *Male* parameter, which is a Boolean variable where the value is 1 if the individual is male and 0 if the individual is female. This parameter is used in the $F_1(\text{Male})$ function that governs the transition probability. In this case, healthy males become sick with higher probability than females. Therefore, simulation should show a higher sickness and death rate amongst males.

This example is still simple enough to implement as two separate cohort models as can be seen in **Table 6.3** on the following page. The transition probabilities are controlled by the *Male* parameter, which is used in event #5 and in the counter event #6. Microsimulation becomes more significant and challenging when individuals have more characteristics. This is explored further in the next examples.



$$F_1(\text{Male}) = 0.1 \cdot (1 + \text{Male})$$

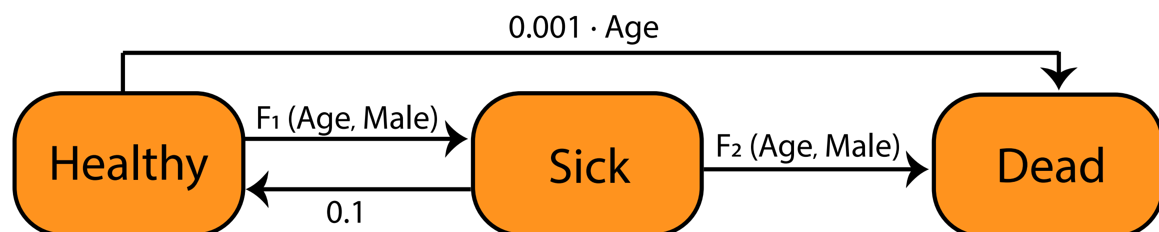
Figure 6.3: State transition diagram of a simple Markov model. There are three disease states: *Healthy*, *Sick*, and *Dead*, where the Dead state is terminal. The transition probability now depends on the cohort (male or female) and can be expressed as a function of a Boolean covariate *Male*.
Initial conditions: *Healthy* = (50 males, 50 females), *Sick* = (0,0) and *Dead* = (0,0).
Output: Number of men and women in each disease state for years 1-10.

Table 6.3: SBML events for Example 3.

	Trigger	Assignments
0	$\text{InstructionNumber}[d0] = 0$	$\text{Time}[d0] := \text{Time}[d0] + 1$ $\text{InstructionNumber}[d0] := 0.1$
1	$\text{InstructionNumber}[d0] = 0.1$	$\text{InstructionNumber}[d0] := 0.2$
2	$\text{InstructionNumber}[d0] = 0.2 \wedge \text{Dead}[d0] = 0 \wedge \text{Time}[d0] < 10$	$\text{InstructionNumber}[d0] := 1$
3	$\text{InstructionNumber}[d0] = 1$	$\text{Random}[d0] := \text{uniform}(0, 1)$ $\text{InstructionNumber}[d0] := 1.5$
4	InstructionNumber[d0] = 1.5 $\wedge$ Healthy[d0] = 1 $= 1 \wedge \text{Random}[d0] \geq 0 \wedge \text{Random}[d0] < (0 + 0.01)$	$\text{Healthy}[d0] := 0$ $\text{Dead}[d0] := 1$ $\text{InstructionNumber}[d0] := 0$
5	InstructionNumber[d0] = 1.5 $\wedge$ Healthy[d0] = 1 $= 1 \wedge \text{Random}[d0] \geq (0 + 0.01) \wedge \text{Random}[d0] < (0 + 0.01) + 0.1 * (1 + \text{Male}[d0])$	$\text{Healthy}[d0] := 0$ $\text{Sick}[d0] := 1$ $\text{InstructionNumber}[d0] := 0$
6	$\text{InstructionNumber}[d0] = 1.5 \wedge \text{Healthy}[d0] = 1 \wedge \text{Random}[d0] \geq (0 + 0.01) + 0.1 * (1 + \text{Male}[d0]) \wedge \text{Random}[d0] < 1$	$\text{InstructionNumber}[d0] := 0$
7	$\text{InstructionNumber}[d0] = 1$	$\text{Random}[d0] := \text{uniform}(0, 1)$ $\text{InstructionNumber}[d0] := 1.5$
8	InstructionNumber[d0] = 1.5 $\wedge$ Sick[d0] = 1 $= 1 \wedge \text{Random}[d0] \geq 0 \wedge \text{Random}[d0] < (0 + 0.1)$	$\text{Sick}[d0] := 0$ $\text{Healthy}[d0] := 1$ $\text{InstructionNumber}[d0] := 0$
9	InstructionNumber[d0] = 1.5 $\wedge$ Sick[d0] = 1 $= 1 \wedge \text{Random}[d0] \geq (0 + 0.1) \wedge \text{Random}[d0] < (0 + 0.1) + 0.3$	$\text{Sick}[d0] := 0$ $\text{Dead}[d0] := 1$ $\text{InstructionNumber}[d0] := 0$
10	$\text{InstructionNumber}[d0] = 1.5 \wedge \text{Sick}[d0] = 1 \wedge \text{Random}[d0] \geq (0 + 0.1) + 0.3 \wedge \text{Random}[d0] < 1$	$\text{InstructionNumber}[d0] := 0$

6.2.4 Example 4: State Transition Model Dependent on Changing Parameters

This example is depicted in **Figure 6.4** on the current page. This model can no longer be implemented using Markov cohort models due to the yearly change in *Age* and the stratification by *Male* and *Age* of the transition probabilities. This example captures the heterogeneity of the population by describing each individual's behavior. SBML arrays allows for the definition of distinct individuals. **Table 6.4** on the following page presents the event sets for this example. SBML events plays a crucial role by increasing the *Age* every year before transition probabilities are calculated. The new element in this model is the change of *Age* before determining transitions in each simulation timestep. This can be seen in Instructions #3 to #5 that behave similar to transitions — note that some of the code is redundant and can be replaced by one event since event #5 never fires. However, this example maintains this code structure for compatibility and future extendibility. Once again, our method uses *InstructionNumber* to guide the model during simulation such that state transitions are considered at each simulation time step only after *InstructionNumber* reaches the value of 2. Despite the complexity of this example, it is not yet representative of the full range of phenomena we wish to model that include treatment and cost. The next example shows how this is accomplished.



$$F_1(Age, Male) = \min(0.8, 0.1 \cdot (1 + Male) + 0.01 \cdot Age)$$
$$F_2(Age, Male) = \min(0.9, 0.01 \cdot Age + 0.2 \cdot Male)$$

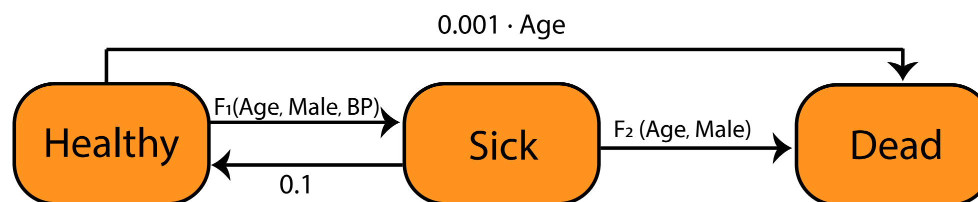
Figure 6.4: State transition model dependent on changing parameters. There are 3 disease states: Healthy, Sick, and Dead, where the Dead state is terminal.
Pre-Transition Rules: Age increased by 1 each cycle.
Initial conditions: Healthy = (50 Male, 50 Female with Age =1,2,. . . ,50 for each individual), Sick = (0,0) and Dead = (0,0).
Output: Number of men and women in each disease state for years 1-10 and their ages in each state.

Table 6.4: SBML events for Example 4.

	Trigger	Assignments
0	$\text{InstructionNumber[d0]} = 0$	$\text{Time[d0]} := \text{Time[d0]} + 1$ $\text{InstructionNumber[d0]} := 0.1$
1	$\text{InstructionNumber[d0]} = 0.1$	$\text{InstructionNumber[d0]} := 0.2$
2	$\text{InstructionNumber[d0]} = 0.2 \wedge \text{Dead[d0]} = 0 \wedge \text{Time[d0]} < 10$	$\text{InstructionNumber[d0]} := 1$
3	$\text{InstructionNumber[d0]} = 1$	$\text{Random[d0]} := \text{uniform}(0, 1)$ $\text{InstructionNumber[d0]} := 1.5$
4	$\text{InstructionNumber[d0]} = 1.5 \wedge 1$	$\text{Age[d0]} := \text{Age[d0]} + 1$ $\text{InstructionNumber[d0]} := 2$
5	$\text{InstructionNumber[d0]} = 1.5 \wedge 0$	$\text{InstructionNumber[d0]} := 2$
6	$\text{InstructionNumber[d0]} = 2$	$\text{Random[d0]} := \text{uniform}(0, 1)$ $\text{InstructionNumber[d0]} := 2.5$
7	$\text{InstructionNumber[d0]} = 2.5 \wedge \text{Healthy[d0]} = 1 \wedge \text{Random[d0]} \geq 0 \wedge \text{Random[d0]} < \left(0 + \frac{\text{Age[d0]}}{1000}\right)$	$\text{Random[d0]} := \text{uniform}(0, 1)$ $\text{InstructionNumber[d0]} := 1.5$
8	$\text{InstructionNumber[d0]} = 2.5 \wedge \text{Healthy[d0]} = 1 \wedge \text{Random[d0]} \geq \left(0 + \frac{\text{Age[d0]}}{1000}\right) \wedge \text{Random[d0]} < \left(0 + \frac{\text{Age[d0]}}{1000}\right) + \min(0.8, 0.1 * (1 + \text{Male[d0]}) + 0.01 * \text{Age[d0]})$	$\text{Sick[d0]} := 1$ $\text{Healthy[d0]} := 0$ $\text{InstructionNumber[d0]} := 0$
9	$\text{InstructionNumber[d0]} = 2.5 \wedge \text{Healthy[d0]} = 1 \wedge \text{Random[d0]} \geq \left(0 + \frac{\text{Age[d0]}}{1000}\right) + \min(0.8, 0.1 * (1 + \text{Male[d0]}) + 0.01 * \text{Age[d0]}) \wedge \text{Random[d0]} < 1$	$\text{InstructionNumber[d0]} := 0$
10	$\text{InstructionNumber[d0]} = 2$	$\text{Random[d0]} := \text{uniform}(0, 1)$ $\text{InstructionNumber[d0]} := 2.5$
11	$\text{InstructionNumber[d0]} = 2.5 \wedge \text{Sick[d0]} = 1 \wedge \text{Random[d0]} \geq 0 \wedge \text{Random[d0]} < (0 + 0.1)$	$\text{Sick[d0]} := 0$ $\text{Healthy[d0]} = 1$ $\text{InstructionNumber[d0]} := 0$
12	$\text{InstructionNumber[d0]} = 2.5 \wedge \text{Sick[d0]} = 1 \wedge \text{Random[d0]} \geq (0 + 0.1) \wedge \text{Random[d0]} < (0 + 0.1) + \min(0.9, 0.2 * \text{Male[d0]} + 0.01 * \text{Age[d0]})$	$\text{Sick[d0]} := 1$ $\text{Healthy[d0]} := 0$ $\text{InstructionNumber[d0]} := 0$
13	$\text{InstructionNumber[d0]} = 2.5 \wedge \text{Sick[d0]} = 1 \wedge \text{Random[d0]} \geq (0 + 0.1) + \min(0.9, 0.2 * \text{Male[d0]} + 0.01 * \text{Age[d0]}) \wedge \text{Random[d0]} < 1$	$\text{InstructionNumber[d0]} := 0$

6.2.5 Example 5: State Transition Model with Treatment and Costs

This example is depicted in **Figure 6.5** on the current page. This model adds *Blood Pressure* (*BP*) as another parameter that increases yearly at different rates. Once *BP* is above a threshold, treatment is administered that drops it back closer to previous values. Moreover, costs include elements of *Age* and *Treatment*. Even this relatively simple example is complex enough to show why individual modeling is needed, and hence making SBML Arrays essential. **Table 6.5** on the following page shows the event scheme implementation in SBML. Notice that in this example there are multiple post transition rules implemented as event triplets: #3-#5 handle *Age* increment in pre-transition, #6-#8 handle *BP* pre-transition update, #17-#19 determine whether treatment is administered post-transition, #20-#22 adjust *BP* according to treatment for next timestep post-treatment calculation, #23-#25 calculate yearly cost that includes treatment cost, and finally #26-#28 accumulate total cost. The important elements of this simulation are the pre-transition rules and post-transition rules. Each of those rule sets needs to be executed in sequential order during simulation. SBML events allow for timing these using the *InstructionCounter*.



$$F_1(Age, Male, BP) = \min(0.8, 0.1 \cdot (1 + Male) + 0.01 \cdot Age + (0.01 \cdot (BP-120))^2)$$
$$F_2(Age, Male) = \min(0.9, 0.01 \cdot Age + 0.2 \cdot Male)$$

Figure 6.5: State transition diagram with functions depending on Age, Male, BP (Blood Pressure). There are 3 disease states: Healthy, Sick, and Dead, where the Dead state is terminal.
Pre-Transition Rules: Age increased by 1 and BP by Age/10 each simulation cycle.
Post-Transition Rules: $Treatment = BP > 140$ (i.e. becomes 1 when BP crosses 140 threshold); $BP = BP - Treatment * 10$ (i.e. a drop of 10 once treatment is applied); $CostThisYear = Age + Treatment * 10$ (i.e. cost depends on age and if treatment was taken); $Cost = Cost + CostThisYear$ (i.e. accumulates cost over time).
Initial conditions: Healthy = (50 Male, 50 Female with Age =1,2,. . . ,50 for each individual), BP =120, Sick = (0,0) and Dead = (0,0).
Output: Number of men and women in each disease state for years 1-10 and their ages and costs in each state. A stratified report by male and female and young – up to age 30 and old above age 30 is produced.

Table 6.5: SBML events for Example 5.

	Trigger	Assignments
0	$\text{InstructionNumber[d0]} = 0$	$\text{Time[d0]} := \text{Time[d0]} + 1$ $\text{InstructionNumber[d0]} := 0.1$
1	$\text{InstructionNumber[d0]} = 0.1$	$\text{InstructionNumber[d0]} := 0.2$
2	$\text{InstructionNumber[d0]} = 0.2 \land \text{Dead[d0]} = 0 \land \text{Time[d0]} < 10$	$\text{InstructionNumber[d0]} := 1$
3	$\text{InstructionNumber[d0]} = 1$	$\text{Random[d0]} := \text{uniform}(0, 1)$ $\text{InstructionNumber[d0]} := 1.5$
4	InstructionNumber[d0] = 1.5 $\land$ 1	$\text{Age[d0]} := \text{Age[d0]} + 1$ $\text{InstructionNumber[d0]} := 2$
5	InstructionNumber[d0] = 1.5 $\land$ 0	$\text{InstructionNumber[d0]} := 2$
6	$\text{InstructionNumber[d0]} = 2$	$\text{Random[d0]} := \text{uniform}(0, 1)$ $\text{InstructionNumber[d0]} := 2.5$
7	$\text{InstructionNumber[d0]} = 2.5 \land 1$	$\text{InstructionNumber[d0]} := 3$ $\text{BP[d0]} := \text{BP[d0]} + \frac{\text{Age[d0]}}{10}$
8	InstructionNumber[d0] = 2.5 $\land$ 0	$\text{InstructionNumber[d0]} := 3$
9	$\text{InstructionNumber[d0]} = 3$	$\text{Random[d0]} := \text{uniform}(0, 1)$ $\text{InstructionNumber[d0]} := 3.5$
10	$\text{InstructionNumber[d0]} = 3.5 \land \text{Healthy[d0]} = 1 \land \text{Random[d0]} \geq 0 \land \text{Random[d0]} < \left(0 + \frac{\text{Age[d0]}}{1000}\right)$	$\text{Healthy[d0]} := 0$ $\text{Dead[d0]} := 1$ $\text{InstructionNumber[d0]} := 4$
11	$\text{InstructionNumber[d0]} = 3.5 \land \text{Healthy[d0]} = 1 \land \text{Random[d0]} \geq \left(0 + \frac{\text{Age[d0]}}{1000}\right) \land \text{Random[d0]} < \left(0 + \frac{\text{Age[d0]}}{1000}\right) + \min(0.8, 0.1 * (1 + \text{Male[d0]}) + 0.01 * \text{Age[d0]} + \left(\frac{\text{BP[d0]} - 120}{100}\right)^2)$	$\text{Sick[d0]} := 1$ $\text{Healthy[d0]} := 0$ $\text{InstructionNumber[d0]} := 4$
12	$\text{InstructionNumber[d0]} = 3.5 \land \text{Healthy[d0]} = 1 \land \text{Random[d0]} \geq \left(0 + \frac{\text{Age[d0]}}{1000}\right) + \min(0.8, 0.1 * (1 + \text{Male[d0]}) + 0.01 * \text{Age[d0]} + \left(\frac{\text{BP[d0]} - 120}{100}\right)^2) \land \text{Random[d0]} < 1$	$\text{InstructionNumber[d0]} := 4$

Table 6.5: (continued) SBML events for Example 5.

	Trigger	Assignments
13	InstructionNumber[d0] $= 3$	Random[d0] $:=$ uniform$(0,1)$ InstructionNumber[d0] $:= 3.5$
14	InstructionNumber[d0] $= 3.5 \wedge$ Sick[d0] $= 1 \wedge$ Random[d0] $\geq 0 \wedge$ Random[d0] $< (0+0.1)$	Sick[d0] $:= 0$ Healthy[d0] $:= 1$ InstructionNumber[d0] $:= 4$
15	InstructionNumber[d0] $= 3.5 \wedge$ Sick[d0] $= 1 \wedge$ Random[d0] $\geq (0+0.1) \wedge$ Random[d0] $< (0+0.1) + \min(0.9, 0.2 *$ Male[d0] $+ 0.01 *$ Age[d0]$)$	Sick[d0] $:= 0$ Dead[d0] $:= 1$ InstructionNumber[d0] $:= 4$
16	InstructionNumber[d0] $= 3.5 \wedge$ Sick[d0] $= 1 \wedge$ Random[d0] $< (0+0.1) + \min(0.9, 0.2 *$ Male[d0] $+ 0.01 *$ Age[d0] $\wedge$ Random[d0] < 1	InstructionNumber[d0] $:= 4$
17	InstructionNumber[d0] $= 4$	Random[d0] $:=$ uniform$(0,1)$ InstructionNumber[d0] $:= 4.5$
18	InstructionNumber[d0] = 4.5 $\wedge$ 1	Treatment[d0] $:=$ Age[d0] $+ 1$ InstructionNumber[d0] $:= 5$
19	InstructionNumber[d0] $= 4.5 \wedge 0$	InstructionNumber[d0] $:= 5$
20	InstructionNumber[d0] $= 5$	Random[d0] $:=$ uniform$(0,1)$ InstructionNumber[d0] $:= 5.5$
21	InstructionNumber[d0] $= 5.5 \wedge 1$	BP[d0] $:=$ BP[d0] $-$ Treatment[d0]*10 InstructionNumber[d0] $:= 6$
22	InstructionNumber[d0] $= 5.5 \wedge 0$	InstructionNumber[d0] $:= 6$
23	InstructionNumber[d0] $= 6$	InstructionNumber[d0] $:= 6$
24	InstructionNumber[d0] $= 6.5 \wedge 1$	CostThisYear[d0] $:=$ Age[d0] $+$ Treatment[d0] $* 10$ InstructionNumber[d0] $:= 7$
25	InstructionNumber[d0] $= 6.5 \wedge 0$	InstructionNumber[d0] $:= 7$
26	InstructionNumber[d0] $= 7$	InstructionNumber[d0] $:= 7.5$
27	InstructionNumber[d0] $= 7.5 \wedge 1$	Cost[d0] $:=$ Cost[d0] $+$ CostThisYear[d0] InstructionNumber[d0] $:= 0$
28	InstructionNumber[d0] $= 7.5 \wedge 0$	InstructionNumber[d0] $:= 0$

6.3 Results

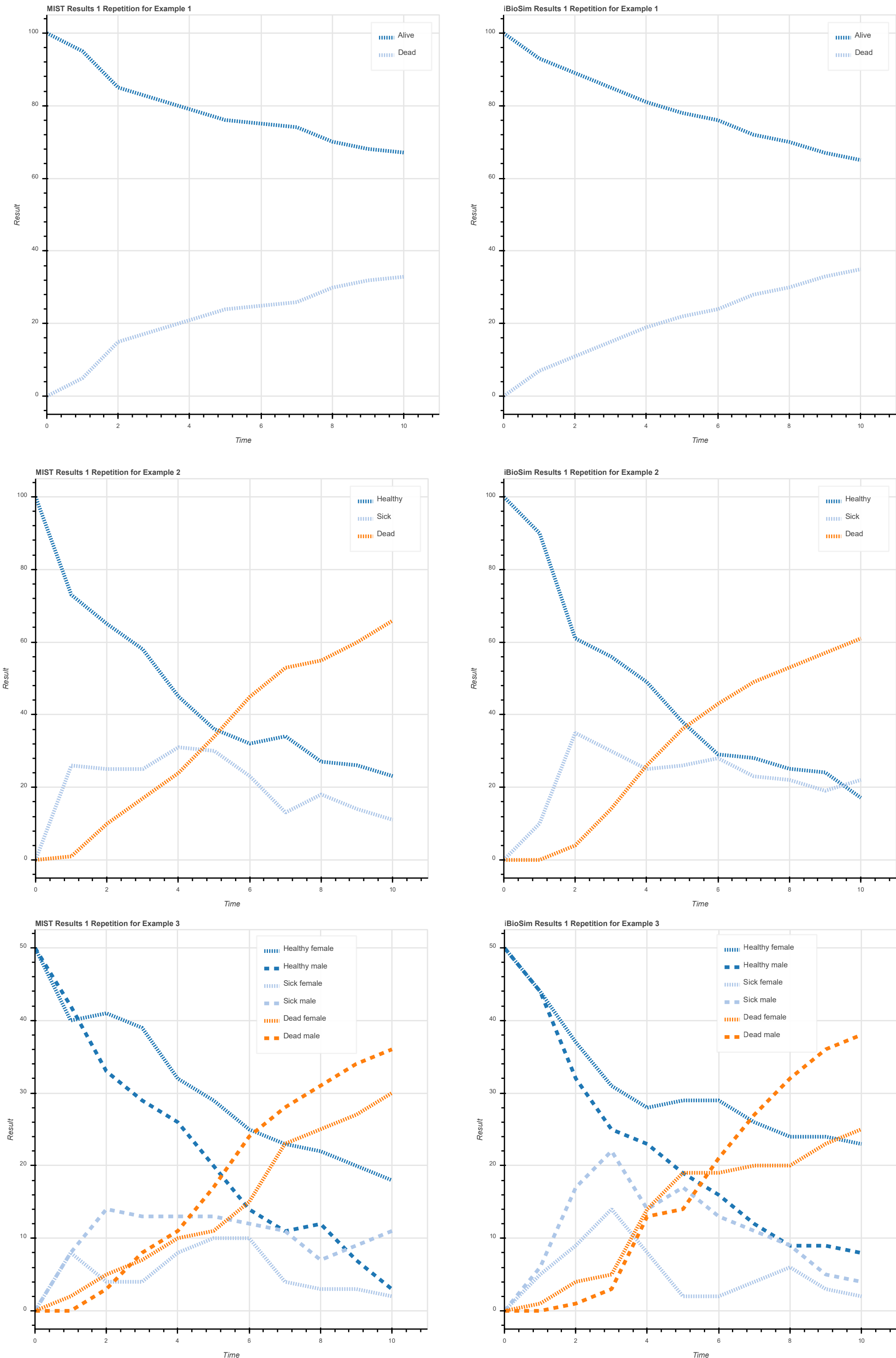
The examples described above are implemented in the Python language as SBML files and then simulated using iBioSim, which supports SBML Arrays. Since these examples are not intuitive, a reference is needed to provide some comparison of results. The MIcro Simulation Tool (MIST) is used to implement the same examples. Since MIST is particularly designed for disease modeling, comparable results provide sufficient support to the claim that SBML Arrays is suitable to create reproducible disease models. **Figure 6.6** on page 109, **Figure 6.7** on page 112, **Figure 6.8** on page 115 presents the result of simulation using MIST compared to SBML Arrays implemented in iBioSim. Since this is random simulation, results should not match exactly using a single run of the simulation, **Figure 6.6** on page 109 shows this case, yet they are comparable enough to indicate a similar simulation. To verify that the models indeed are identical, the models are executed 10 times and results are averaged as shown in **Figure 6.7** on page 112. The average results of 100 repetitions are shown in **Figure 6.8** on page 115. The plots show clear convergence as more repetitions are added. To provide additional support, we conducted statistical analysis of results for each example in a similar way. Statistical analysis has been performed on each example: Example 1 (**Figure 6.9** on page 118), Example 2 (**Figure 6.10** on page 119), Example 3 (**Figure 6.11** on page 120, **Figure 6.12** on page 121), Example 4 (**Figure 6.13** on page 122, **Figure 6.14** on page 124), and Example 5 (**Figure 6.15** on page 126, **Figure 6.16** on page 133). In these analyses, the vertical axis represents the absolute difference between MIST and iBioSim results. The left column shows the mean as circles and standard deviation as squares for each year in simulation. The right column shows the average difference of all years and the horizontal axis represents repetitions. The convergence of the model results is clearly seen from those statistics for most plots where both the mean and standard deviation difference is reduced by adding iterations. There are several outliers where mean does not follow this trend, such as in example 3 healthy male 10 repetitions mean, yet even in those cases standard deviation statistic improves or stays similar implying convergence. In Example 5 there are three cases where standard deviation does not improve for 100 repetitions: dead young male and age old female, and cost old female. However, in those examples the mean statistic improves and considering that example 5 is highly stratified, has some relatively rare events and has somewhat volatile

changes in age, so it is quite reasonable and expected. Therefore, we conclude that the examples are reproduced properly between tools as clearly seen in **Figure 6.8** on page 115.

6.4 Summary

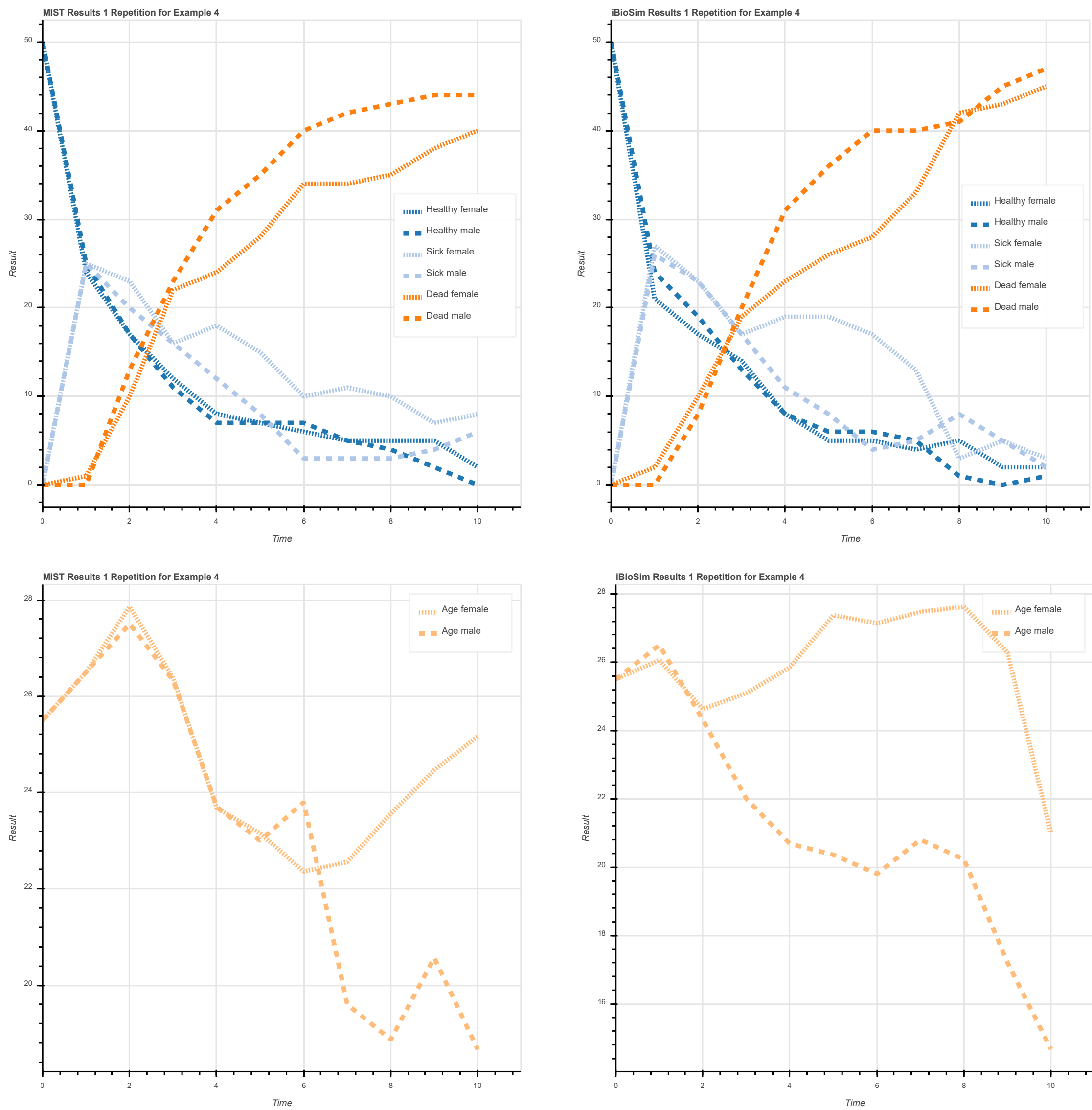
Disease modelers have been modeling progression of diseases for several decades using tools such as Markov Models or microsimulation. However, they need to address a serious challenge; many models they create are not reproducible. Moreover, there is no proper practice that ensures reproducible models, since modelers rely on loose guidelines that change periodically, rather than well-defined machine-readable standards. Recently, the SBML Arrays package has been developed, which extends the standard to allow handling simulation of populations. This chapter demonstrates how such microsimulation disease models can be encoded in SBML. As demonstrated through several abstract examples, the disease models created in `MIST` and converted to SBML can be successfully simulated in `iBioSim`. To support this reproducibility, the models, example code, and results for both implementations are available online at: https://github.com/Jacob-Barhak/DiseaseModelsSBML. This repository includes detailed instructions to replicate the results in this paper in both `MIST` and `iBioSim` as well as Python scripts to assist SBML creation and additional statistical analysis. Although the results were generated with `MIST` and iBioSim. It is important to remember that this paper promotes SBML with arrays as a transfer mechanism between systems rather than focusing on a specific system. In addition, all of the examples have been uploaded to the BioModels database [148]. The models used in this paper have been assigned the following identifiers: MODEL1803120002, MODEL1803120003, MODEL1803120004, MODEL1803120005, and MODEL1803120006 for Example 1, Example 2, Example 3, Example 4, and Example 5, respectively.

Note that reproducibility has many facets. Different implementations of the models with different tools may generate different results for such probabilistic models shown in this chapter. Therefore, asking for the exact same output files generated by two systems is not practical. However, it is expected that the same tools after receiving the SBML file will be able to internally to repeat the same results given the same random seed. It is also expected that repetition of the same model simulations on different systems have to converge towards the same statistical solution.



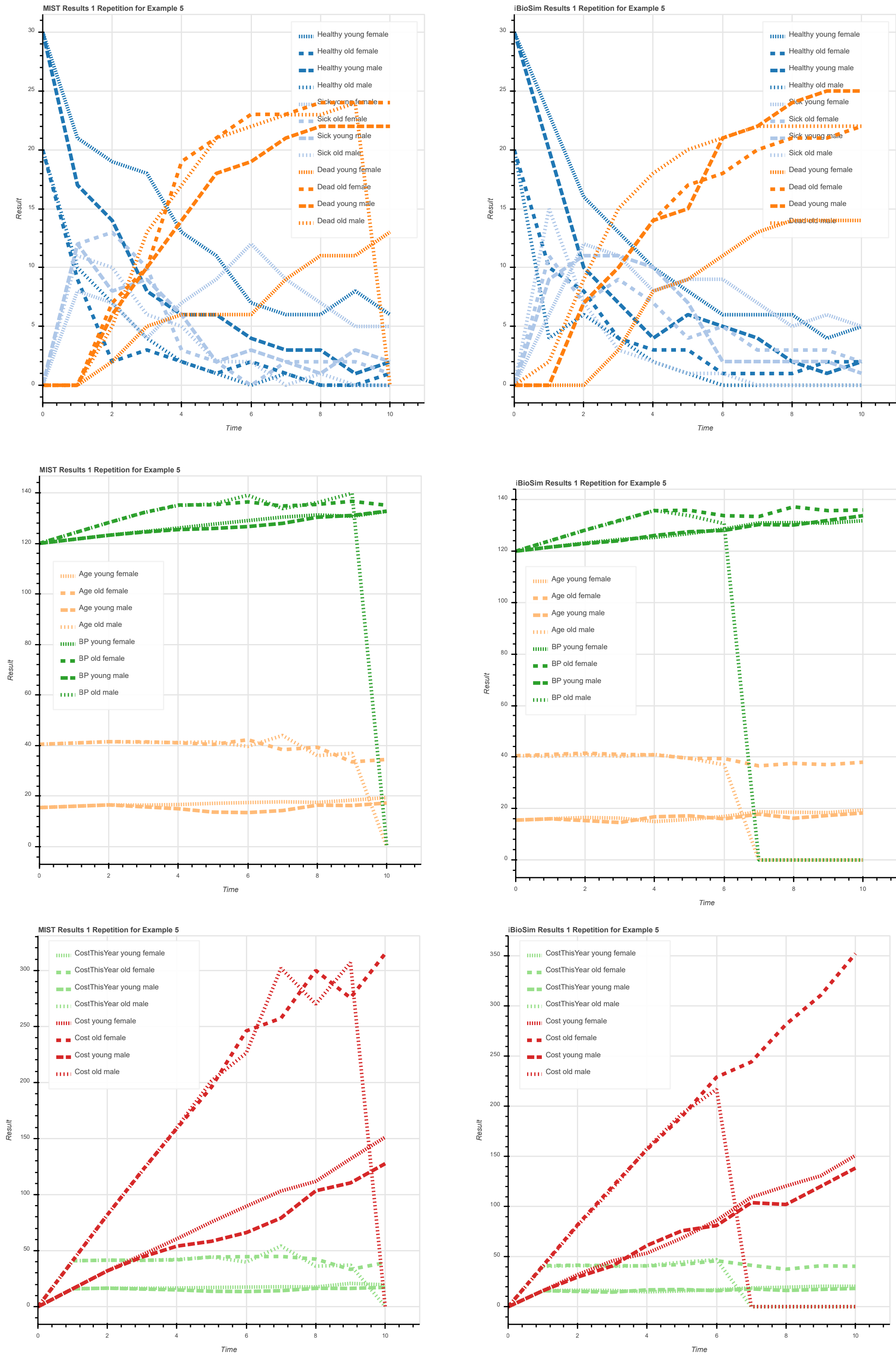
(a)

Figure 6.6: This figure shows the results comparison between MIST and iBioSim for one run. (a) Examples 1, 2, and 3. (b) Example 4. (c) Example 5.



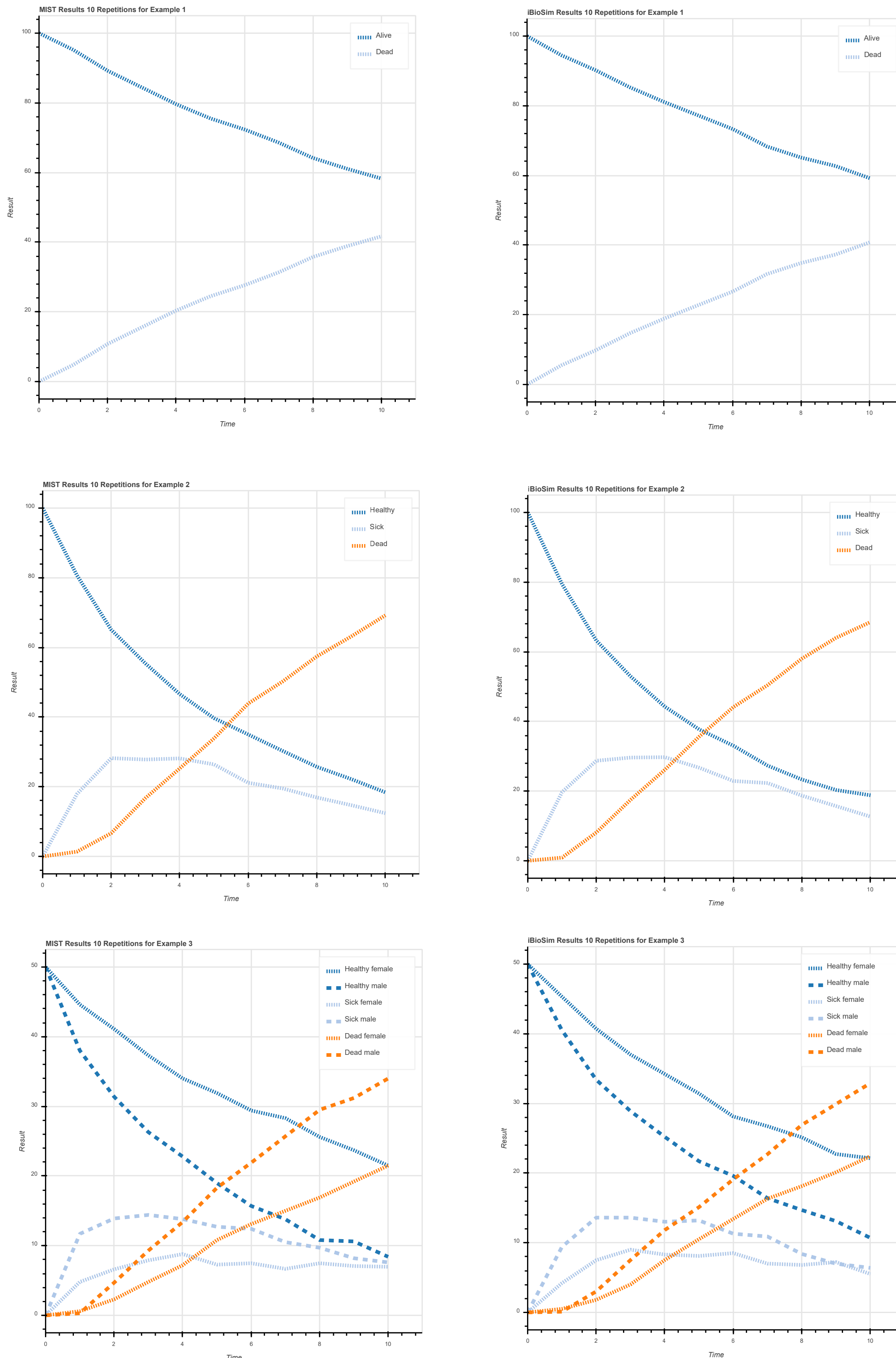
(b)

Figure 6.6: (continued) This figure shows the results comparison between `MIST` and `iBioSim` for one run. (a) Examples 1, 2, and 3. (b) Example 4. (c) Example 5.



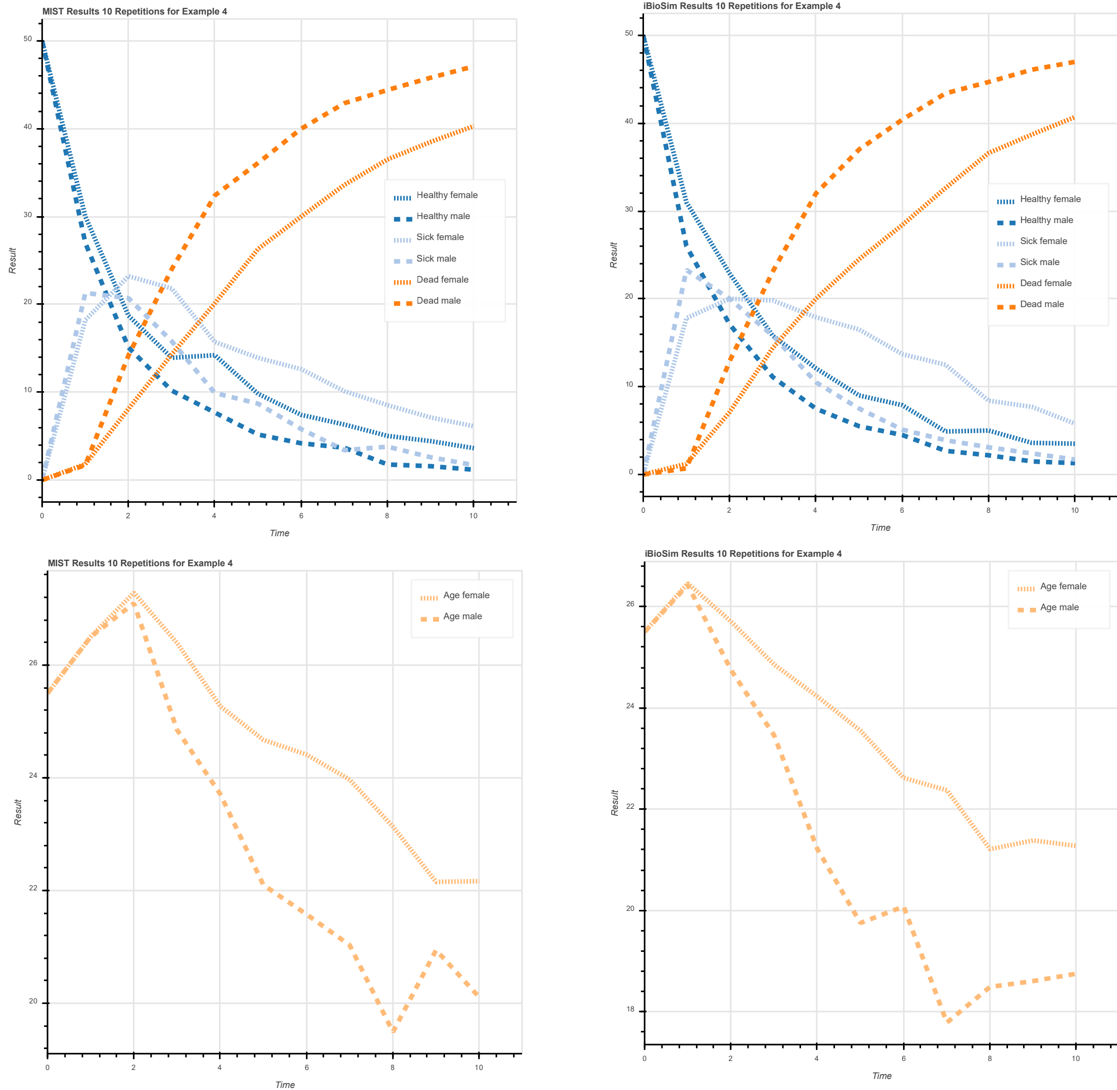
(c)

Figure 6.6: (continued) This figure shows the results comparison between `MIST` and `iBioSim` for one run. (a) Examples 1, 2, and 3. (b) Example 4. (c) Example 5.



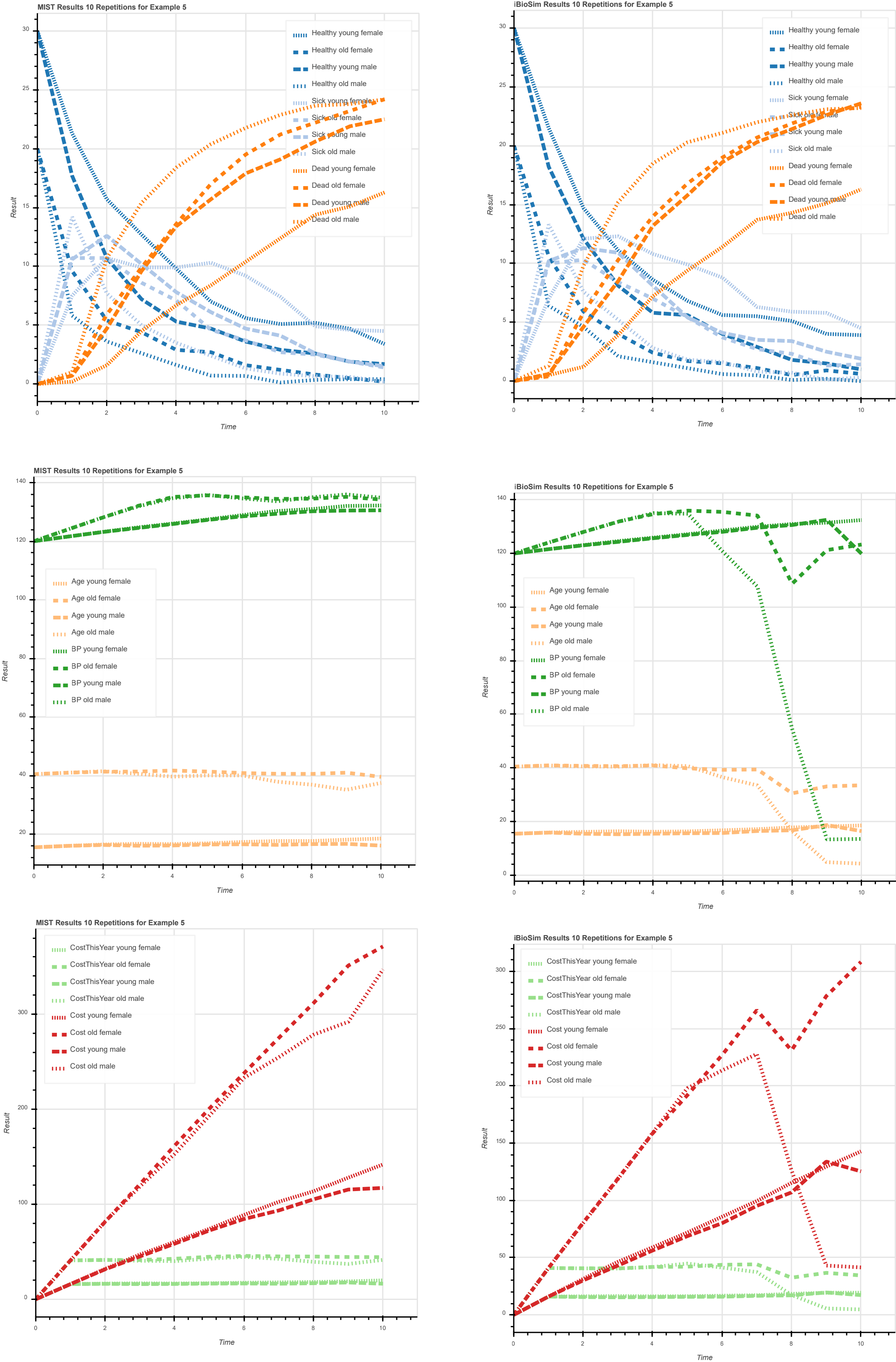
(a)

Figure 6.7: This figure shows the results comparison between MIST and iBioSim for 10 runs. (a) Examples 1, 2, and 3. (b) Example 4. (c) Example 5.



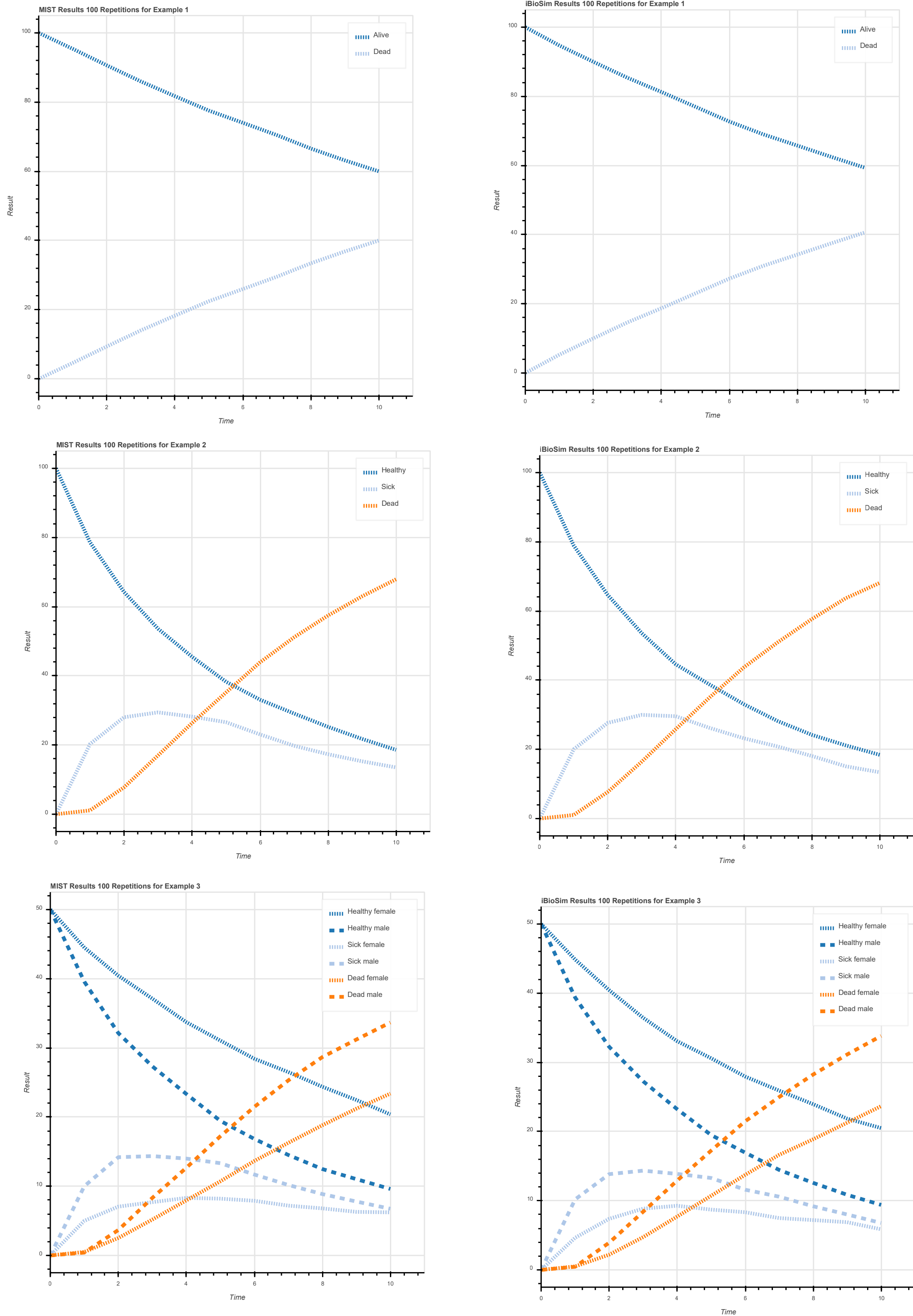
(b)

Figure 6.7: (continued) This figure shows the results comparison between `MIST` and `iBioSim` for 10 runs. (a) Examples 1, 2, and 3. (b) Example 4. (c) Example 5.



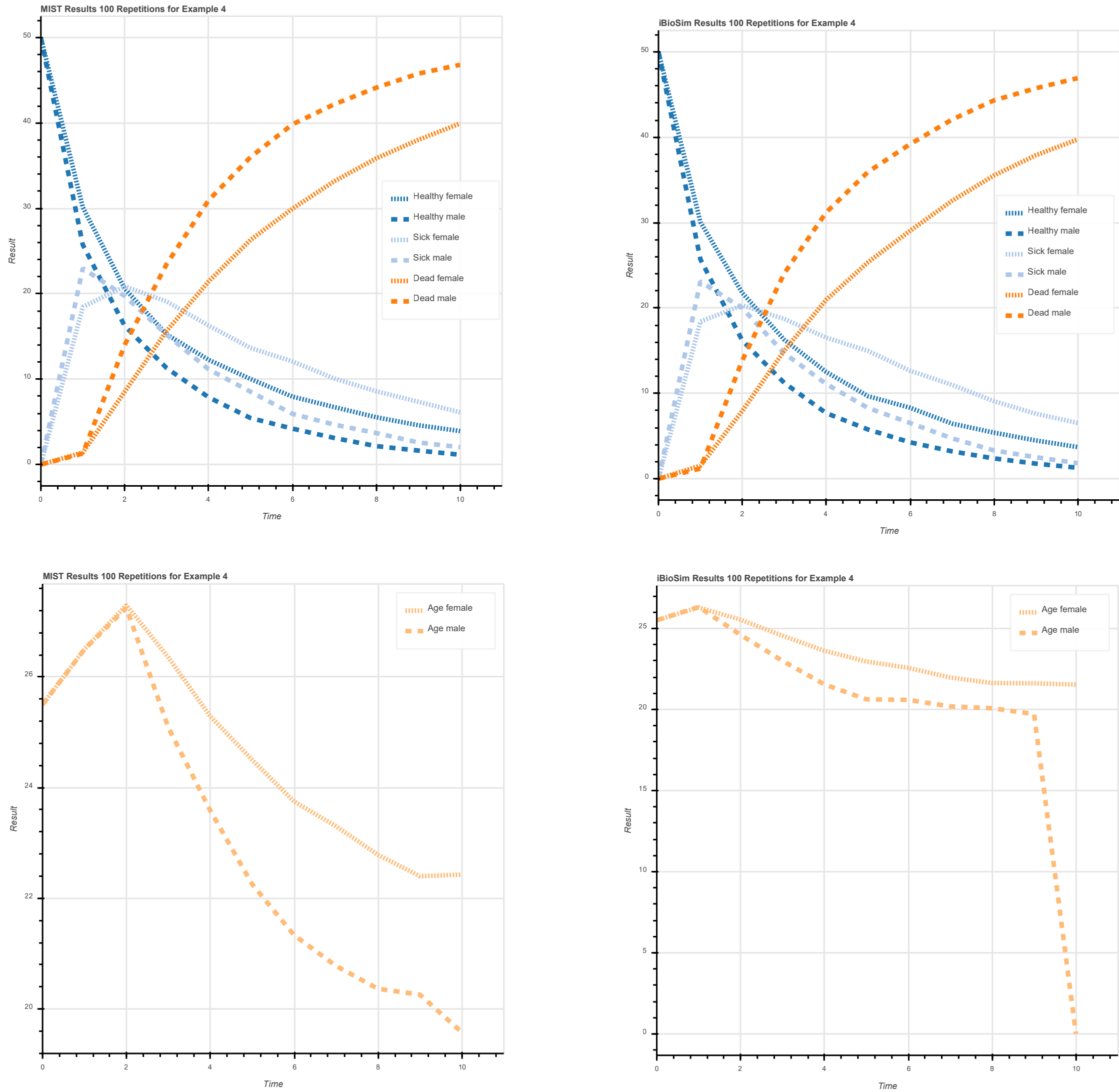
(c)

Figure 6.7: (continued) This figure shows the results comparison between MIST and iBioSim for 10 runs. (a) Examples 1, 2, and 3. (b) Example 4. (c) Example 5.



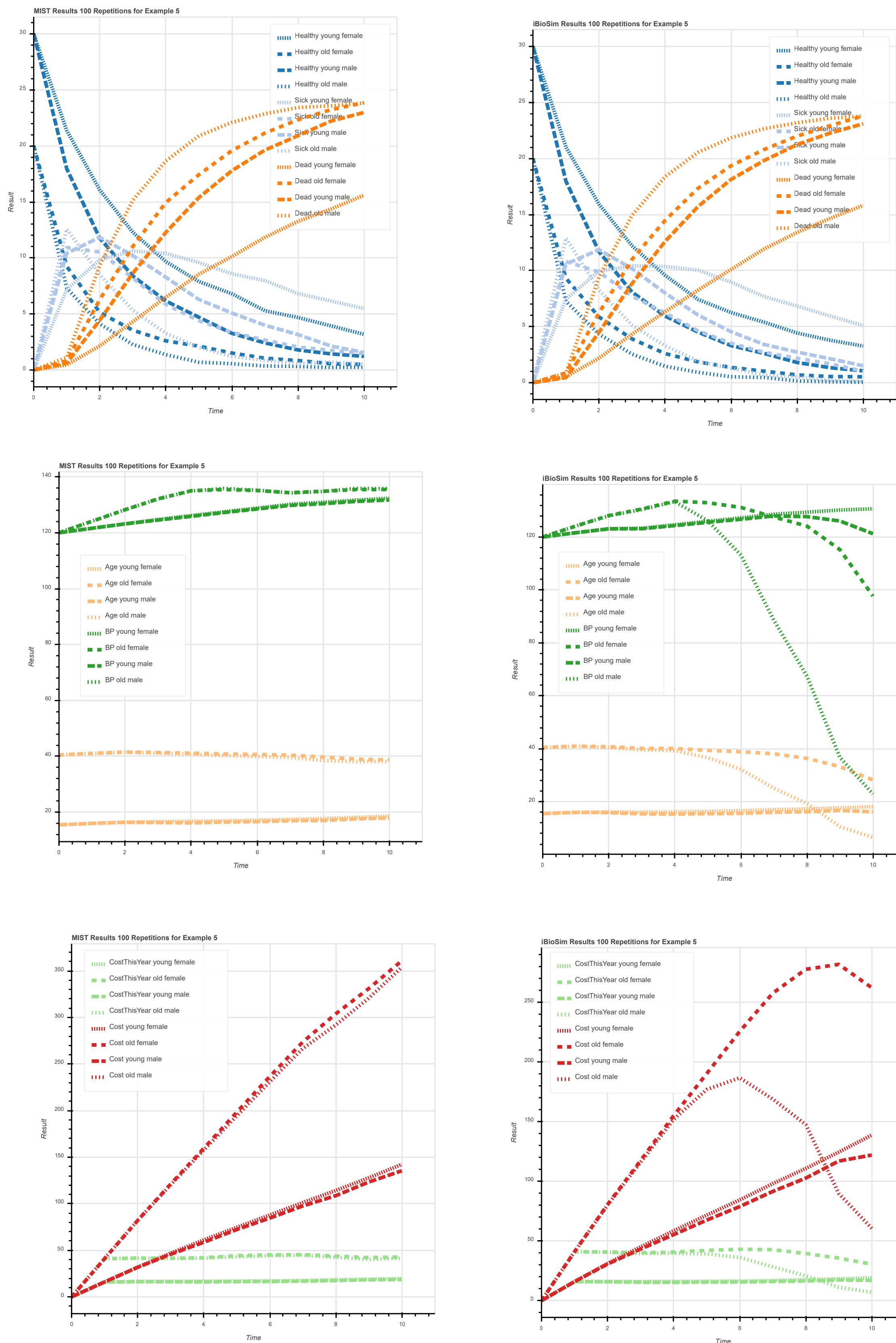
(a)

Figure 6.8: This figure shows the results comparison between `MIST` and `iBioSim` for 100 runs. (a) Examples 1, 2, and 3. (b) Example 4. (c) Example 5.



(b)

Figure 6.8: (continued) This figure shows the results comparison between `MIST` and `iBioSim` for 100 runs. (a) Examples 1, 2, and 3. (b) Example 4. (c) Example 5.



(c)

Figure 6.8: (continued) This figure shows the results comparison between `MIST` and `iBioSim` for 100 runs. (a) Examples 1, 2, and 3. (b) Example 4. (c) Example 5.

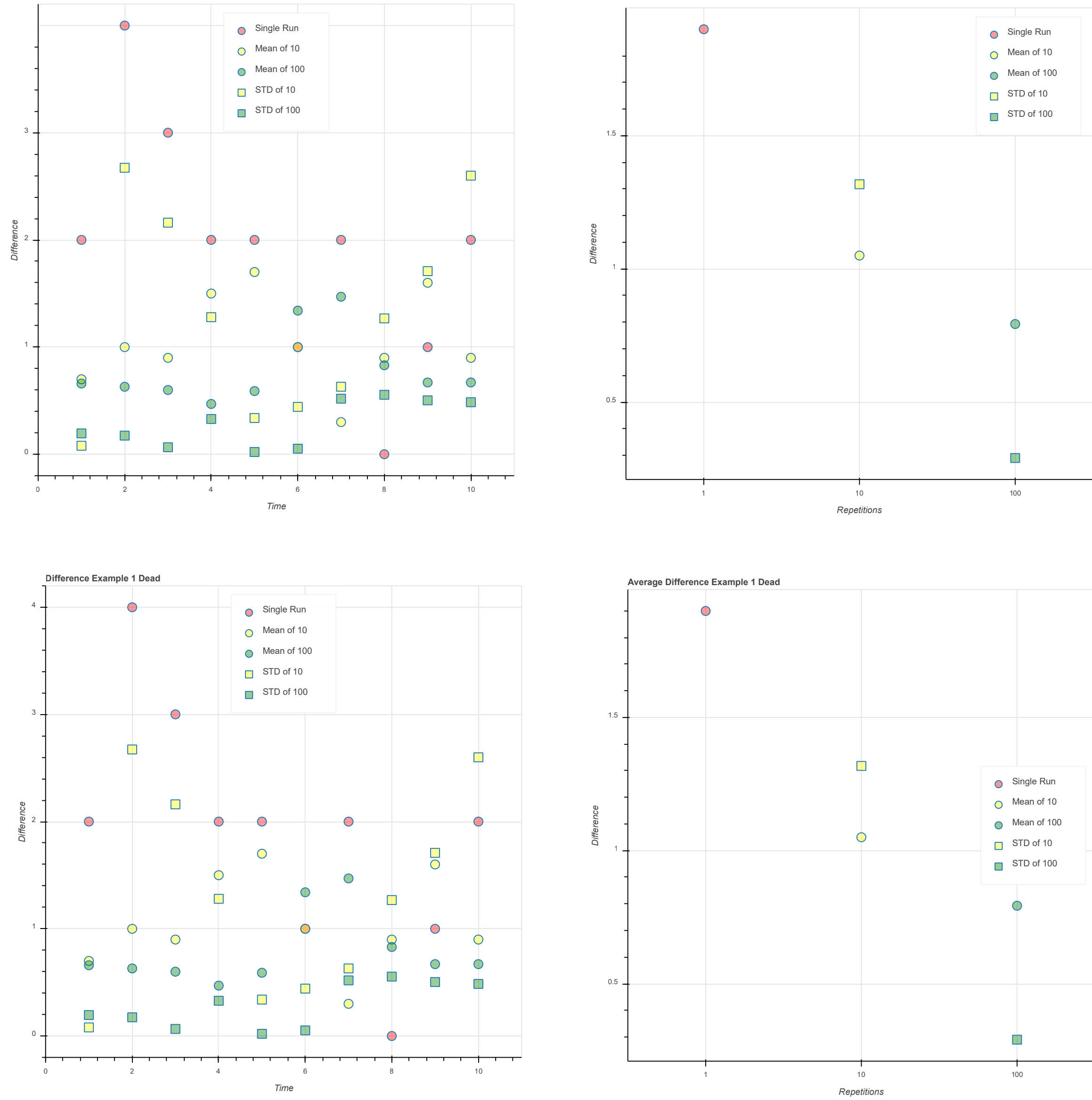


Figure 6.9: Statistical analysis for example 1.

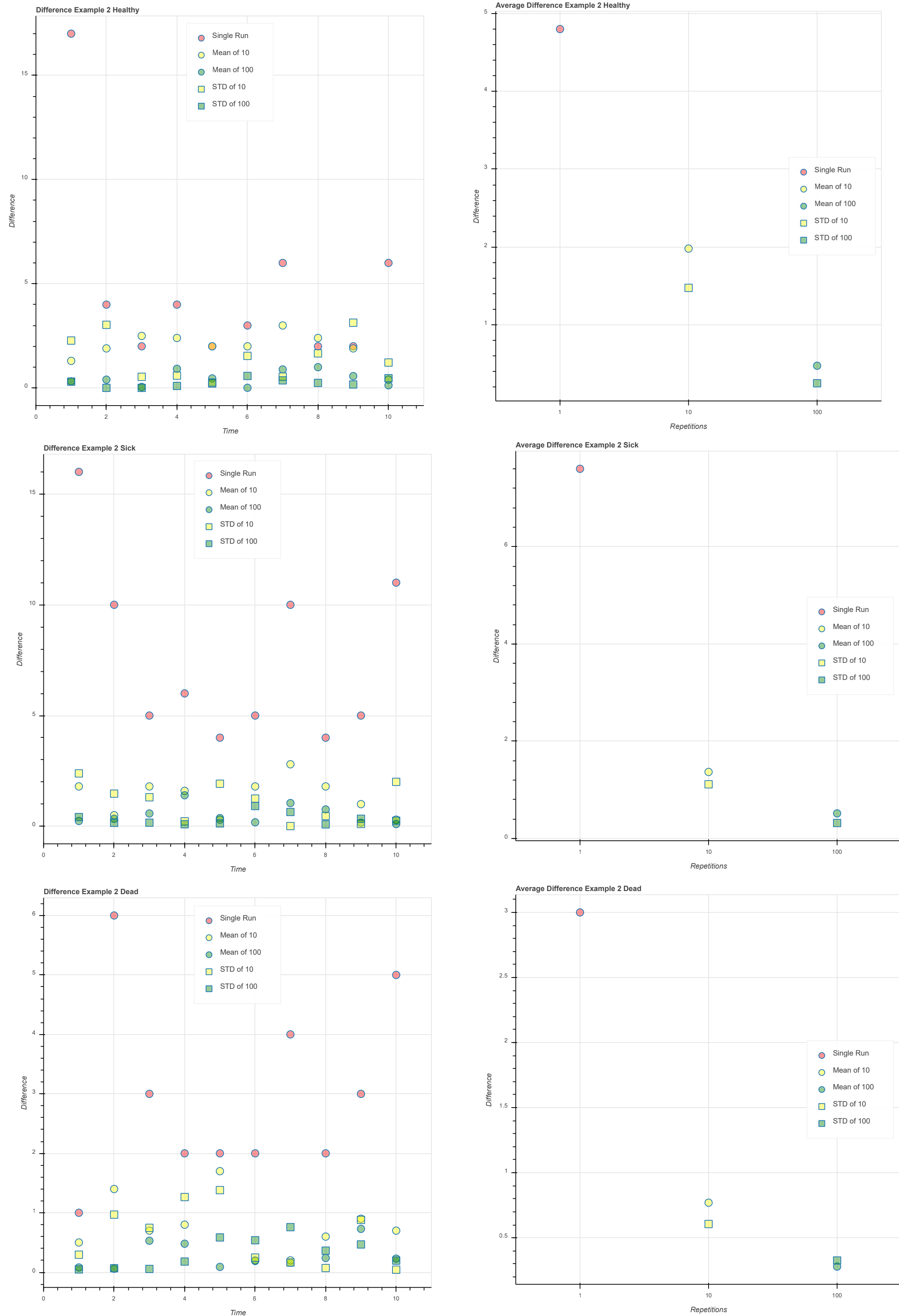


Figure 6.10: Statistical analysis for example 2.

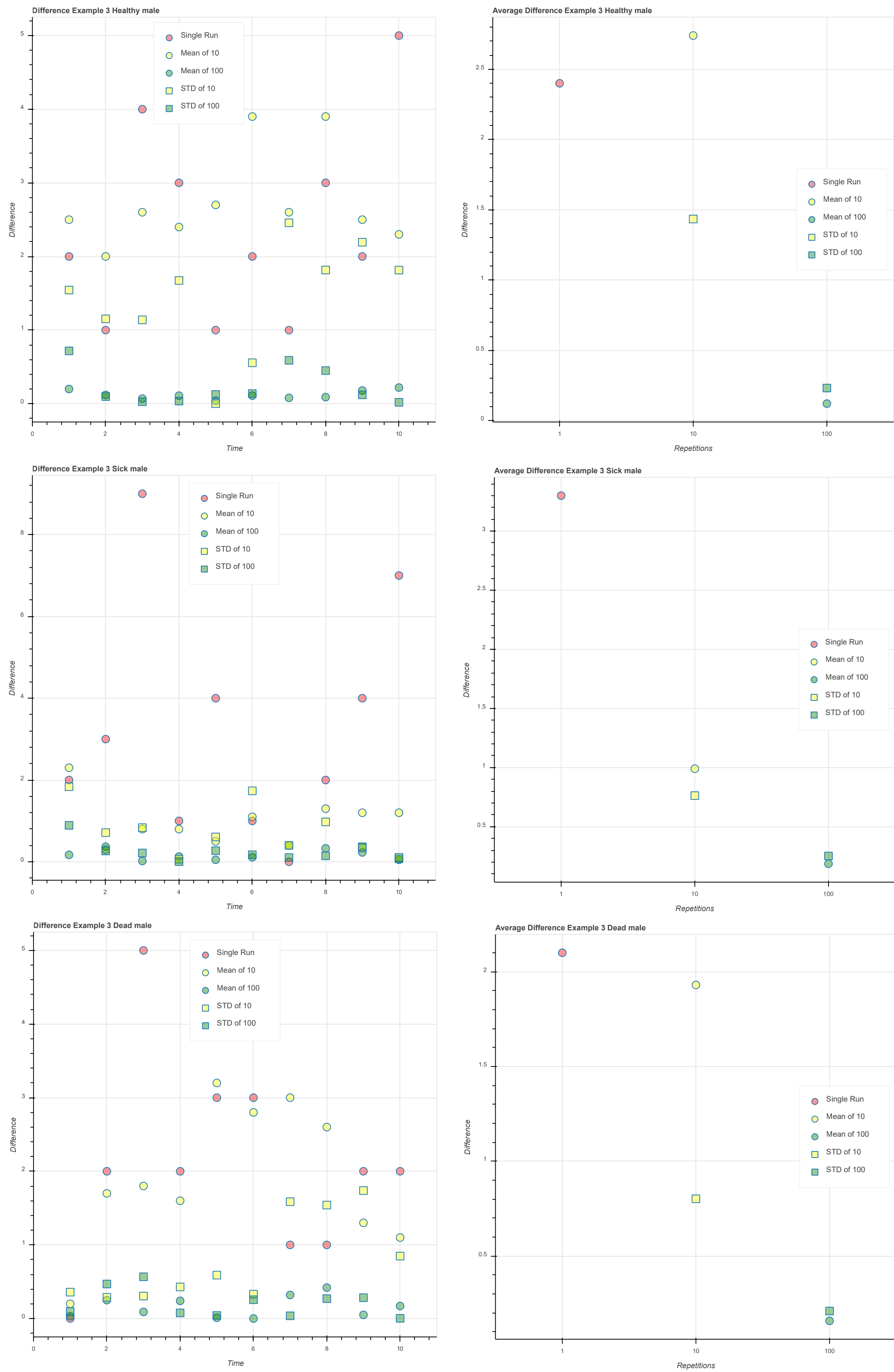


Figure 6.11: Statistical analysis for males in example 3.

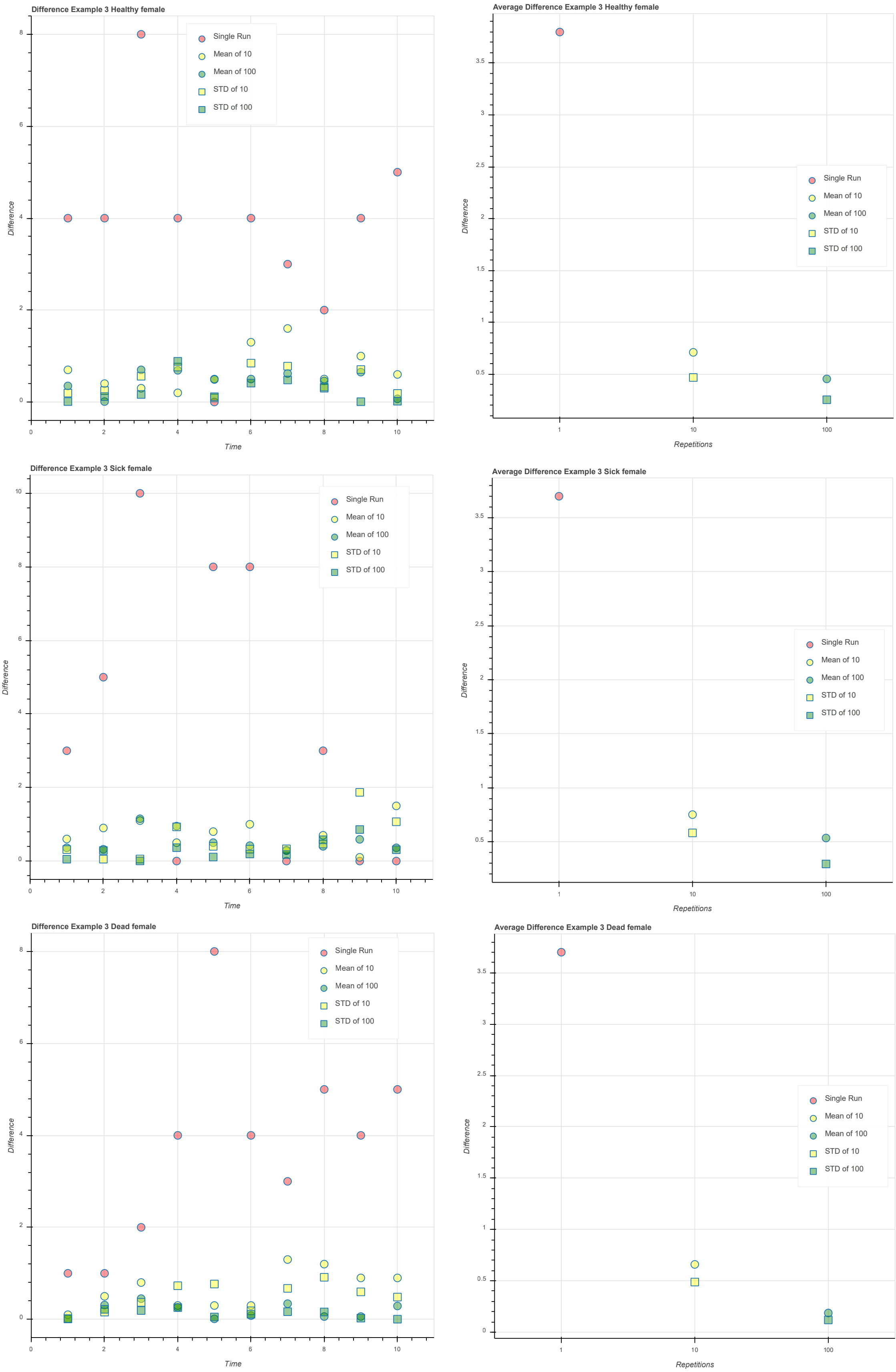
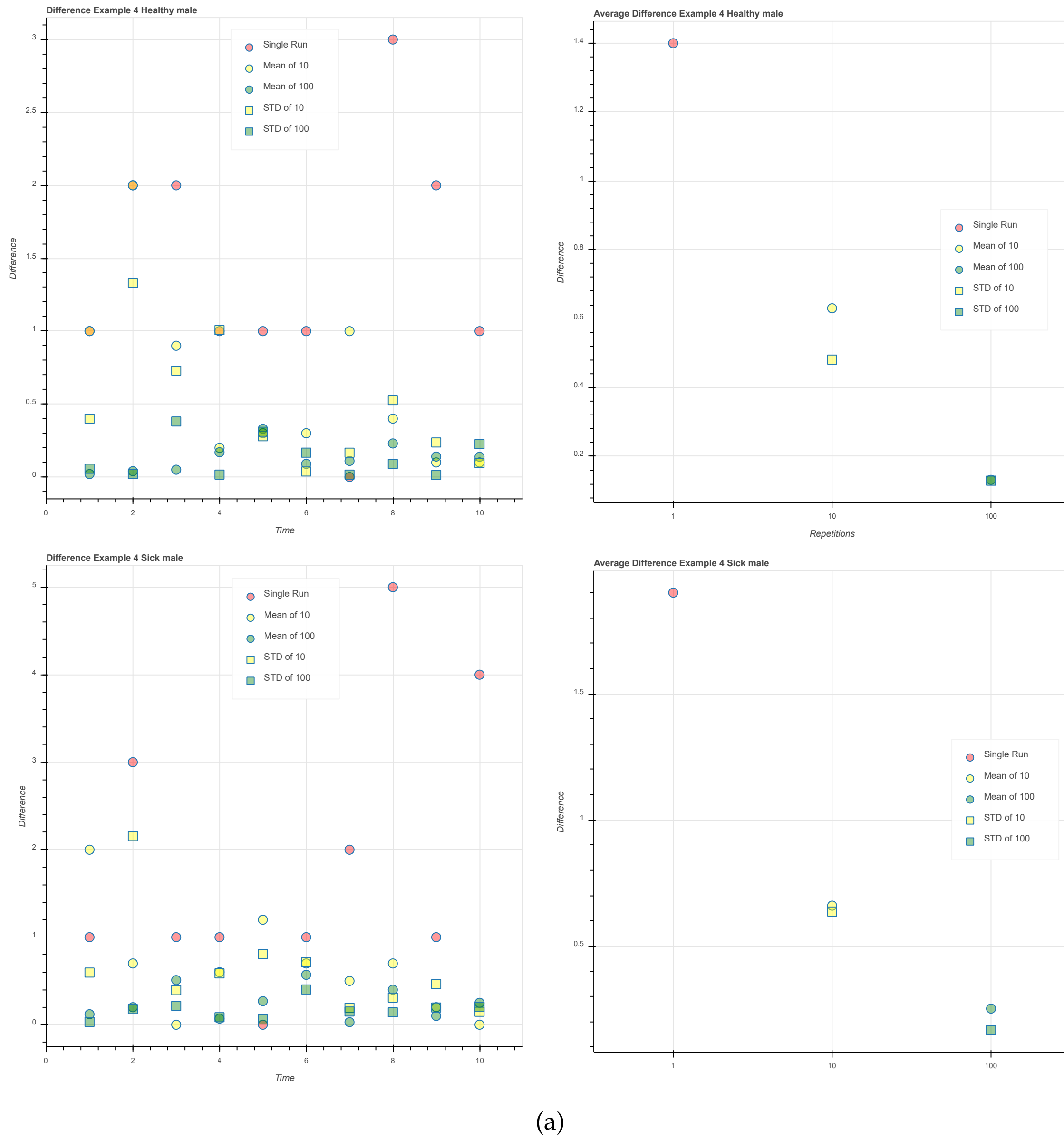
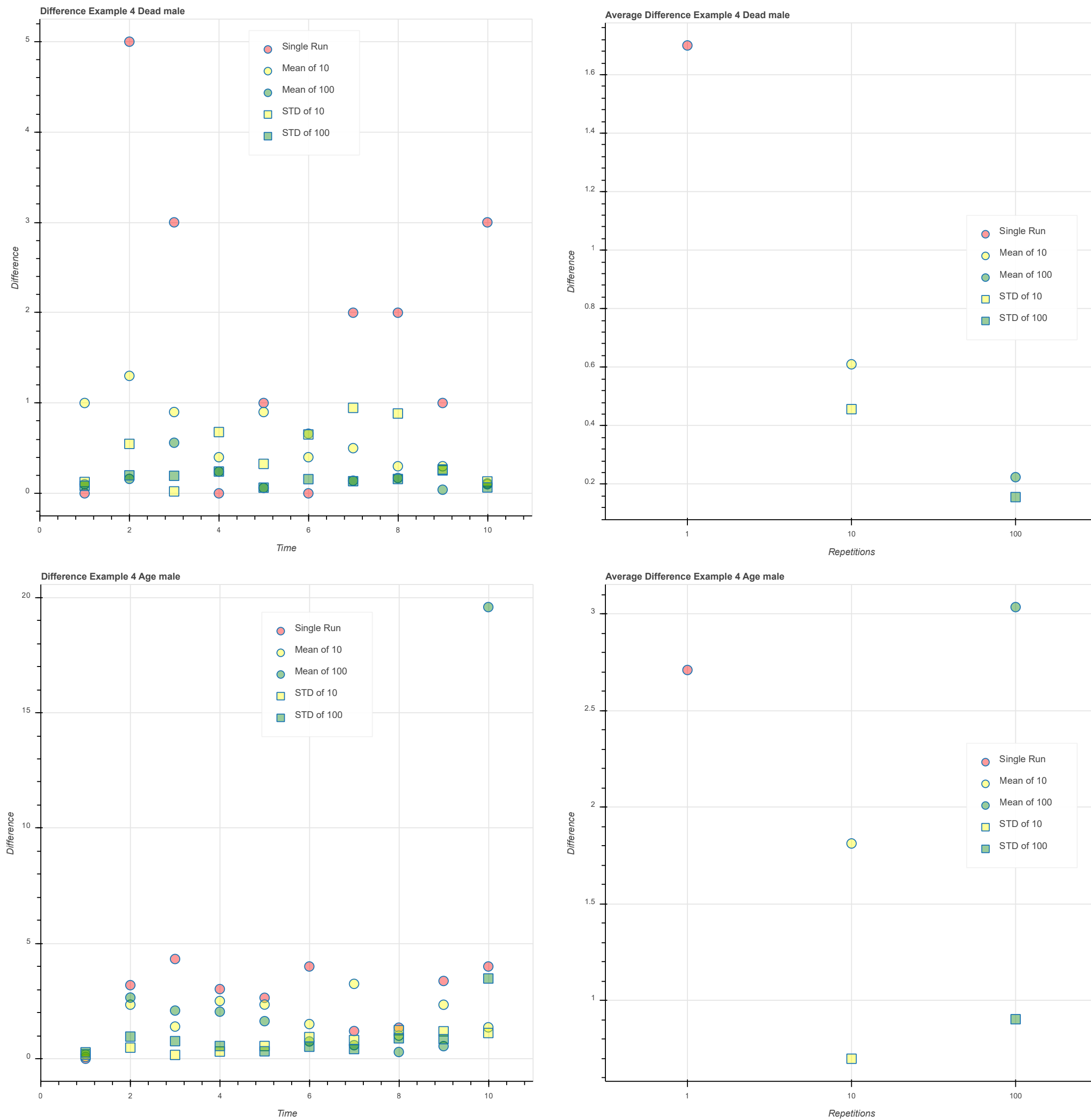


Figure 6.12: Statistical analysis for females in example 3.



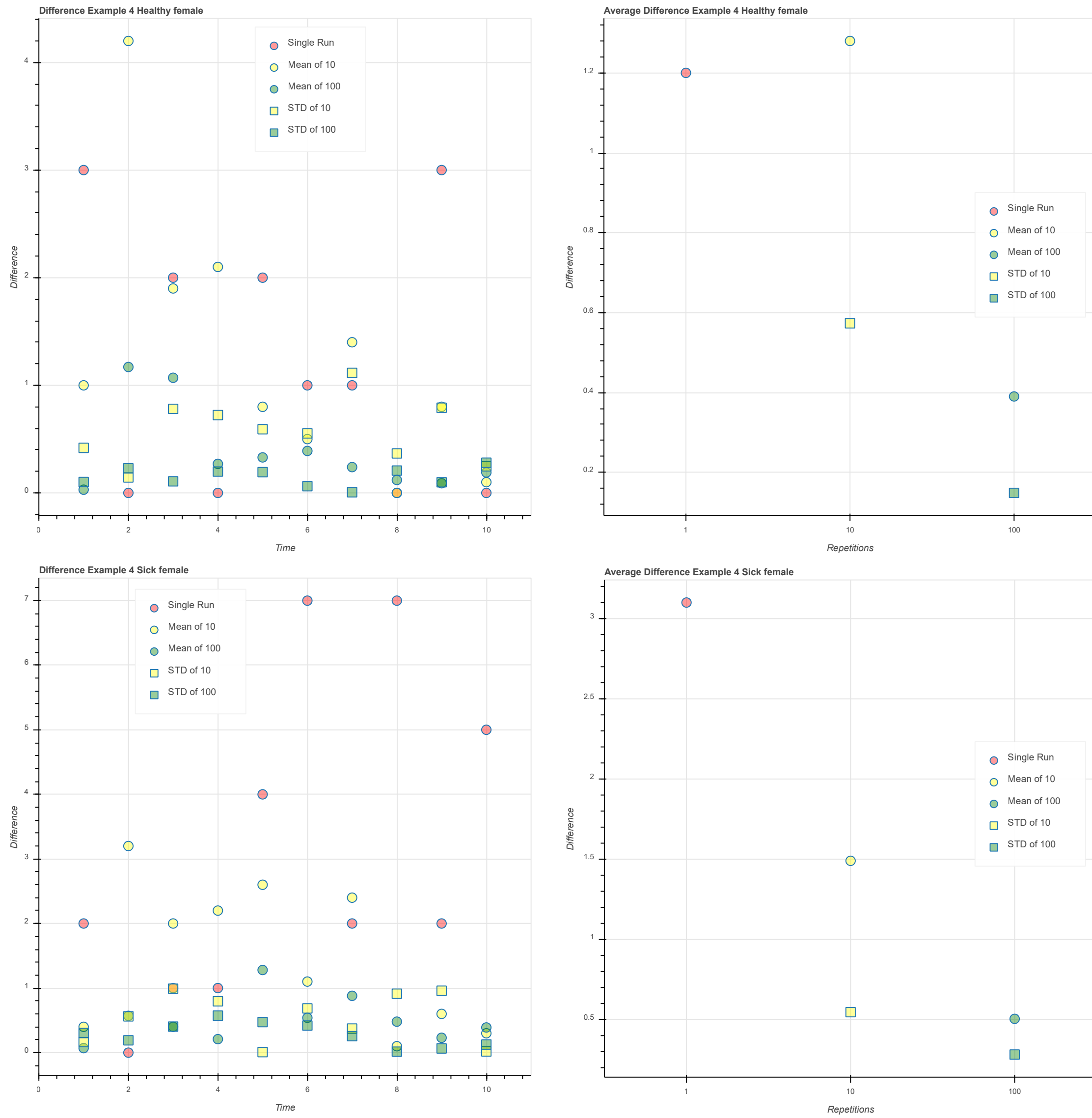
(a)

Figure 6.13: Statistical analysis for males in example 4. (a) Healthy and Sick. (b) Dead and Age.



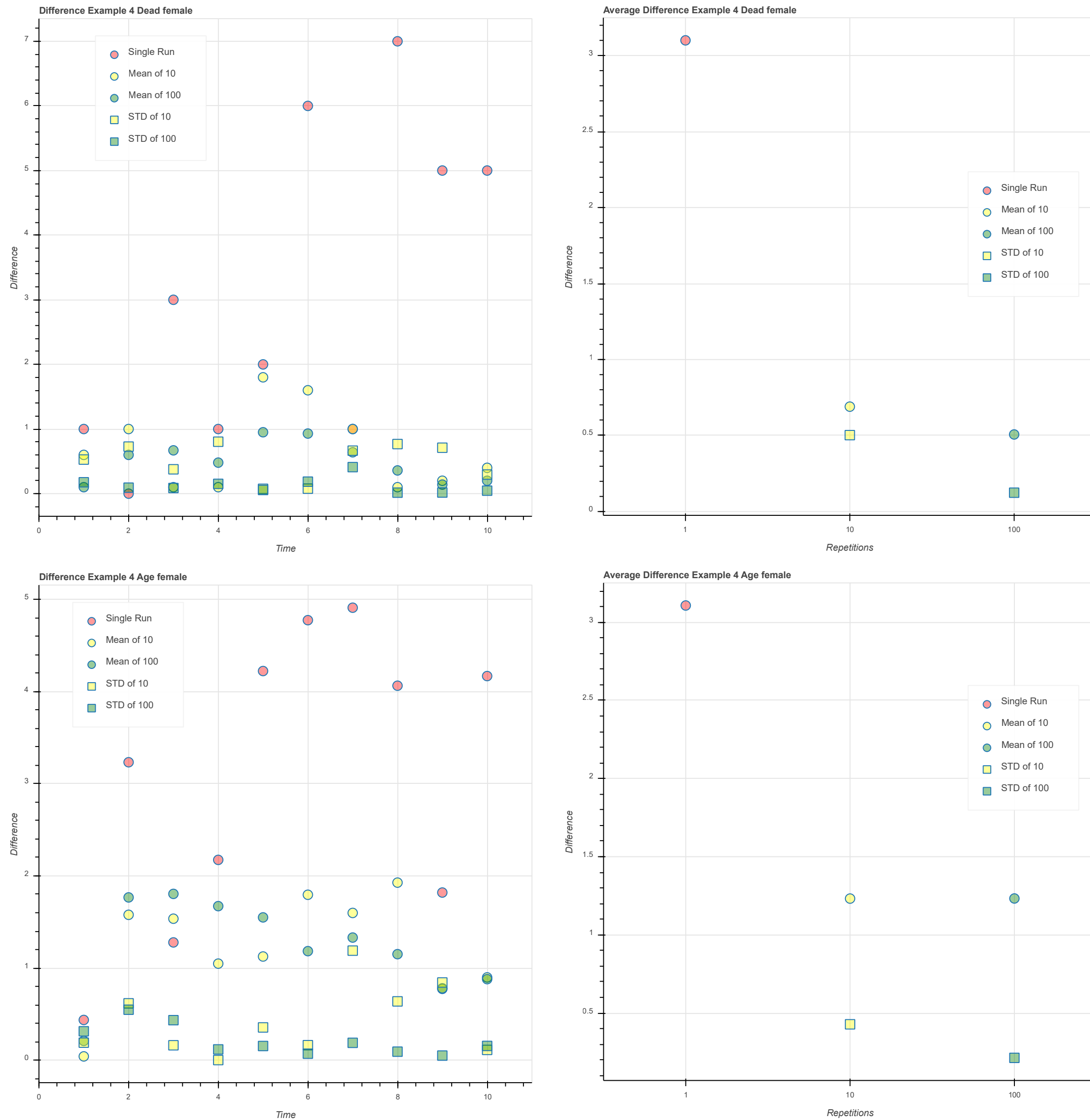
(b)

Figure 6.13: Statistical analysis for males in example 4. (a) Healthy and Sick. (b) Dead and Age.



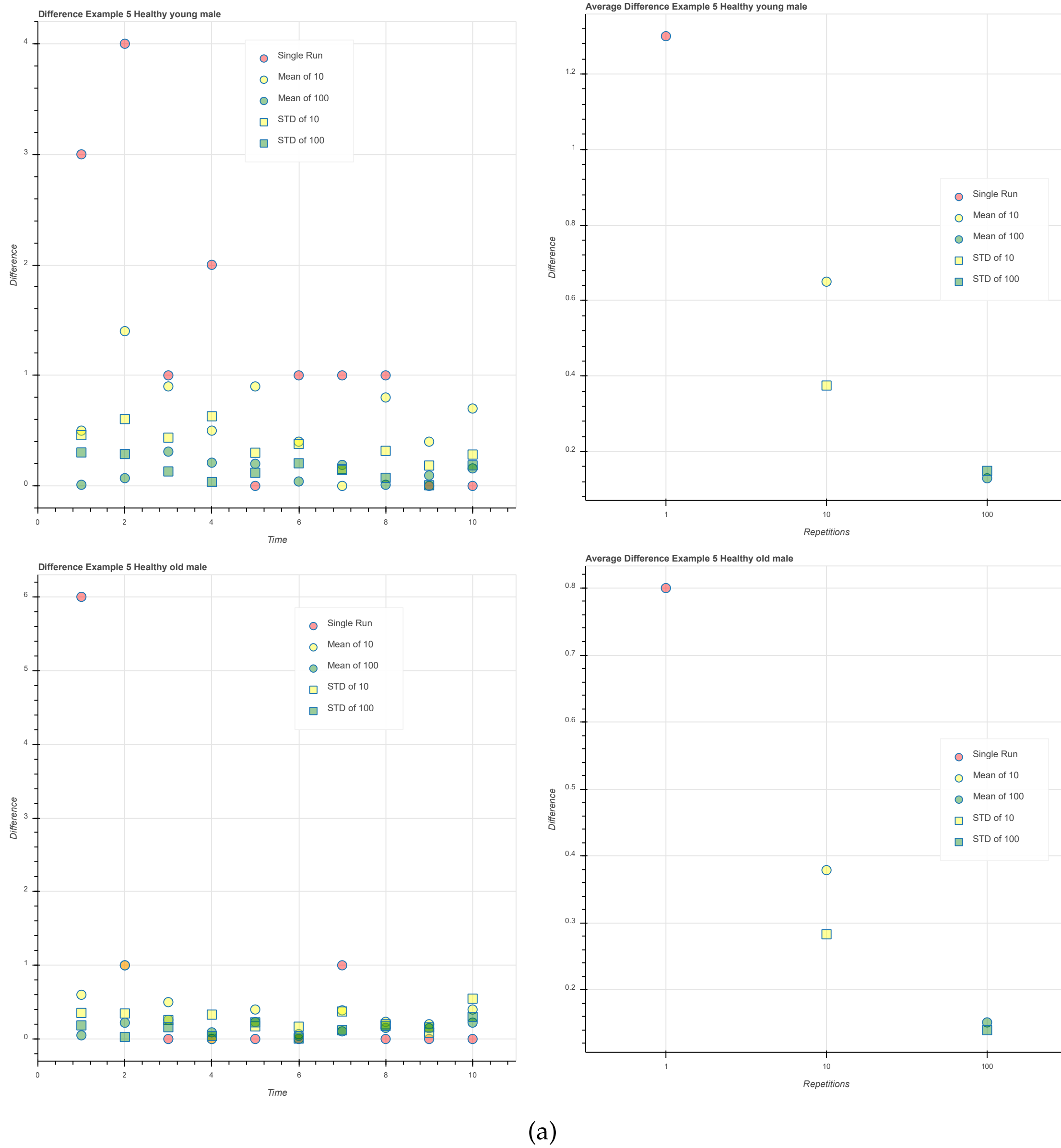
(a)

Figure 6.14: Statistical analysis for females in example 4. (a) Healthy and Sick. (b) Dead and Age.



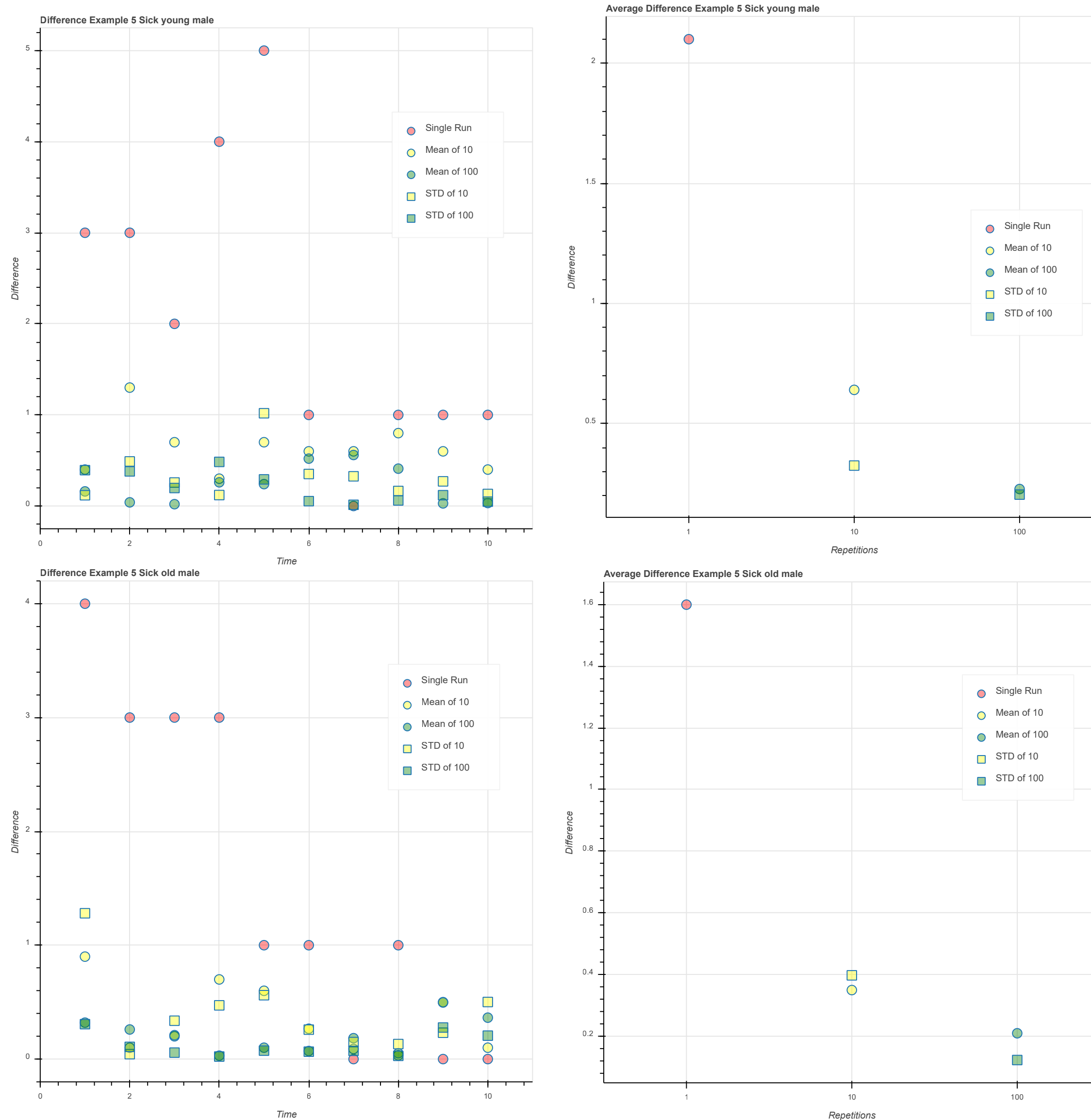
(b)

Figure 6.14: (continued) Statistical analysis for females in example 4. (a) Healthy and Sick. (b) Dead and Age.



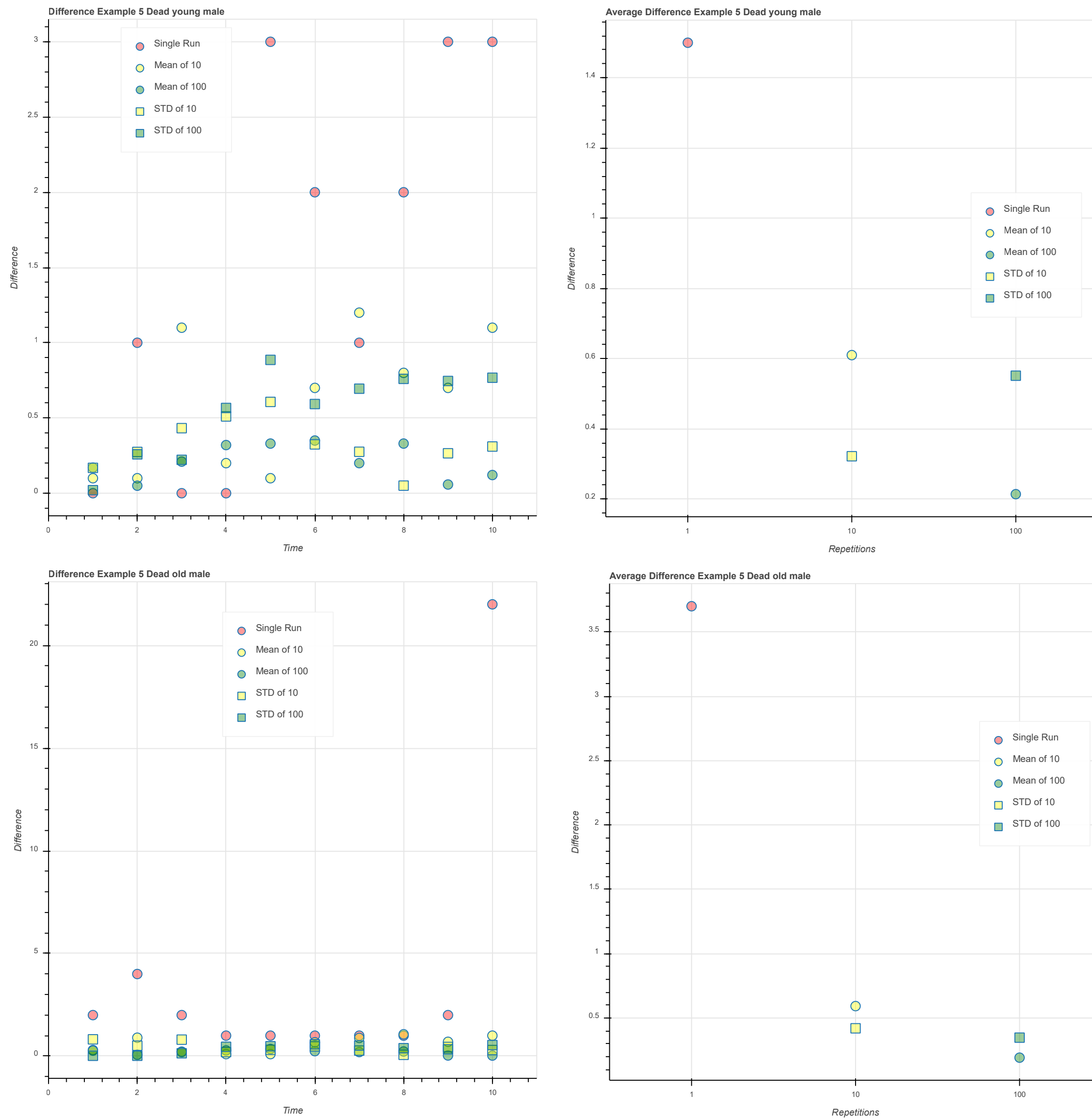
(a)

Figure 6.15: (continued) Statistical analysis for males in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



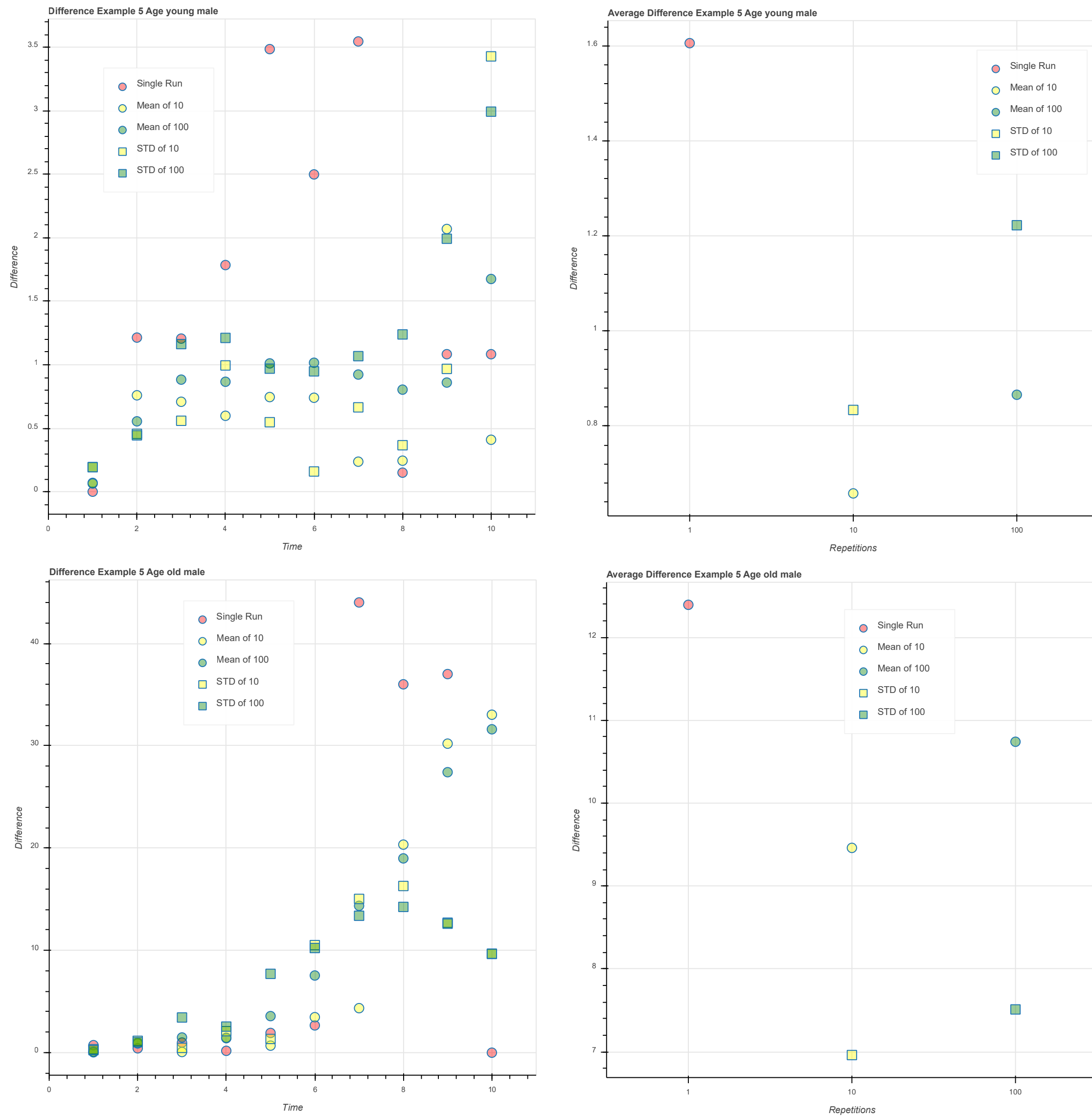
(b)

Figure 6.15: (continued) Statistical analysis for males in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



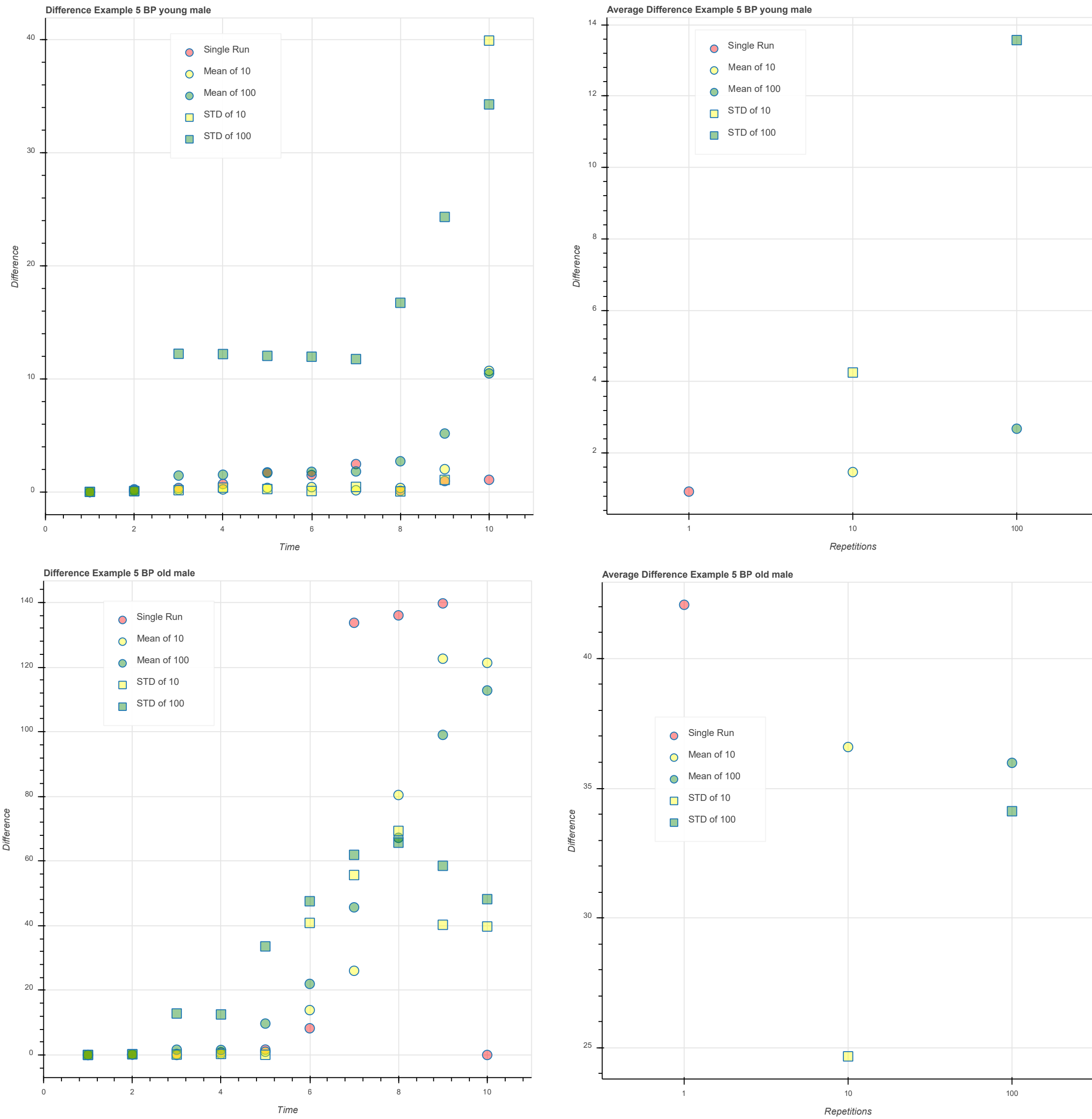
(c)

Figure 6.15: (continued) Statistical analysis for males in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



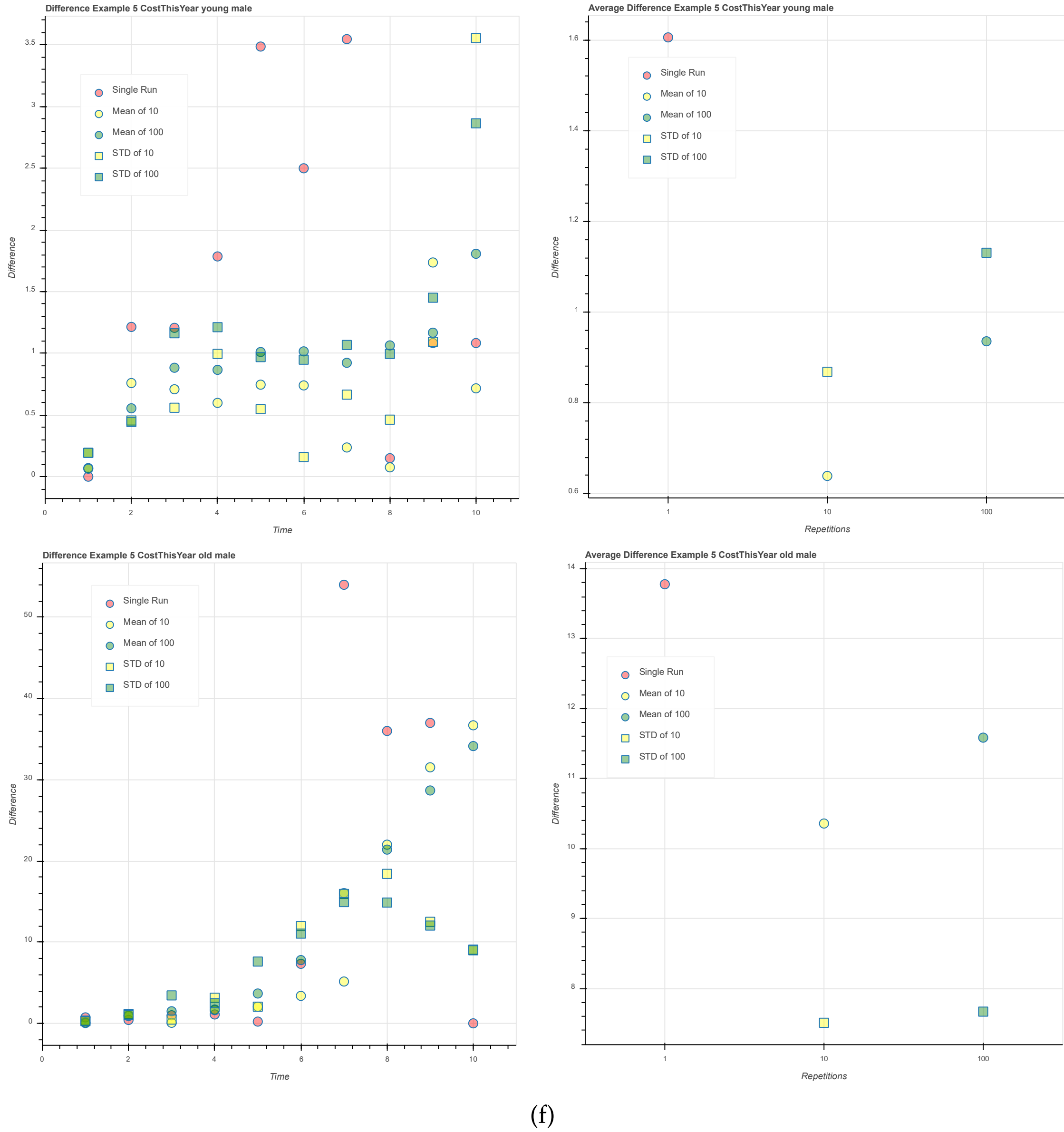
(d)

Figure 6.15: (continued) Statistical analysis for males in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



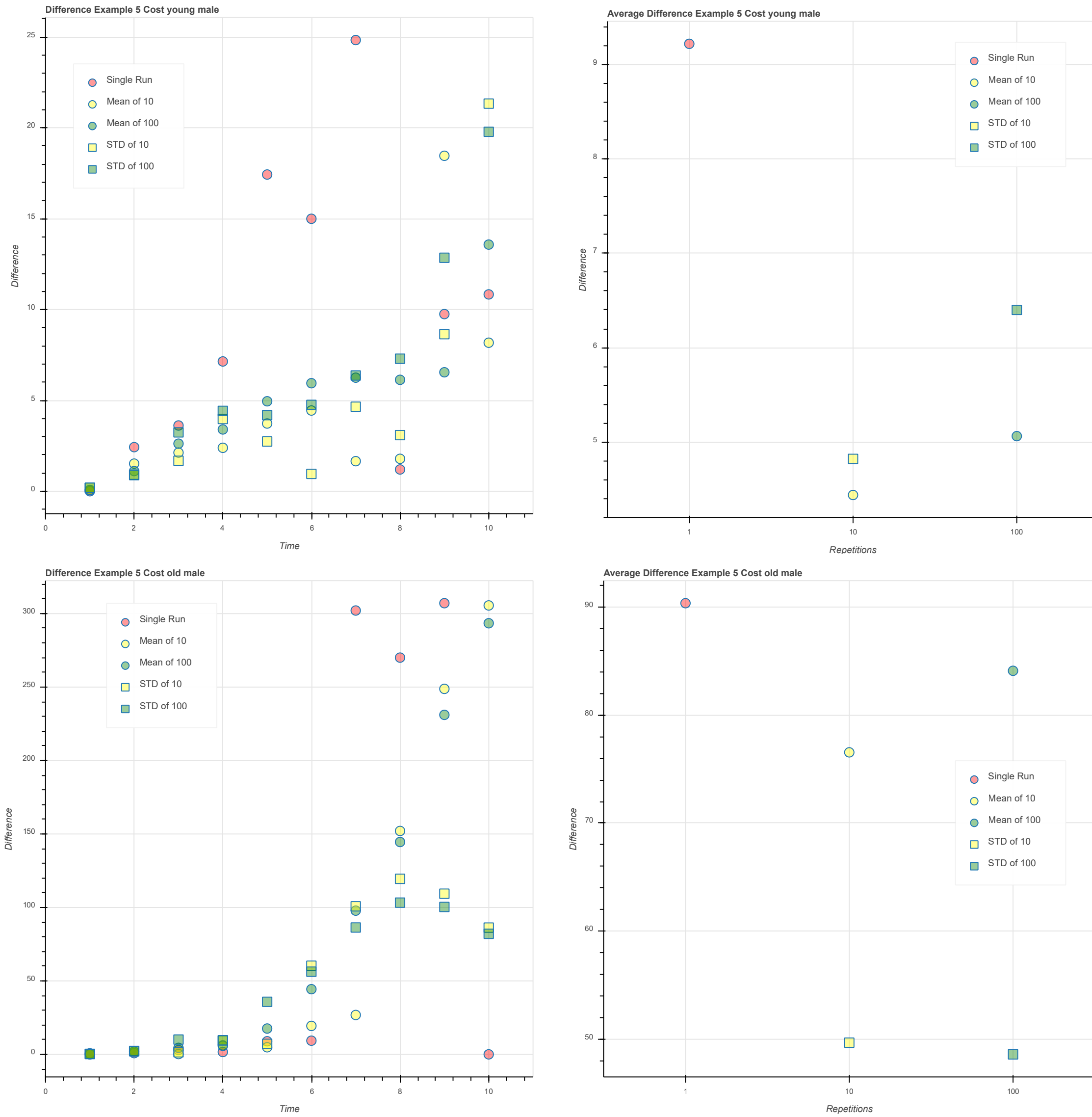
(e)

Figure 6.15: (continued) Statistical analysis for males in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



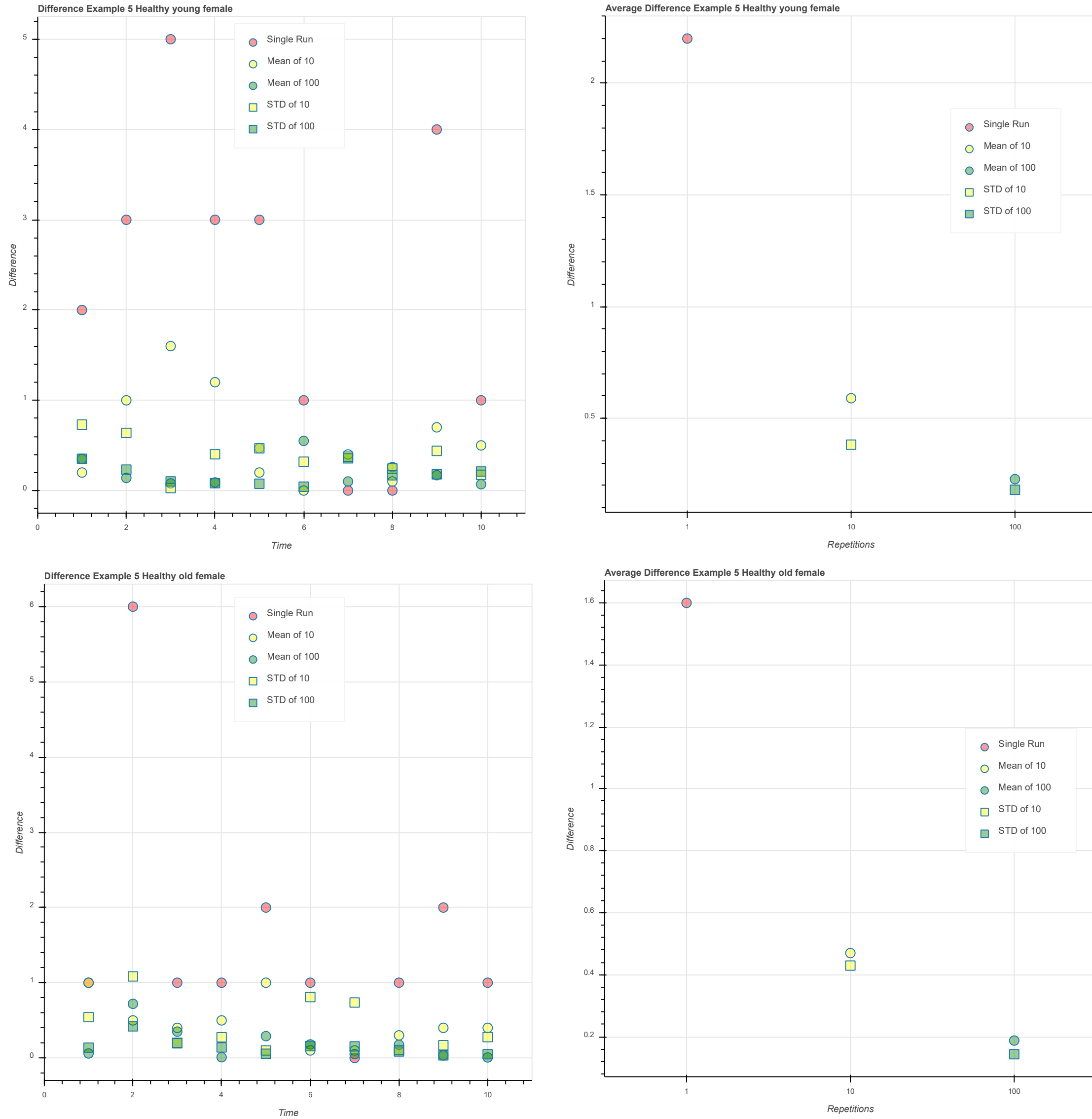
(f)

Figure 6.15: (continued) Statistical analysis for males in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



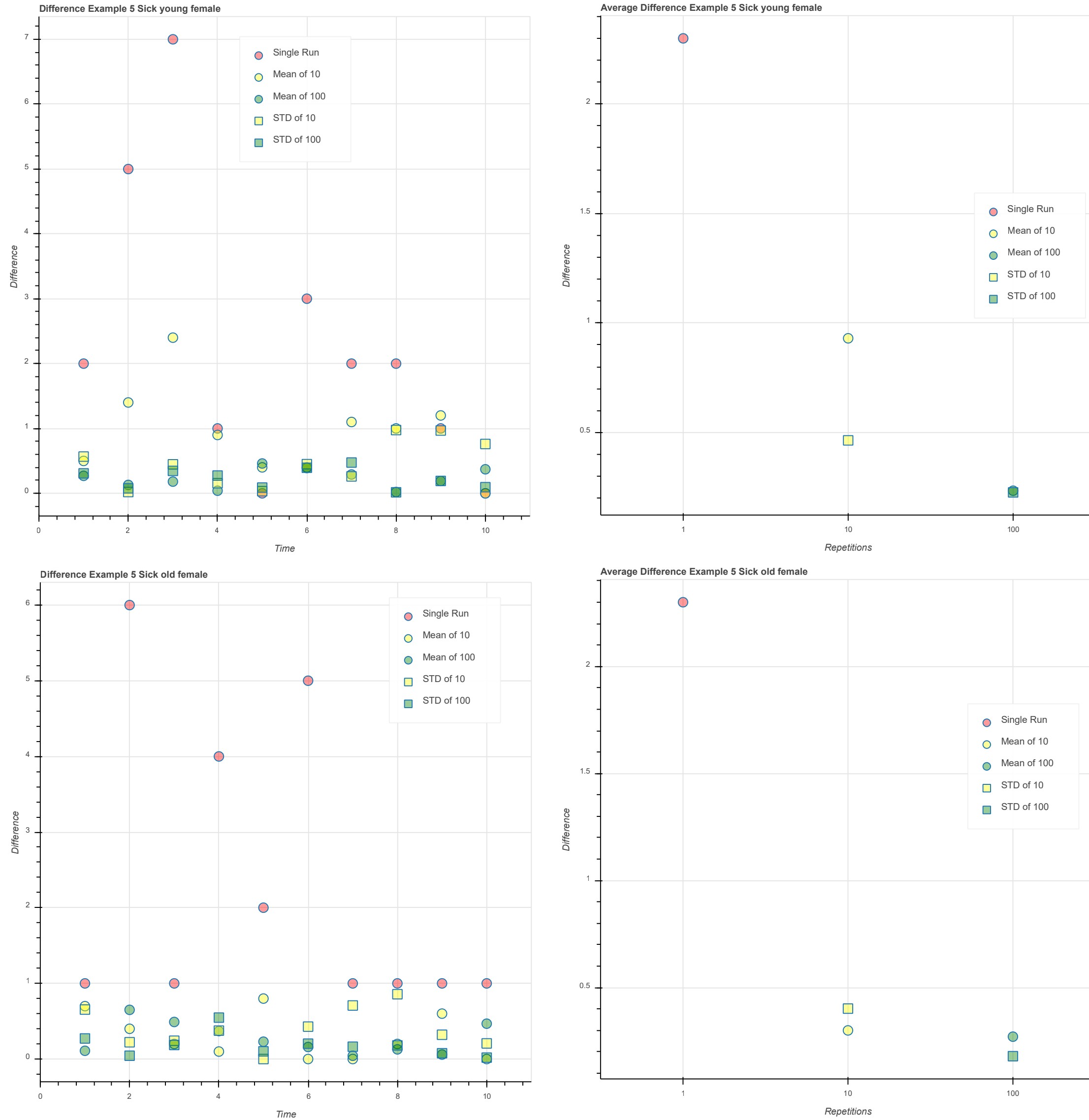
(g)

Figure 6.15: (continued) Statistical analysis for males in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



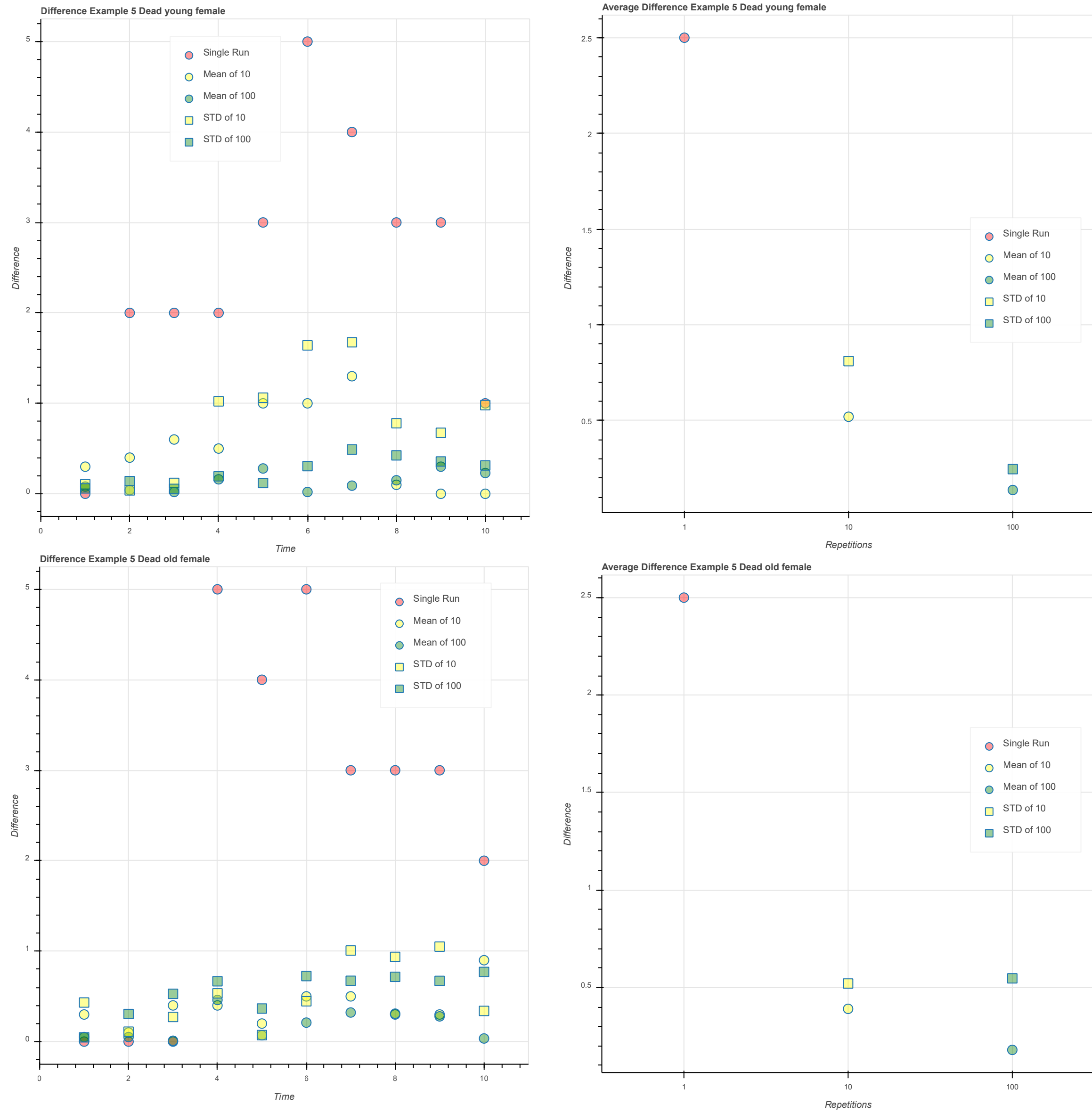
(a)

Figure 6.16: Statistical analysis for females in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



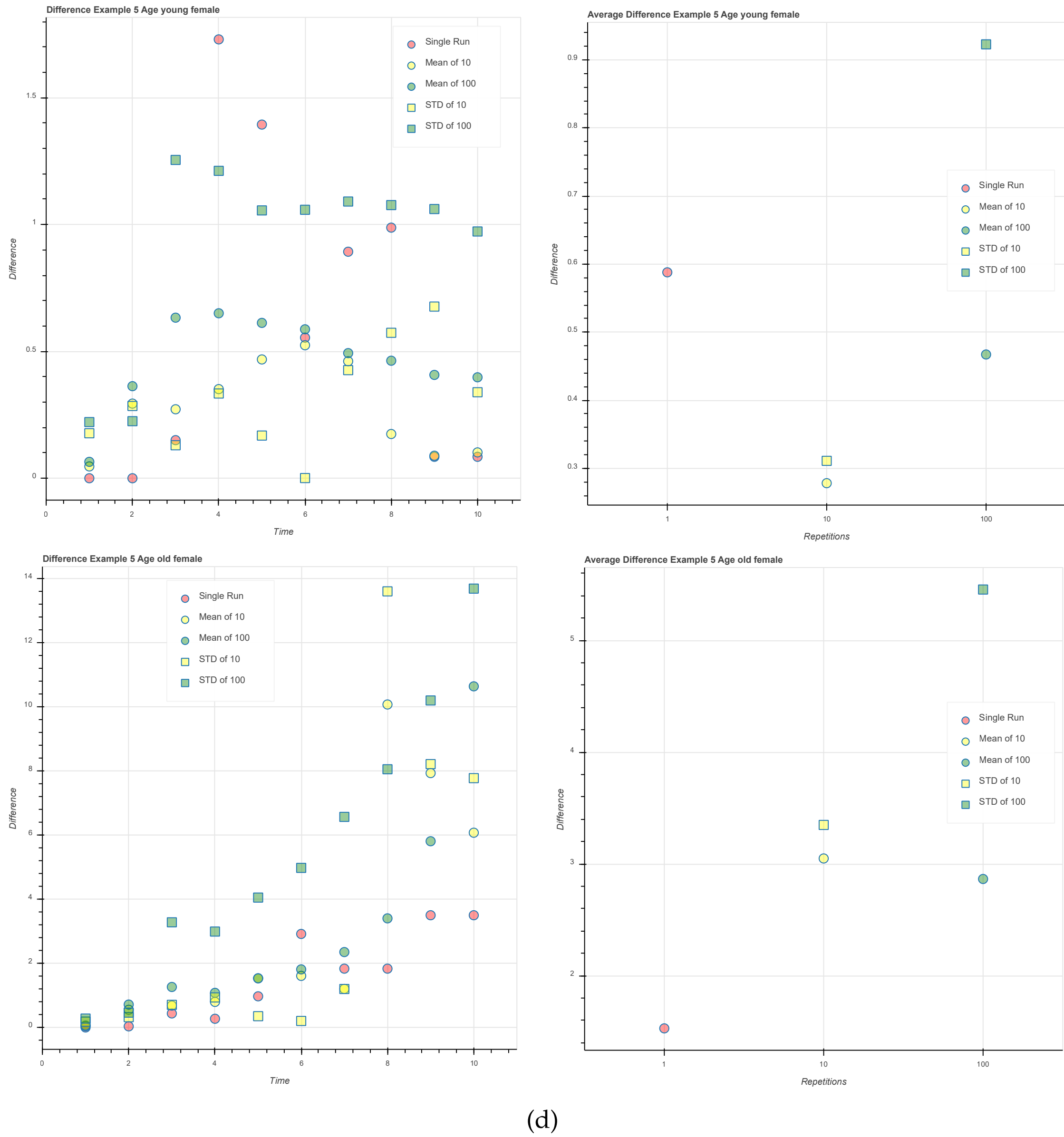
(b)

Figure 6.16: (continued) Statistical analysis for females in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



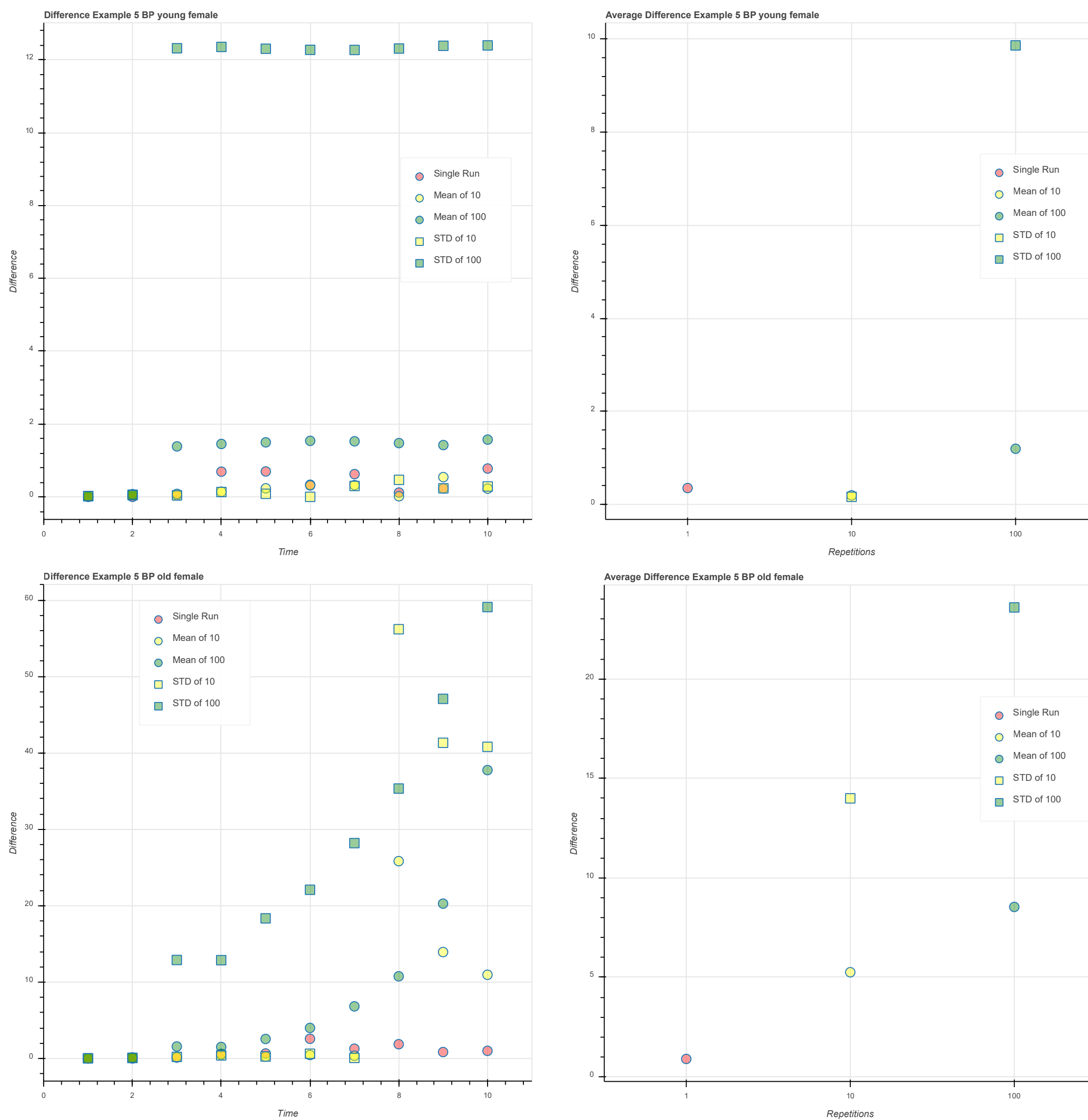
(c)

Figure 6.16: (continued) Statistical analysis for females in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



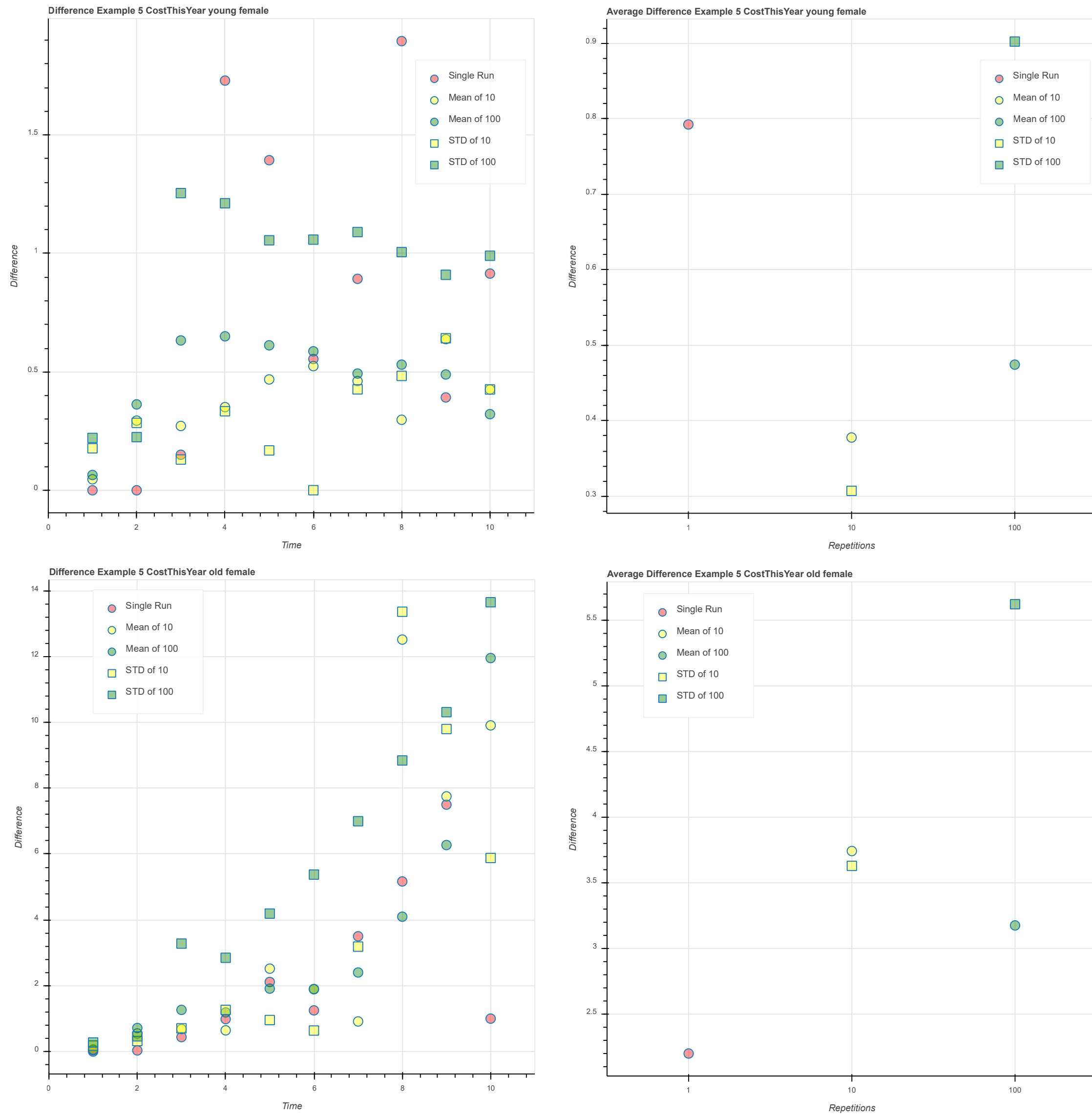
(d)

Figure 6.16: (continued) Statistical analysis for females in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



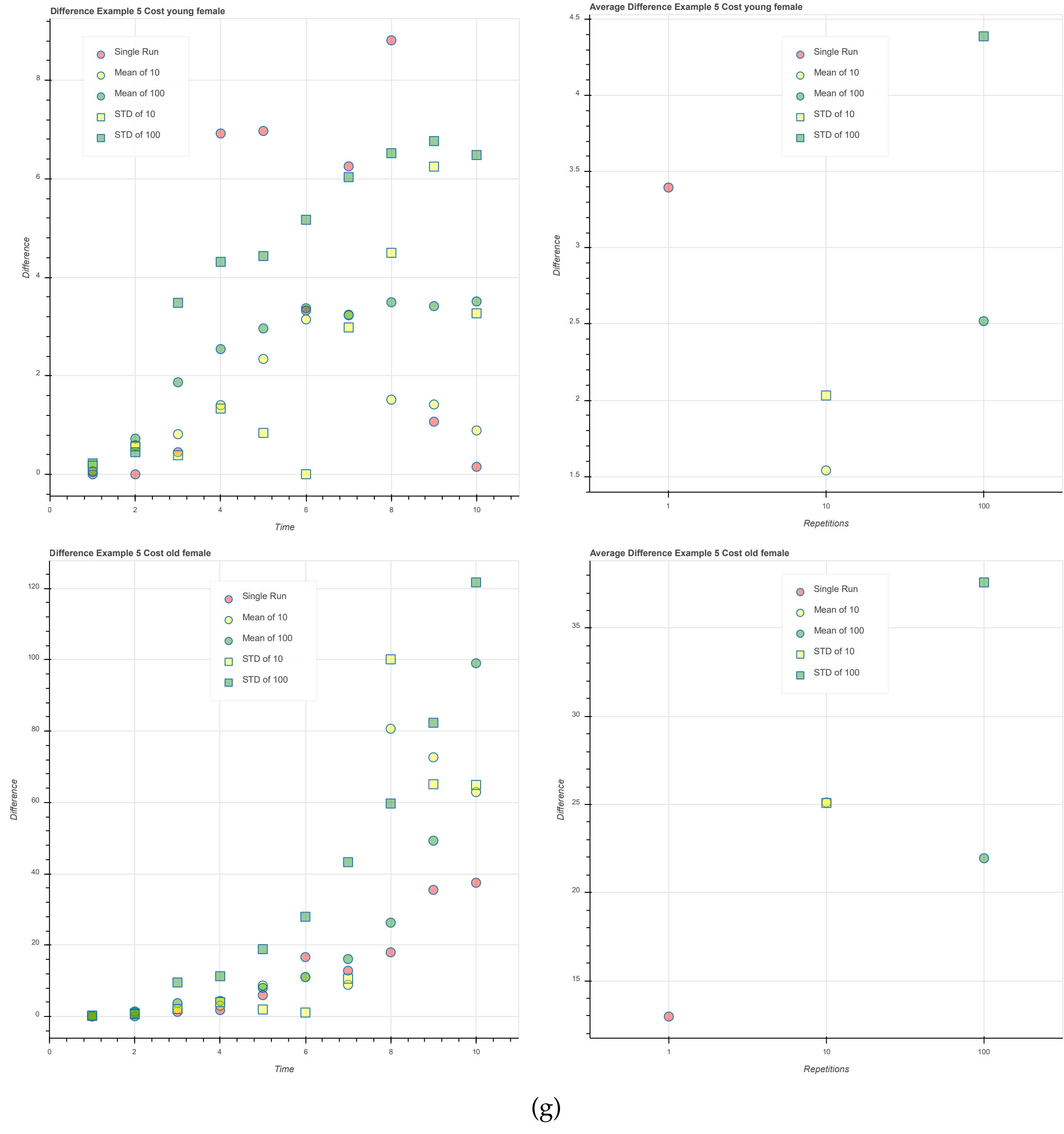
(e)

Figure 6.16: (continued) Statistical analysis for females in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



(f)

Figure 6.16: (continued) Statistical analysis for females in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.



(g)

Figure 6.16: (continued) Statistical analysis for females in example 5. (a) Healthy. (b) Sick. (c) Dead. (d) Age. (e) Blood Pressure. (f) Cost this year. (g) Cost.

CHAPTER 7

CONCLUSIONS

In the middle of a reproducibility crisis across many disciplines, community-driven standards in systems and synthetic biology are being developed to assist in the reproducibility of biological models and designs. One such effort is the SBML standard. While SBML has allowed the representation of many biological models, modifications to the standard are necessary to enable the representation of more complex models. This is particularly true for heterogeneous population models. However, as models become more complex, efficient simulation methods need to be developed that scale accordingly. This dissertation proposes extensions and methodologies for the creation of heterogeneous population models and hybrid models, and simulation methods that can simulate such models efficiently. This chapter concludes the dissertation by highlighting the main contributions of this research, which are summarized in Section 7.1, and future directions of this research, which are discussed in Section 7.2.

7.1 Summary

This dissertation presents efficient modeling and simulation methods for heterogeneous populations using the SBML standard. The SBML standard is widely adopted and it is very active within the multi-scale modeling community that tries to address different types of modeling that traverse scales, from cells to organs to populations. When modeling many types of systems in different scales, it is essential to have many modeling capabilities. SBML has 280 tools reported that support it as well as an established development process, specifications, and annual meetings. This makes it an established infrastructure for modeling transport mechanism. Therefore, enriching SBML to support microsimulation may make it an attractive candidate for adoption for modelers that need to support many modeling systems. Population models are often described using hierarchy, which is an abstraction used for the reasoning of complex designs by building larger designs from

smaller designs. SBML can represent hierarchical designs using the comp package. Even though SBML can represent hierarchical models, simulations tools typically flatten out the hierarchy and consequently lose important structural information in the model. This dissertation proposes a hierarchical simulation method based on Gillespie's SSA that avoids flattening. Results have shown that the proposed simulator improves runtime and memory usage. While the hierarchical simulator has to perform replacements and deletions on the fly, such overhead can be avoided with efficient data structures. Although SBML is powerful and allows the representation of complex models, the standard is not efficient for the representation of large designs, such as heterogeneous populations. Oftentimes, population models require a large number of copies of regular structures. For this reason, the arrays package has been proposed to address this issue. This dissertation demonstrates that the arrays package can be used to construct large lattice-based population models quite easily. In addition, this dissertation proposes an efficient simulation method that takes advantage of the arrays structure by preventing the unnecessary duplication of data caused by flattening. Results have shown that memory usage is improved significantly. However, the method adds some overhead when retrieving the value of a certain variables within an array due to the necessity of computing indices when accessing arrayed elements.

Another major contribution of this dissertation is towards reproducibility. In biology, different modeling formalisms best describe different biological models. Hence, it is important for a model to be able to combine different formalisms. This dissertation demonstrates how hybrid models can be described in SBML using hierarchy. This is demonstrated by using DFBA as an example. Such models have been created and successfully exchanged between two tools, where the hierarchical simulator provides the necessary infrastructure for simulating such models. In addition, this dissertation shows how SBML can contribute to the disease modeling field by modeling microsimulation disease models using the SBML arrays package. Once disease models are implemented in SBML, it opens a multitude of software tool options for disease modelers and may have considerable impact on the field (in particular, a significant impact on model reproducibility). Once model reproducibility is no longer an issue, model credibility will certainly increase.

7.2 Future Work

This section discusses future directions of the work presented in this dissertation.

7.2.1 Simulation of Population Dynamics

In electrical engineering, the size of the circuit is constant. Once an integrated circuit is fabricated, the number of transistors stays the same. Unlike electronic circuits, cellular populations are inherently dynamic. Namely, cells undergo dynamic processes such as cell division and death, and such events add or remove sub-models from the simulation dynamically. The proposed hierarchical simulator can be modified to support such changing model structures, and this requirement is actually a major motivation for the development of this simulator.

7.2.2 Explore Ways to Improve Performance

A future enhancement is to improve even further the efficiency of the simulation methods presented in this dissertation. One way to accomplish this is through dynamic model abstraction. In previous work, significant improvements in analysis time are achieved by removing unimportant details using automated model abstraction before simulation which improves simulation time while still delivering accurate results. A dynamic hierarchical simulator has the potential to allow these abstractions to be performed on-the-fly to manage complexity as needed to balance computational cost with accuracy. Finally, given that dynamic hierarchical models are inherently concurrent, parallel processing can also be explored to further improve simulation time.

7.2.3 Improve Arrays Support

The arrays simulator presented in this dissertation handles arrays on-the-fly. Handling arrays on-the-fly allows one to leverage sparse array data structures to further improve the memory-efficiency. Namely, the simulator can be extended to support sparse arrays. Furthermore, the simulator can potentially support the analysis of arrayed models with dynamically changing sizes, which cannot be accomplished when a model is flattened because flattening is that the model has to be statically computable.

7.2.4 Extend Hybrid Modeling Capabilities

Currently, the proposed approach supports the modeling of DFBA models based on the SOA simulation algorithm. Most DFBA models are stiff and small time steps are required for stability, making the SOA approach computationally expensive. Another disadvantage of the SOA approach is that it requires a sufficiently small fixed time step to give accurate results. Future directions include the exploration of adaptive time steps for executing the DFBA with SOA, alternative DFBA methods, such as DOA or DA, and extending our scheme to encode such models. In addition, only small to medium-size DFBA models have been encoded in our proposed approach. For future work, genome-scale metabolic models will need to be encoded. This would allow to evaluation of the scalability and performance of the proposed approach.

The hierarchical simulator has shown that it is capable of simulation models that couple different modeling formalisms. This can be further generalized and allow the simulation of models with different representations (e.g. MATLAB, scripting languages, CellML, and others).

7.2.5 Extend Disease Modeling Capabilities

The long-term goal of this effort of implementing disease modeling examples in SBML is to eventually allow converting `MIST` examples to SBML using the SBML Arrays package. The provided examples pave the way in this direction. Those examples do not cover all possible modeling elements used in epidemiological modeling, such as infectious disease modeling, discrete event simulation, or population generation. Only the very basic essential building block elements, that are regularly used to model chronic disease progression at the individual level, are presented here. Those examples are sufficient to support tasks such as life expectancy estimation and cost effectiveness analysis, which are core uses of disease models. Future work will include adding more elements such as handling event states, splitting and joining disease processes and other elements supported by MIST with the intention to promote SBML Arrays to be part of the SBML standard. In this work, the model transport is partially manual since MIST did not write the SBML code that was transported to iBioSim. Future work will address automation of this mechanism. This work is intended to establish feasibility of SBML Arrays as a model transport mechanism.

It is only a step towards adoption by SBML editors into the SBML standard, which already provides SBML Arrays specification and tools towards such a goal.

7.2.6 Improve SED-ML support

The simulators presented in this dissertation support SED-ML through `iBioSim`. While the tool supports many features of SED-ML, there are many features that are not supported. However, because the simulators presented in this dissertation can be executed as stand-alone applications, they are easier to implement the features of SED-ML that are not supported in `iBioSim`, such as repeated tasks and data generators.

7.2.7 Enriched Cellular Population Modeling

The benefit of supporting standards is that workflows can be created. For future work, this research can be extended to allow a workflow for the generation of population models of cell-cell communication models from sub-modules inferred by data. Namely, well-characterized sub-models represented in SBML can be placed in a population and simulation can be used to predict the behavior of such models.

REFERENCES

[1] A. Aderem, "Systems biology: its practice and challenges," *Cell*, vol. 121, no. 4, pp. 511–513, 2005.

[2] H. Kitano, "Computational systems biology," *Nature*, vol. 420, no. 6912, pp. 206–210, 2002.

[3] D. Endy, "Foundations for engineering biology," *Nature*, pp. 438–439, 2005.

[4] A. Arkin, "Setting the standard in synthetic biology," *Nature Biotechnology*, vol. 26, no. 7, pp. 771–774, jul 2008.

[5] J. B. Lucks and A. P. Arkin, "The hunt for the biological transistor," *IEEE Spectrum*, vol. 48, pp. 38–43, 2011.

[6] K. L. Kurita, E. Glassey, and R. G. Linington, "Integration of high-content screening and untargeted metabolomics for comprehensive functional annotation of natural product libraries," *Proceedings of the National Academy of Sciences*, vol. 112, no. 39, pp. 11 999–12 004, 2015. [Online]. Available: http://www.pnas.org/content/112/39/11999

[7] C. J. Paddon and J. D. Keasling, "Semi-synthetic artemisinin: a model for the use of synthetic biology in pharmaceutical development," *Nature Reviews Microbiology*, vol. 12, no. 5, pp. 355–367, apr 2014.

[8] N. S. Forbes, "Engineering the perfect (bacterial) cancer therapy," *Nature Reviews Cancer*, 2010.

[9] L. W. M. Loo, I. Cheng, M. Tiirikainen, A. Lum-Jones, A. Seifried, L. M. Dunklee, J. M. Church, R. Gryfe, D. J. Weisenberger, R. W. Haile, S. Gallinger, D. J. Duggan, S. N. Thibodeau, G. Casey, and L. Le Marchand, "cis-expression qtl analysis of established colorectal cancer risk variants in colon tumors and adjacent normal tissue," *PLOS ONE*, vol. 7, no. 2, pp. 1–10, 02 2012. [Online]. Available: https://doi.org/10.1371/journal.pone.0030477

[10] M. S. Antunes, K. J. Morey, J. J. Smith, K. D. Albrecht, T. A. Bowen, J. K. Zdunek, J. F. Troupe, M. J. Cuneo, C. T. Webb, H. W. Hellinga, and J. I. Medford, "Programmable ligand detection system in plants through a synthetic signal transduction pathway," *PLOS ONE*, vol. 6, no. 1, pp. 1–11, 01 2011. [Online]. Available: https://doi.org/10.1371/journal.pone.0016292

[11] F. Aquea, F. Federici, C. Moscoso, A. Vega, P. Jullian, J. Haseloff, and P. Arce-Johnson, "A molecular framework for the inhibition of arabidopsis root growth in response to boron toxicity," *Plant, Cell & Environment*, vol. 35, no. 4, pp. 719–734, nov 2011.

[12] D. F. Savage, J. Way, and P. A. Silver, "Defossiling fuel: How synthetic biology can transform biofuel production," *ACS Chemical Biology*, vol. 3, no. 1, pp. 13–16, 2008.

[13] J. A. N. Brophy and C. A. Voigt, "Principles of genetic circuit design," *Nature Methods*, vol. 11, no. 5, pp. 508–520, may 2014.

[14] C. J. Myers, *Engineering Genetic Circuits*. Chapman and Hall/CRC, 2009.

[15] D. Densmore, "Bio-design automation: Nobody said it would be easy," *ACS Synthetic Biology*, vol. 1, no. 8, pp. 296–296, 2012, pMID: 23651283. [Online]. Available: https://doi.org/10.1021/sb300062c

[16] A. A. K. Nielsen, B. S. Der, J. Shin, P. Vaidyanathan, V. Paralanov, E. A. Strychalski, D. Ross, D. Densmore, and C. A. Voigt, "Genetic circuit design automation," *Science*, vol. 352, no. 6281, 2016. [Online]. Available: http://science.sciencemag.org/content/352/6281/aac7341

[17] C. J. Myers, N. Barker, H. Kuwahara, K. Jones, C. Madsen, and N. P. D. Nguyen, "Genetic design automation," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, Nov. 2009.

[18] H. M. Salis, E. A. Mirsky, and C. A. Voigt, "Automated design of synthetic ribosome binding sites to control protein expression," *Nature Biotechnology*, vol. 27, no. 10, pp. 946–950, oct 2009.

[19] S. S. Jang, K. T. Oishi, R. G. Egbert, and E. Klavins, "Specification and simulation of synthetic multicelled behaviors," *ACS Synthetic Biology*, vol. 1, no. 8, pp. 365–374, jul 2012.

[20] L. Huynh, A. Tsoukalas, M. Koppe, and I. Tagkopoulos, "SBROME: A scalable optimization and module matching framework for automated biosystems design," *ACS Synthetic Biology*, vol. 2, no. 5, pp. 263–273, 2013.

[21] N. J. Hillson, R. D. Rosengarten, and J. D. Keasling, "j5 DNA assembly design automation software," *ACS Synthetic Biology*, vol. 1, no. 1, pp. 14–21, 2012.

[22] T. S. Ham, Z. Dmytriv, H. Plahar, J. Chen, N. J. Hillson, and J. D. Keasling, "Design, implementation and practice of JBEI-ICE: an open source biological part registry platform and tools," *Nucleic Acids Research*, vol. 40, no. 18, p. e141, 2012.

[23] D. Chandran, F. T. Bergmann, and H. M. Sauro, "TinkerCell: modular CAD tool for synthetic biology," *J. Biol. Eng.*, vol. 3, no. 19, 2009.

[24] J. R. Karr, J. C. Sanghvi, D. N. Macklin, M. V. Gutschow, J. M. Jacobs, J. Bolival, Benjamin, N. Assad-Garcia, J. I. Glass, and M. W. Covert, "A whole-cell computational model predicts phenotype from genotype," *Cell*, no. 2, pp. 389–401, 2015/04/15 2012.

[25] L. You, R. S. Cox, R. Weiss, and F. H. Arnold, "Programmed population control by cell–cell communication and regulated killing," *Nature*, vol. 428, no. 6985, pp. 868–871, apr 2004.

[26] D. Karig, K. M. Martini, T. Lu, N. A. DeLateur, N. Goldenfeld, and R. Weiss, "Stochastic turing patterns in a synthetic bacterial population," *Proceedings of the National Academy of Sciences*, vol. 115, no. 26, pp. 6572–6577, jun 2018.

[27] A. Phillips and L. Cardelli, “A correct abstract machine for the stochastic pi-calculus,” in *Concurrent Models in Molecular Biology*, August 2004. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-correct-abstract-machine-for-the-stochastic-pi-calculus/

[28] M. L. Blinov, J. R. Faeder, B. Goldstein, and W. S. Hlavacek, “BioNetGen: software for rule-based modeling of signal transduction based on the interactions of molecular domains,” *Bioinformatics*, vol. 20, no. 17, pp. 3289–3291, jun 2004.

[29] M. H. Swat, G. L. Thomas, J. M. Belmonte, A. Shirinifard, D. Hmeljak, and J. A. Glazier, “Multi-scale modeling of tissues using CompuCell3d,” in *Methods in Cell Biology*. Elsevier, 2012, pp. 325–366.

[30] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano *et al.*, “The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models.” *Bioinformatics*, vol. 19, no. 4, pp. 524–531, Mar. 2003. [Online]. Available: + http://dx.doi.org/10.1093/bioinformatics/btg015

[31] C. J. Myers, N. Barker, K. Jones, H. Kuwahara, C. Madsen, and N.-P. D. Nguyen, “ibiosim: a tool for the analysis and design of genetic circuits,” *Bioinformatics*, vol. 25, no. 21, pp. 2848–2849, 2009. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btp457

[32] C. Madsen, C. J. Myers, T. Patterson, N. Roehner, J. T. Stevens, and C. Winstead, “Design and test of genetic circuits using iBiosim,” *IEEE Design Test of Computers*, vol. 29, no. 3, pp. 32–39, June 2012.

[33] L. Watanabe, T. Nguyen, M. Zhang, Z. Zundel, Z. Zhang, C. Madsen, N. Roehner, and C. Myers, “iBioSim 3: A tool for model-based genetic circuit design,” *ACS Synthetic Biology*, jul 2018.

[34] S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer, “COPASI — a complex pathway simulator,” *Bioinformatics*, vol. 22, no. 24, pp. 3067–3074, 2006.

[35] H. M. Sauro, T. T. Karlsson, M. Swat, M. Galdzicki, and A. Somogyi, “libRoadRunner: a high performance SBML compliant simulator,” *bioRxiv*, 2013.

[36] A. Dörr, R. Keller, A. Zell, and A. Dräger, “Sbmlsimulator: A java tool for model simulation and parameter estimation in systems biology,” *Computation*, vol. 2, no. 4, pp. 246–257, 2014. [Online]. Available: http://www.mdpi.com/2079-3197/2/4/246

[37] R. Keller, A. Dörr, A. Tabira, A. Funahashi, M. J. Ziller, R. Adams, N. Rodriguez, N. L. Novère, N. Hiroi, H. Planatscher *et al.*, “The systems biology simulation core algorithm,” *BMC systems biology*, vol. 7, no. 1, p. 55, 2013.

[38] F. A. Kolpakov, “BioUML-framework for visual modeling and simulation biological systems,” in *Proceedings of the International Conference on Bioinformatics of Genome Regulation and Structure*, 2002, pp. 130–133.

[39] M. Baker, “1,500 scientists lift the lid on reproducibility,” *Nature*, vol. 533, no. 7604, pp. 452–454, may 2016.

[40] J. Vitek and T. Kalibera, “Repeatability, reproducibility, and rigor in systems research,” in *Proceedings of the ninth ACM international conference on Embedded software - EMSOFT́11*. ACM Press, 2011.

[41] J. Ioannidis and C. Doucouliagos, “Whats to know about the credibility of empirical economics?” *Journal of Economic Surveys*, pp. n/a–n/a, apr 2013.

[42] O. S. Collaboration, “Estimating the reproducibility of psychological science,” *Science*, vol. 349, no. 6251, pp. aac4716–aac4716, aug 2015.

[43] S. N. Goodman, D. Fanelli, and J. P. A. Ioannidis, “What does research reproducibility mean?” *Science Translational Medicine*, vol. 8, no. 341, pp. 341ps12–341ps12, jun 2016.

[44] B. Mons, “Which gene did you mean?” *BMC Bioinformatics*, vol. 6, no. 1, p. 142, 2005.

[45] D. Garijo, S. Kinnings, L. Xie, L. Xie, Y. Zhang, P. E. Bourne, and Y. Gil, “Quantifying reproducibility in computational biology: The case of the tuberculosis drugome,” *PLoS ONE*, vol. 8, no. 11, p. e80278, nov 2013.

[46] D. Waltemath and O. Wolkenhauer, “How modeling standards, software, and initiatives support reproducibility in systems biology and systems medicine,” *IEEE Transactions on Biomedical Engineering*, vol. 63, no. 10, pp. 1999–2006, oct 2016.

[47] J. P. A. Ioannidis, “Why most published research findings are false,” *PLoS Medicine*, vol. 2, no. 8, p. e124, aug 2005.

[48] V. Stodden, P. Guo, and Z. Ma, “Toward reproducible computational research: An empirical analysis of data and code policy adoption by journals,” *PLoS ONE*, vol. 8, no. 6, p. e67111, jun 2013.

[49] M. Galdzicki, K. P. Clancy, E. Oberortner, M. Pocock, J. Y. Quinn, C. A. Rodriguez, N. Roehner, M. L. Wilson, L. Adam, J. C. Anderson, B. A. Bartley, J. Beal, D. Chandran, J. Chen, D. Densmore, D. Endy, R. Grunberg, J. Hallinan, N. J. Hillson, J. D. Johnson, A. Kuchinsky, M. Lux, G. Misirli, J. Peccoud, H. A. Plahar, E. Sirin, G.-B. Stan, A. Villalobos, A. Wipat, J. H. Gennari, C. J. Myers, and H. M. Sauro, “The synthetic biology open language (sbol) provides a community standard for communicating designs in synthetic biology,” *Nat Biotech*, vol. 32, no. 6, pp. 545–550, Jun 2014, computational Biology. [Online]. Available: http://dx.doi.org/10.1038/nbt.2891

[50] D. Waltemath, R. Adams, F. T. Bergmann, M. Hucka, F. Kolpakov, A. K. Miller, I. I. Moraru, D. Nickerson, S. Sahle, J. L. Snoep, and N. Le Novère, “Reproducible computational biology experiments with SED-ML-the Simulation Experiment Description Markup Language,” *BMC systems biology*, vol. 5, no. 1, p. 198, 2011.

[51] M. Swat, S. Moodie, S. Wimalaratne, N. Kristensen, M. Lavielle, A. Mari, P. Magni, M. Smith, R. Bizzotto, L. Pasotti, E. Mezzalana, E. Comets, C. Sarr, N. Terranova, E. Blaudez, P. Chan, J. Chard, K. Chatel, M. Chenel, D. Edwards, C. Franklin, T. Giorgino, M. Glont, P. Girard, P. Grenon, K. Harling, A. Hooker, R. Kaye, R. Keizer, C. Kloft, J. Kok, N. Kokash, C. Laibe, C. Laveille, G. Lestini, F. Mentré, A. Munafo, R. Nordgren, H. Nyberg, Z. Parra-Guillen, E. Plan, B. Ribba, G. Smith, I. Trocóniz,

F. Yvon, P. Milligan, L. Harnisch, M. Karlsson, H. Hermjakob, and N. L. Novère, “Pharmacometrics markup language (PharmML): Opening new perspectives for model exchange in drug development,” *CPT: Pharmacometrics & Systems Pharmacology*, vol. 4, no. 6, pp. 316–319, jun 2015.

[52] D. J. Wilkinson, “Stochastic modelling for quantitative description of heterogeneous biological systems,” *Nature Reviews Genetics*, vol. 10, no. 2, pp. 122–133, feb 2009.

[53] N. Le Novère, “Quantitative and logic modelling of molecular and gene networks,” *Nature Reviews Genetics*, 2015.

[54] N. Friedman, M. Linial, I. Nachman, and D. Pe'er, “Using bayesian networks to analyze expression data,” *Journal of Computational Biology*, vol. 7, no. 3-4, pp. 601–620, aug 2000.

[55] M. Stephens and D. J. Balding, “Bayesian statistical methods for genetic association studies,” *Nature Reviews Genetics*, vol. 10, no. 10, pp. 681–690, oct 2009.

[56] N. E. Lewis, H. Nagarajan, and B. O. Palsson, “Constraining the metabolic genotype–phenotype relationship using a phylogeny of in silico methods,” *Nature Reviews Microbiology*, vol. 10, no. 4, pp. 291–305, feb 2012.

[57] D. T. Gillespie, “Exact stochastic simulation of coupled chemical reactions,” *Journal of Physical Chemistry*, vol. 81, no. 25, pp. 2340–2361, 1977.

[58] M. Rathinam, L. R. Petzold, Y. Cao, and D. T. Gillespie, “Stiffness in stochastic chemically reacting systems: The implicit tau-leaping method,” *The Journal of Chemical Physics*, vol. 119, no. 24, pp. 12 784–12 794, 2003. [Online]. Available: https://doi.org/10.1063/1.1627296

[59] D. T. Gillespie and L. R. Petzold, “Improved leap-size selection for accelerated stochastic simulation,” *The Journal of Chemical Physics*, vol. 119, no. 16, pp. 8229–8234, 2003. [Online]. Available: https://doi.org/10.1063/1.1613254

[60] A. Slepoy, A. P. Thompson, and S. J. Plimpton, “A constant-time kinetic monte carlo algorithm for simulation of large biochemical reaction networks,” *The Journal of Chemical Physics*, vol. 128, no. 20, p. 205101, may 2008.

[61] M. A. Gibson and J. Bruck, “Efficient exact stochastic simulation of chemical systems with many species and many channels,” *The Journal of Physical Chemistry A*, vol. 104, no. 9, pp. 1876–1889, mar 2000.

[62] H. Kuwahara, C. J. Myers, M. S. Samoilov, N. A. Barker, and A. P. Arkin, “Automated abstraction methodology for genetic regulatory networks,” in *Transactions on Computational Systems Biology VI*, C. Priami and G. Plotkin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 150–175.

[63] L. P. Smith, M. Hucka, S. Hoops, A. Finney, M. Ginkel, C. J. Myers, I. Moraru, and W. Liebermeister, “SBML level 3 package: Hierarchical model composition, version 1 release 3,” *Journal of Integrative Bioinformatics*, vol. 12, no. 2, jun 2015.

[64] T. S. Gardner, C. R. Cantor, and J. J. Collins, "Construction of a genetic toggle switch in escherichia coli," *Nature*, vol. 403, no. 6767, pp. 339–342, 01 2000. [Online]. Available: http://dx.doi.org/10.1038/35002131

[65] M. Elowitz and S. Leibler, "A synthetic oscillatory network of transcriptional regulators," *Nature*, vol. 403, no. 6767, pp. 335–338, 2000.

[66] W. C. Ruder, T. Lu, and J. Collins, "Synthetic biology moving into the clinic," *Science*, vol. 333, pp. 1248–1252, 2011.

[67] W. Weber and M. Fussenegger, "Emerging biomedical applications of synthetic biology," *Nature Reviews Genetics*, vol. 13, no. 1, pp. 21–35, 2012.

[68] D. Waltemath, J. R. Karr, F. T. Bergmann, V. Chelliah, M. Hucka, M. Krantz, W. Liebermeister, P. Mendes, C. J. Myers, P. Pir, B. Alaybeyoglu, N. K. Aranganathan, K. Baghalian, A. T. Bittig, P. E. P. Burke, M. Cantarelli, Y. H. Chew, R. S. Costa, J. Cursons, T. Czauderna, A. P. Goldberg, H. F. Gomez, J. Hahn, T. Hameri, D. F. H. Gardiol, D. Kazakiewicz, I. Kiselev, V. Knight-Schrijver, C. Knupfer, M. Konig, D. Lee, A. Lloret-Villas, N. Mandrik, J. K. Medley, B. Moreau, H. Naderi-Meshkin, S. K. Palaniappan, D. Priego-Espinosa, M. Scharm, M. Sharma, K. Smallbone, N. J. Stanford, J.-H. Song, T. Theile, M. Tokic, N. Tomar, V. Toure, J. Uhlendorf, T. M. Varusai, L. H. Watanabe, F. Wendland, M. Wolfien, J. T. Yurkovich, Y. Zhu, A. Zardilis, A. Zhukova, and F. Schreiber, "Toward community standards and software for whole-cell modeling," *IEEE Transactions on Biomedical Engineering*, vol. 63, no. 10, pp. 2007–2014, oct 2016.

[69] A. Bordbar, J. M. Monk, Z. A. King, and B. O. Palsson, "Constraint-based models predict metabolic and associated cellular functions," *Nature Reviews Genetics*, vol. 15, no. 2, pp. 107–120, jan 2014.

[70] J. D. Orth, I. Thiele, and B. Ø. Palsson, "What is flux balance analysis?" *Nature Biotechnology*, vol. 28, no. 3, pp. 245–248, mar 2010.

[71] M. Hucka, D. P. Nickerson, G. D. Bader, F. T. Bergmann, J. Cooper, E. Demir, A. Garny, M. Golebiewski, C. J. Myers, F. Schreiber, D. Waltemath, and N. Le Novère, "Promoting coordinated development of community-based information standards for modeling in biology: The combine initiative," *Frontiers in Bioengineering and Biotechnology*, vol. 3, p. 19, 2015. [Online]. Available: https://www.frontiersin.org/article/10.3389/fbioe.2015.00019

[72] M. Hucka, F. T. Bergmann, A. Dräger, S. Hoops, S. M. Keating, N. L. Novère, C. J. Myers, B. G. Olivier, S. Sahle, J. C. Schaff, L. P. Smith, D. Waltemath, and D. J. Wilkinson, "The systems biology markup language (SBML): Language specification for level 3 version 2 core," *Journal of Integrative Bioinformatics*, vol. 15, no. 1, mar 2018.

[73] B. G. Olivier and F. T. Bergmann, "The systems biology markup language (SBML) level 3 package: Flux balance constraints," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, jun 2015.

[74] M. Hucka and L. P. Smith, "SBML level 3 package: Groups, version 1 release 1," *Journal of Integrative Bioinformatics*, vol. 13, no. 3, sep 2016.

[75] R. Gauges, U. Rost, S. Sahle, K. Wengler, and F. T. Bergmann, "The systems biology markup language (SBML) level 3 package: Layout, version 1 core," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, jun 2015.

[76] F. Zhang and M. Meier-Schellersheim, "SBML level 3 package: Multistate, multicomponent and multicompartment species, version 1, release 1," *Journal of Integrative Bioinformatics*, vol. 15, no. 1, apr 2018.

[77] C. Chaouiya, S. M. Keating, D. Berenguier, A. Naldi, D. Thieffry, M. P. van Iersel, N. L. Novère, and T. Helikar, "The systems biology markup language (SBML) level 3 package: Qualitative models, version 1, release 1," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, p. 270, 2015.

[78] F. T. Bergmann, S. M. Keating, R. Gauges, S. Sahle, and K. Wengler, "SBML level 3 package: Render, version 1, release 1," *Journal of Integrative Bioinformatics*, vol. 15, no. 1, apr 2018.

[79] B. J. Bornstein, S. M. Keating, A. Jouraku, and M. Hucka, "LibSBML: an API library for SBML," *Bioinformatics*, vol. 24, no. 6, pp. 880–881, 2008. [Online]. Available: + http://dx.doi.org/10.1093/bioinformatics/btn051

[80] N. Rodriguez, A. Thomas, L. Watanabe, I. Y. Vazirabad, V. Kofia, H. F. Gómez, F. Mittag, J. Matthes, J. Rudolph, F. Wrzodek, E. Netz, A. Diamantikos, J. Eichner, R. Keller, C. Wrzodek, S. Fröhlich, N. E. Lewis, C. J. Myers, N. Le Novère, B. Ø. Palsson, M. Hucka, and A. Dräger, "JSBML 1.0: providing a smorgasbord of options to encode systems biology models," *Bioinformatics*, vol. 31, no. 20, pp. 3383–3386, 2015. [Online]. Available: + http://dx.doi.org/10.1093/bioinformatics/btv341

[81] N. Roehner, J. Beal, K. Clancy, B. Bartley, G. Misirli, R. Grünberg, E. Oberortner, M. Pocock, M. Bissell, C. Madsen, T. Nguyen, M. Zhang, Z. Zhang, Z. Zundel, D. Densmore, J. H. Gennari, A. Wipat, H. M. Sauro, and C. J. Myers, "Sharing structure and function in biological design with SBOL 2.0," *ACS Synthetic Biology*, vol. 5, no. 6, pp. 498–506, 2016.

[82] N. L. Novère, M. Hucka, H. Mi, S. Moodie, F. Schreiber, A. Sorokin, E. Demir, K. Wegner, M. I. Aladjem, S. M. Wimalaratne, F. T. Bergman, R. Gauges, P. Ghazal, H. Kawaji, L. Li, Y. Matsuoka, A. Villéger, S. E. Boyd, L. Calzone, M. Courtot, U. Dogrusoz, T. C. Freeman, A. Funahashi, S. Ghosh, A. Jouraku, S. Kim, F. Kolpakov, A. Luna, S. Sahle, E. Schmidt, S. Watterson, G. Wu, I. Goryanin, D. B. Kell, C. Sander, H. Sauro, J. L. Snoep, K. Kohn, and H. Kitano, "The systems biology graphical notation," *Nature Biotechnology*, vol. 27, no. 8, pp. 735–741, aug 2009.

[83] F. T. Bergmann, R. Adams, S. Moodie, J. Cooper, M. Glont, M. Golebiewski, M. Hucka, C. Laibe, A. K. Miller, D. P. Nickerson, B. G. Olivier, N. Rodriguez, H. M. Sauro, M. Scharm, S. Soiland-Reyes, D. Waltemath, F. Yvon, and N. Le Novère, "Combine archive and omex format: one file to share all information to reproduce a modeling project," *BMC Bioinformatics*, vol. 15, no. 1, p. 369, Dec 2014. [Online]. Available: https://doi.org/10.1186/s12859-014-0369-z

[84] N. A. Barker, C. J. Myers, and H. Kuwahara, "Learning genetic regulatory network connectivity from time series data," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 8, no. 1, pp. 152–165, Jan 2011.

[85] N.-P. D. Nguyen, H. Kuwahara, C. J. Myers, and J. P. Keener, "The design of a genetic muller c-element," in *13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'07)*, March 2007, pp. 95–104.

[86] C. Winstead, C. Madsen, and C. Myers, "iSSA: An incremental stochastic simulation algorithm for genetic circuits," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. IEEE, 2010, pp. 553–556.

[87] C. Madsen, Z. Zhang, N. Roehner, C. Winstead, and C. Myers, "Stochastic model checking of genetic circuits," *J. Emerg. Technol. Comput. Syst.*, vol. 11, no. 3, pp. 23:1–23:21, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2644817

[88] J. T. Stevens and C. J. Myers, "Dynamic modeling of cellular populations within iBiosim," *ACS Synthetic Biology*, vol. 2, no. 5, pp. 223–229, 2013.

[89] M. Zhang, J. A. McLaughlin, A. Wipat, and C. J. Myers, "SBOLDesigner 2: An intuitive tool for structural genetic design," *ACS Synthetic Biology*, 2017.

[90] J. A. McLaughlin, C. J. Myers, Z. Zundel, G. Mısırlı, M. Zhang, I. D. Ofiteru, A. Goñi-Moreno, and A. Wipat, "SynBioHub: A standards-enabled design repository for synthetic biology," *ACS Synthetic Biology*, vol. 7, no. 2, pp. 682–688, jan 2018.

[91] R. S. Cox, C. Madsen, J. A. McLaughlin, T. Nguyen, N. Roehner, B. Bartley, J. Beal, M. Bissell, K. Choi, K. Clancy, R. Grünberg, C. Macklin, G. Misirli, E. Oberortner, M. Pocock, M. Samineni, M. Zhang, Z. Zhang, Z. Zundel, J. H. Gennari, C. Myers, H. Sauro, and A. Wipat, "Synthetic biology open language (SBOL) version 2.2.0," *Journal of Integrative Bioinformatics*, vol. 15, no. 1, apr 2018.

[92] G. Mısırlı, A. Wipat, J. Mullen, K. James, M. Pocock, W. Smith, N. Allenby, and F. S. Hallinan, "Bacillondex: An integrated data resource for systems and synthetic biology," *Journal of Integrative Bioinformatics (JIB)*, vol. 10, no. 2, pp. 103–116, 2013.

[93] G. Mısırlı, J. Hallinan, M. Pocock, P. Lord, J. A. McLaughlin, H. Sauro, and A. Wipat, "Data integration and mining for synthetic biology design," *ACS synthetic biology*, vol. 5, no. 10, pp. 1086–1097, 2016.

[94] G. Mısırlı, T. Nguyen, J. A. McLaughlin, P. Vaidyanathan, T. S. Jones, D. Densmore, C. Myers, and A. Wipat, "A computational workflow for the automated generation of models of genetic designs," *ACS Synthetic Biology*, jun 2018.

[95] N. Roehner, Z. Zhang, T. Nguyen, and C. J.Myers, "Generating Systems Biology Markup Language Models from the Synthetic Biology Open Language," *ACS Synthetic Biology*, vol. 4, no. 8, pp. 873–879, 2015.

[96] T. Nguyen, N. Roehner, Z. Zundel, and C. J. Myers, "A converter from the Systems Biology Markup Language to the Synthetic Biology Open Language," *ACS Synthetic Biology*, vol. 5, no. 6, pp. 479–486, 2016.

[97] N. Roehner and C. J. Myers, "Directed acyclic graph-based technology mapping of genetic circuit models," *ACS Synth. Biol.*, vol. 3, pp. 543–555, 2014.

[98] S. Hennig, G. Rödel, and K. Ostermann, "Artificial cell-cell communication as an emerging tool in synthetic biology applications," *Journal of Biological Engineering*, vol. 9, no. 1, aug 2015.

[99] S. Toda, L. R. Blauch, S. K. Y. Tang, L. Morsut, and W. A. Lim, "Programming self-organizing multicellular structures with synthetic cell-cell signaling," *Science*, p. eaat0271, may 2018.

[100] K. Thurley, L. F. Wu, and S. J. Altschuler, "Modeling cell-to-cell communication networks using response-time distributions," *Cell Systems*, vol. 6, no. 3, pp. 355–367.e5, mar 2018.

[101] M. Hucka, L. Smith, F. Bergmann, and S. M. Keating, "Sbml test suite release 3.3.0," 2017.

[102] L. Watanabe and C. J. Myers, "Efficient analysis of systems biology markup language models of cellular populations using arrays," *ACS Synthetic Biology*, vol. 5, no. 8, pp. 835–841, mar 2016.

[103] E. V. Nikolaev and E. D. Sontag, "Quorum-sensing synchronization of synthetic toggle switches: A design based on monotone dynamical systems theory," *PLOS Computational Biology*, vol. 12, no. 4, p. e1004881, apr 2016.

[104] M. B. Miller and B. L. Bassler, "Quorum sensing in bacteria," *Annual Review of Microbiology*, vol. 55, no. 1, pp. 165–199, oct 2001.

[105] A. A. Cuellar, C. M. Lloyd, P. F. Nielsen, D. P. Bullivant, D. P. Nickerson, and P. J. Hunter, "An overview of CellML 1.1, a biological model description language," *SIMULATION*, vol. 79, no. 12, pp. 740–747, dec 2003.

[106] A. K. Miller, J. Marsh, A. Reeve, A. Garny, R. Britten, M. Halstead, J. Cooper, D. P. Nickerson, and P. F. Nielsen, "An overview of the CellML API and its implementation," *BMC Bioinformatics*, vol. 11, no. 1, p. 178, 2010.

[107] R. Thomas, "Boolean formalization of genetic control circuits," *Journal of Theoretical Biology*, vol. 42, no. 3, pp. 563–585, dec 1973.

[108] S. Kauffman, "Metabolic stability and epigenesis in randomly constructed genetic nets," *Journal of Theoretical Biology*, vol. 22, no. 3, pp. 437–467, mar 1969.

[109] M. K. Morris, J. Saez-Rodriguez, P. K. Sorger, and D. A. Lauffenburger, "Logic-based models for the analysis of cell signaling networks," *Biochemistry*, vol. 49, no. 15, pp. 3216–3224, apr 2010.

[110] A. Varma and B. O. Palsson, "Stoichiometric flux balance models quantitatively predict growth and metabolic by-product secretion in wild-type Escherichia coli W3110." *Applied and environmental microbiology*, vol. 60, no. 10, pp. 3724–3731, 1994.

[111] A. Varma, B. W. Boesch, and B. O. Palsson, "Biochemical production capabilities ofescherichia coli," *Biotechnology and Bioengineering*, vol. 42, no. 1, pp. 59–73, jun 1993.

[112] J. M. Savinell and B. O. Palsson, "Network analysis of intermediary metabolism using linear optimization. i. development of mathematical formalism," *Journal of Theoretical Biology*, vol. 154, no. 4, pp. 421–454, feb 1992.

[113] R. Mahadevan, J. S. Edwards, and F. J. Doyle, "Dynamic flux balance analysis of diauxic growth in escherichia coli," *Biophysical Journal*, vol. 83, no. 3, pp. 1331–1340, sep 2002.

[114] R.-Y. Luo, S. Liao, G.-Y. Tao, Y.-Y. Li, S. Zeng, Y.-X. Li, and Q. Luo, "Dynamic analysis of optimality in myocardial energy metabolism under normal and ischemic conditions," *Molecular Systems Biology*, vol. 2, jun 2006.

[115] F. Pizarro, C. Varela, C. Martabit, C. Bruno, J. R. Pérez-Correa, and E. Agosin, "Coupling kinetic expressions and metabolic networks for predicting wine fermentations," *Biotechnology and Bioengineering*, vol. 98, no. 5, pp. 986–998, 2007.

[116] G. Lequeux, J. Beauprez, J. Maertens, E. V. Horen, W. Soetaert, E. Vandamme, and P. A. Vanrolleghem, "Dynamic metabolic flux analysis demonstrated on cultures where the limiting substrate is changed from carbon to nitrogen andVice versa," *Journal of Biomedicine and Biotechnology*, vol. 2010, pp. 1–19, 2010.

[117] A. L. Meadows, R. Karnik, H. Lam, S. Forestell, and B. Snedecor, "Application of dynamic flux balance analysis to an industrial escherichia coli fermentation," *Metabolic Engineering*, vol. 12, no. 2, pp. 150–160, mar 2010.

[118] T. J. Hanly and M. A. Henson, "Dynamic flux balance modeling of microbial cocultures for efficient batch fermentation of glucose and xylose mixtures," *Biotechnology and Bioengineering*, vol. 108, no. 2, pp. 376–385, oct 2010.

[119] J. L. Hjersted, M. A. Henson, and R. Mahadevan, "Genome-scale analysis ofSaccharomyces cerevisiae metabolism and ethanol production in fed-batch culture," *Biotechnology and Bioengineering*, vol. 97, no. 5, pp. 1190–1204, 2007.

[120] K. Höffner, S. M. Harwood, and P. I. Barton, "A reliable simulator for dynamic flux balance analysis," *Biotechnology and Bioengineering*, vol. 110, no. 3, pp. 792–802, oct 2012.

[121] J. A. Gomez, K. Höffner, and P. I. Barton, "DFBAlab: a fast and reliable MATLAB code for dynamic flux balance analysis," *BMC Bioinformatics*, vol. 15, no. 1, dec 2014.

[122] M. König, "matthiaskoenig/sbmlutils: sbmlutils-v0.1.8 10.5281/zenodo.1045519," Nov. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.1045519

[123] L. H. Watanabe and C. J. Myers, "Hierarchical stochastic simulation algorithm for SBML models of genetic circuits," *Frontiers in Bioengineering and Biotechnology*, vol. 2, nov 2014.

[124] E. T. Somogyi, J.-M. Bouteiller, J. A. Glazier, M. König, J. K. Medley, M. H. Swat, and H. M. Sauro, "libRoadRunner: a high performance SBML simulation and analysis library: Table 1." *Bioinformatics*, vol. 31, no. 20, pp. 3315–3321, jun 2015.

[125] A. Ebrahim, J. A. Lerman, B. O. Palsson, and D. R. Hyduke, "COBRApy: COnstraints-based reconstruction and analysis for python," *BMC Systems Biology*, vol. 7, no. 1, p. 74, 2013.

[126] A. Zhukova, A. Zhukova, D. Waltemath, N. Juty, C. Laibe, and N. L. Novère, "Kinetic simulation algorithm ontology," *Nature Precedings*, sep 2011.

[127] R. Mahadevan and C. Schilling, "The effects of alternate optimal solutions in constraint-based genome-scale metabolic models," *Metabolic Engineering*, vol. 5, no. 4, pp. 264–276, oct 2003.

[128] M. Courtot, N. Juty, C. Knupfer, D. Waltemath, A. Zhukova, A. Drager, M. Dumontier, A. Finney, M. Golebiewski, J. Hastings, S. Hoops, S. Keating, D. B. Kell, S. Kerrien, J. Lawson, A. Lister, J. Lu, R. Machne, P. Mendes, M. Pocock, N. Rodriguez, A. Villeger, D. J. Wilkinson, S. Wimalaratne, C. Laibe, M. Hucka, and N. L. Novere, "Controlled vocabularies and semantics in systems biology," *Molecular Systems Biology*, vol. 7, no. 1, pp. 543–543, apr 2014.

[129] M. Konig, A. Drager, and H.-G. Holzhutter, "CySBML: a cytoscape plugin for SBML," *Bioinformatics*, vol. 28, no. 18, pp. 2402–2403, jul 2012.

[130] J. D. Orth, B. Ø. Palsson, and R. M. T. Fleming, "Reconstruction and use of microbial metabolic networks: the core escherichia coli metabolic model as an educational guide," *EcoSal Plus*, vol. 4, no. 1, sep 2010.

[131] Z. A. King, J. Lu, A. Dräger, P. Miller, S. Federowicz, J. A. Lerman, A. Ebrahim, B. O. Palsson, and N. E. Lewis, "BiGG models: A platform for integrating, standardizing and sharing genome-scale models," *Nucleic Acids Research*, vol. 44, no. D1, pp. D515–D522, oct 2015.

[132] A. J. Hayes, J. Leal, A. M. Gray, R. R. Holman, and P. M. Clarke, "UKPDS outcomes model 2: a new version of a model to simulate lifetime health outcomes of patients with type 2 diabetes mellitus using data from the 30 year united kingdom prospective diabetes study: UKPDS 82," *Diabetologia*, vol. 56, no. 9, pp. 1925–1933, jun 2013.

[133] R. B. D'Agostino, R. S. Vasan, M. J. Pencina, P. A. Wolf, M. Cobain, J. M. Massaro, and W. B. Kannel, "General cardiovascular risk profile for use in primary care: The framingham heart study," *Circulation*, vol. 117, no. 6, pp. 743–753, jan 2008.

[134] D. Salem and R. Smith, "A mathematical model of ebola virus disease: Using sensitivity analysis to determine effective intervention targets," in *Proceedings of the Summer Computer Simulation Conference*, ser. SCSC '16. San Diego, CA, USA: Society for Computer Simulation International, 2016, pp. 3:1–3:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=3015574.3015577

[135] A. Lasry, G. S. Zaric, and M. W. Carter, "Multi-level resource allocation for hiv prevention: A model for developing countries," *European Journal of Operational Research*, vol. 180, no. 2, pp. 786 – 799, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221706002578

[136] H. S. Leff, M. Dada, and S. C. Graves, "An lp planning model for a mental health community support system," *Management Science*, vol. 32, no. 2, pp. 139–155, 1986. [Online]. Available: https://doi.org/10.1287/mnsc.32.2.139

[137] S. Scholz, M. Batram, and W. Greiner, "The silas model: Sexual infections as large-scale agent-based simulation," in *Proceedings of the Conference on Summer Computer Simulation*, ser. SummerSim '15. San Diego, CA, USA: Society for Computer Simulation International, 2015, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2874916.2874970

[138] J. Hippisley-Cox, C. Coupland, Y. Vinogradova, J. Robson, M. May, and P. Brindle, "Derivation and validation of qrisk, a new cardiovascular disease risk score for the united kingdom: prospective open cohort study," *BMJ*, vol. 335, no. 7611, p. 136, 2007. [Online]. Available: http://www.bmj.com/content/335/7611/136

[139] J. Hippisley-Cox, C. Coupland, Y. Vinogradova, J. Robson, R. Minhas, A. Sheikh, and P. Brindle, "Predicting cardiovascular risk in england and wales: prospective derivation and validation of qrisk2," *BMJ*, vol. 336, no. 7659, pp. 1475–1482, 2008. [Online]. Available: http://www.bmj.com/content/336/7659/1475

[140] D. J. Isaman, J. Barhak, and W. Ye, "Indirect estimation of a discrete-state discrete-time model using secondary data analysis of regression data," *Statistics in medicine*, vol. 28, no. 16, pp. 2095–2115, 2009.

[141] J. Barhak, D. J. Isaman, W. Ye, and D. Lee, "Chronic disease modeling and simulation software," *Journal of biomedical informatics*, vol. 43, no. 5, pp. 791–799, 2010.

[142] M. K. Smith, S. L. Moodie, R. Bizzotto, E. Blaudez, E. Borella, L. Carrara, P. Chan, M. Chenel, E. Comets, R. Gieschke, K. Harling, L. Harnisch, N. Hartung, A. C. Hooker, M. O. Karlsson, R. Kaye, C. Kloft, N. Kokash, M. Lavielle, G. Lestini, P. Magni, A. Mari, F. Mentré, C. Muselle, R. Nordgren, H. B. Nyberg, Z. P. Parra-Guillèn, L. Pasotti, N. Rode-Kristensen, M. L. Sardu, G. R. Smith, M. J. Swat, N. Terranova, G. Yngman, F. Yvon, N. Holford, and on behalf of the DDMoRe consortium, "Model description language (mdl): A standard for modeling and simulation," *CPT: Pharmacometrics and Systems Pharmacology*, vol. 6, no. 10, pp. 647–650, 2017. [Online]. Available: http://dx.doi.org/10.1002/psp4.12222

[143] M. I. Davidich and S. Bornholdt, "Boolean network model predicts cell cycle sequence of fission yeast," *PloS one*, vol. 3, no. 2, p. e1672, 2008.

[144] P. J. Goss and J. Peccoud, "Quantitative modeling of stochastic systems in molecular biology by using stochastic petri nets," *Proceedings of the National Academy of Sciences*, vol. 95, no. 12, pp. 6750–6755, 1998.

[145] L. Smith, M. J. Swat, and J. Barhak, "Sharing formats for disease models," in *Proceedings of the Summer Computer Simulation Conference*. Society for Computer Simulation International, 2016, p. 5.

[146] J. Barhak, "MIcro Simulation Tool to support disease modeling - MIST," https://simtk.org/projects/mist/, 2018, accessed: 2018-08-28.

[147] L. Watanabe, J. Barhak, and C. Myers, "Toward reproducible disease models using the systems biology markup language," *SIMULATION*, p. 003754971879321, sep 2018.

[148] L. Belloli, G. Wainer, and R. Najmanovich, "Parsing and model generation for biological processes," in *Proceedings of the Symposium on Theory of Modeling & Simulation*, ser. TMS-DEVS '16. San Diego, CA, USA: Society for Computer Simulation International, 2016, pp. 21:1–21:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2975389.2975410