Title

Approximate Simulations of Dynamical Graph Grammars using the Dynamical Graph Grammar Modeling Library

Author
Medwedeff, Eric

Publication Date
2024

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, IRVINE
and
SAN DIEGO STATE UNIVERSITY, SAN DIEGO

Approximate Simulations of Dynamical Graph Grammars using the Dynamical Graph Grammar Modeling Library

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

JOINT DOCTOR OF PHILOSOPHY

in Computational Science

by

Eric Medwedeff

Dissertation Committee:
Professor Eric Mjolsness, Chair
Professor Jose Castillo
Professor Peter Blomgren
Professor Jun Allard
Professor Charless Fowlkes

July 2024

DEDICATION

I'd like to dedicate this thesis to my mother Merry who, last year, left us way too soon. I'll always remember her as someone who loved to talk and tell stories for days. Not short stories, mind you, but long ones. The thing about her stories that I don't think most people realized, though, was that they always had point - it just took a while to get to that point. Without her knowing, I think she prepared me for writing a thesis. So, thank you mom. Thank you for everything. Thank you for bringing me into this world, and thank you for every day we shared. I'll always remember you, and I miss you every day.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

ACKNOWLEDGMENTS **xiii**

VITA **xiv**

ABSTRACT OF THE DISSERTATION **xvi**

1 Introduction **1**
1.1 Motivation 1
1.2 Overview of Contributions of the Dissertation 3
1.3 Chapter Summaries 5

2 Background **9**
2.1 Introduction 9
2.2 Stochastic Chemical Kinetics 10
2.3 Graph Theory 11
2.4 Representing Dynamic Graphs 14
2.5 Graph Rewriting Systems 16
2.6 Extended Objects and Cell Complex Theory 17
2.7 Grammars and Languages 20
2.8 Related Work 22

3 Dynamical Graph Grammar Formalism **24**
3.1 Introduction 24
3.2 Master Equation 25
3.3 Operator Process Maps 25
3.4 Chemical Reaction Rules 26
3.5 Stochastic Parameterized Grammar Rules 27
3.6 Dynamical Grammars Rules 29
3.7 Dynamical Graph Grammar Rules 29

4 Simulation Algorithms **33**
4.1 Introduction 33
4.2 Exact Hybrid Parameterized ODE SSA 33

4.3 Parallel Exact Hybrid Parameterized ODE SSA 35
4.4 Approximating the Exact Hybrid Parameterized ODE SSA 38
4.5 Phi Function 43
4.6 Conclusion 45

5 Overview and Building Blocks of DGGML 47
5.1 Overview of the Design 47
5.2 Yet Another Graph Library 49
5.3 Subgraph Specific Pattern Recognizer 52
5.4 The Expanded Cell Complex 58
5.5 Graph Transformation 61
5.6 Incremental Update 63
5.7 Differential Equation Solving 67

6 Defining a Grammar in DGGML 70
6.1 Introduction 70
6.2 Building a Model 70
6.3 Defining Stochastic Rules 75
6.4 Defining Deterministic Rules 81
6.5 Conclusion 85

7 Grammar Analysis and Simulation in DGGML 86
7.1 Introduction 86
7.2 Analyzing a Grammar 87
7.3 Simulating a Grammar 92
7.4 Conclusion 99

8 Modeling the Cortical Microtubule Array 100
8.1 Introduction 100
8.2 Biological Motivation 101
8.2.1 Mapping biology and relevant physics to dynamical graph grammars . 102
8.2.2 Overview 106
8.2.3 Experiment 1: Long-time Network Formation 107
8.2.4 Experiment 2: Long-time Local Alignment 115
8.2.5 Experiment 3: Approximate vs. Exact Performance 118
8.3 Conclusion 120

9 Revisiting the Cortical Microtubule Array Model with DGGML 121
9.1 Introduction 121
9.2 Background 122
9.3 Modified CMA Grammar 124
9.4 Results: Boundary Conditions and Alignment 127
9.4.1 The array orientation of square domains is multi-modal 128
9.4.2 Boundary conditions reorient arrays in rectangular domains 135
9.5 Conclusion 141

10 Conclusion **144**
10.1 Summary of Motivation 144
10.2 Conclusions and Future Work 145

Bibliography **151**

Appendix A Minimal CMA DGG **161**

Appendix B Periclinal CMA DGG **171**

Appendix C Approximate Algorithm Derivation **201**

LIST OF FIGURES

Page

2.1 A visual example of a graph labeled by number and color. 12
2.2 Left: A square. Center: The 2-cell, 1-cells, and 0-cells are labeled for the square. Right: The incidence (relationship) graph for the cell complex of a square. 19

5.1 Conceptual overview of library design and organization. The library organization is grouped by components falling into the core, grammar, geometry and topology, state monitoring, differential equations, utility, and dependency categories. The hierarchy of the grouping is visualized as a central blue squircle (square with rounded corners) designating the library itself surrounded by exterior grey squircles, each representing a group of components, where the black directional arrows depict compositional relationships. Within these exterior squircles, blue rectangular boxes represent individual components. We have acronyms ODE (ordinary differential equations), VTK (visualization toolkit), SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers), SIMD JSON (single input, multiple data JavaScript object notation), and YAGL (yet another graph library). 48
5.2 The graph to be searched. 54
5.3 A side-by-side view of the target graph to find one of its transformations. . . . 55
5.4 Example of how a search graph works to find an optimal path. 56
5.5 Cell complex generation and labeling. We start with a square domain, and then subdivide it into a 3×3 grid. From the grid, we generate a regular lattice graph representing the topology of the subdivided space. The regular lattice graph is then labeled by the dimension it belongs to resulting in the cell complex being labeled by dimension. 59
5.6 The pre-expansion of the cell complex generated in figure 5.5 and its expansion into well-separated lower dimensional cells. 60

6.1 A simplified example of the abstract base class for the modeling interface, templated based on the type of graph grammar provided. The GraphGrammarType defines the grammar (gamma) and provides type information for essential data structures: the key generator and system graph. The expanded cell complex is necessary for simulation space generation. The main functions include an initialize function, which must be defined, and optional checkpointing, metric collection, and metric printing functions that take in the current step as input. 71
6.2 An example program for simulating a grammar. Initially, a parser is constructed, and a configuration file is parsed. Subsequently, a user-defined model is created and its parameter settings are configured. A simulation is then instantiated based on the type of model provided by the user. Once the model is set, the simulator executes the simulation. 72
6.3 Example of a JSON-formatted configuration file for a user-defined cortical microtubule array grammar. 73
6.4 A constructor for building a stochastic rule in DGGML, where the type of rule is named by its **with** clause keyword. The arguments in order from left to right: NameType is the rule name, GraphType is the left-hand side graph type, GraphType is the right-hand side graph type, PropensityType is the type of the propensity function, and UpdateType is the type of the updating function. 77
6.5 An alternate design choice for defining a rule, not used in DGGML, is the method chaining approach. It starts by constructing the rule and then allows the user to continue building using a fluent-like interface. The sequence from left to right is as follows: the method name initializes a name with NameType n and calls fromLHS, which sets the left-hand side with GraphType g1. This process repeats for setting the right-hand side with GraphType g2, then setting the propensity with PropensityType p using with, and finally, calling where to set the update with UpdateType u. 77
6.6 The left-hand side graph is constructed by creating nodes with corresponding numbering and node types. Node 1, of type Intermediate, includes a position and unit vector. Node 2 follows suit with a type of Positive. An edge is then added between nodes 1 and 2. The process is repeated for constructing the right-hand side graph. 78
6.7 The propensity function is defined as an anonymous lambda function. Its signature includes a bracketed ampersand to indicate the compiler can capture external variables, such as the settings variable. The function arguments are automatically deduced types, enabling the user-defined function to access the left-hand side graph's matching and its associated mapping to the true labeling in the system without being concerned with resolving types. Within the function, data from the current matching is accessed, distances are calculated, and a threshold check determines the returned propensity. 79

6.8 The update function in the **where** clause is defined as an anonymous lambda function. The function arguments are automatically deduced types, allowing the user-defined function to access the left-hand side graph's matching and its associated mapping, m1, to the true labeling of the matching in the system, which is passed into the function. The same applies to the right-hand side (RHS) graph. Within the function, mappings are utilized to access and update the positions and unit vector of the resulting RHS created after a rewrite. The use of get functions is essential for retrieving the unique data associated with a given node, as it gets obfuscated due to the graph node being defined using a type that can assume several different types. Consequently, this information is only accessible to users who possess knowledge of the numbering and type of a corresponding node. 80
6.9 The code translation of equation 6.1 involves defining the rule's name during construction. The left-hand side and right-hand side graphs are defined before construction, as illustrated in figure 6.6. Similarly, the propensity function is defined before construction, as shown in figure 6.7, and the updating function is defined likewise, as seen in figure 6.8. Finally, the rule is added to a grammar named gamma. 80
6.10 A constructor for building a deterministic rule in DGGML, where the type of rule is named by its **with** clause keyword. The arguments in order from left to right: NameType is the rule name, GraphType is the left-hand side graph type, GraphType is the right-hand side graph type, NEqType is the number of variables to be bound, VarType is the variable binding function, and ODEType is the type of the ordinary differential equation solving function. 81
6.11 The left-hand side (LHS) graph is constructed by creating nodes with corresponding numbering and node types. Node 1, of type Intermediate, includes a position and unit vector. Node 2 follows suit with a type of Positive. An edge is then added between nodes 1 and 2. Since the graph structure of ODEs is unchanged after a rewrite, the right-hand side graph can just be a copy of the LHS. 82
6.12 The variable binding function is defined as an anonymous lambda function. Its function arguments are automatically deduced types, allowing the user-defined function to access the left-hand side graph's matching and its associated mapping to the true labeling in the system without resolving types. Additionally, there is "varset", which is an automatically deducible type defined by DGGML. It provides the ability to insert memory addresses of variables, effectively giving them a name that DGGML can understand. Within the function, the position variables for the node in the matching passed in as the LHS corresponding to node 2 in Equation 6.2, are bound and inserted into the "varset". 83

6.13 The **solving** function is defined as an anonymous lambda function. Its function arguments are automatically deduced types, enabling the user-defined function to access the left-hand side graph's matching and its associated mapping, m1, to the true labeling of the matching in the system, which is passed into the function. The y and ydot types are utilized for integration with the ODE solver, and via the varmap (built by DGGML from the varset), they can be employed to update a local portion of the ODE system's state vector. Within the function, the **solving** equation from (4,5) in equation 6.2 is defined and a condition is included to check if a variable is coupled to another equation, using the correct value if necessary. 84
6.14 The code translation of equation 6.2 involves defining the rule's name during construction. The left-hand side and right-hand side graphs are defined before construction, as illustrated in figure 6.11. The number of vars is also included. Similarly, the vars function is defined prior to construction, as shown in figure 6.12, and the solving function is defined likewise, as seen in figure 6.13. Finally, the rule is added to a grammar named gamma. 84

7.1 A section of an abstract syntax tree (AST) for a DGG, where circle nodes depict syntactic constructs composed of other nodes, while square nodes represent "leaf" or "terminal nodes", denoting the smallest syntactic units. Arrows pointing out from a node indicate its composition of the pointed to nodes. Dotted lines signify the repetition of a subtree with similar properties. 88
7.2 A transformed stochastic rule subtree, originating from an abstract syntax tree representing a grammar. In this hierarchy, squares represent data with arrows leaving a square indicating its composition with the pointed elements. Dotted lines denote the possible omission of additional instances of the names and "WithRules." 89
7.3 An overview of the parsing process, elucidating how graphs from the left-hand side of all grammar rules are parsed into their fundamental components (motifs) for each graph. Subsequently, these motifs are consolidated into a set of unique motifs, the shared components. In this hierarchy, squares represent data, with arrows leaving a square indicating its composition with the pointed to elements. Dotted lines denote the possible omission of additional instances at that particular layer. 90
7.4 A pattern matching hierarchy, elucidating how fundamental components graphs (motifs) from a consolidated set of unique motifs are utilized to identify instances of the matching motif pattern. In this hierarchy, squares represent data, with arrows leaving a square indicating its composition with the pointed elements. Dotted lines denote the possible omission of additional instances at that particular layer. 91

7.5 Expanded cell complex (ECC) with well-separated lower dimensions. Regions of the same pre-expansion dimension are separated from each other. Note how only interior lower dimensional cells are expanded. A reaction grid is aligned with the geocells, and reaction cells are smaller than the geocells. The outer boundary of the ECC is padded with optional ghosted geocells. Ghosted geocells are just geocells that are not processed by Algorithm 4. These optional ghost cells operate as a buffer for any computational errors or as a capture condition in the case of no grammar rules addressing boundary conditions. 95

8.1 Summary of all the rules used in the CMA grammar. 105
8.2 Side by side comparison of beginning and end state of the CMA DGG simulation of 1600 MTs for realization 3. 109
8.3 Six realizations of the change in connected components over time. 109
8.4 Zoomed in plots of the beginning and end of six realizations of the number of connected components changing over simulation iterations, where one realization becomes a fully connected network 110
8.5 Plot of the long-term behavior of all node types, including all six realizations of the CMA DGG simulations. 110
8.6 Plot of the change of zippering nodes over time for all six realizations of the CMA DGG simulations. 111
8.7 Plot of the change of positive nodes over time for all six realizations of the CMA DGG simulations. 112
8.8 Plot of the change of negative nodes over time for all six realizations of the CMA DGG simulations. 113
8.9 Plot of the change of intermediate nodes over time for all six realizations of the CMA DGG simulations. 113
8.10 Plot of run-time per simulation iteration of six realizations of the CMA DGG simulations. 115
8.11 Side-by-side comparison of the end states of the CMA DGG simulation of 1600 MTs on a 100x100 unit grid showing the effect crossover has on the system. . 116
8.12 Sampled correlations of alignment over distance and their exponential fits of ending system states seen in 8.11 117
8.13 Plot of performance analysis of five separate simulation runs for 10 iterations. 118
8.14 Natural log scaled plot of performance analysis of five separate simulation runs for 10 iterations. 119

9.1 Visualization of our approximation of the cell as a polyhedral prism (top: periclinal, sides: anticlinal) with our approximation of the "picket fence" phenomena, where microtubules (MTs) in the anticlinal faces are aligned in a perpendicular array pattern [106]. Newly created MTs forming in the anticlinal face or those aligned to existing MT orientation in the anticlinal faces are likely oriented to enter the periclinal faces at angles perpendicular to the edge with some variance [106]. 124

9.2 Side by side example of what simulated system looks like, and what we mean by boundary graph and nucleation points. 126
9.3 For the square experiment with a CIC boundary, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean. 129
9.4 The plot contains the a best-fit logarithmic curves ($f(t) = a\log(bt)+c$) for the 120 minute range of interest fitted to the local and global mean correlation over time for each of the square simulation experiments. All simulations start empty and have zero correlation. 130
9.5 Histograms of the mean angular orientation for the arrays at $t = 120$ minutes for the square CIC boundary and CLASP+Crossover and averaged over 16 runs. 131
9.6 Histograms of the mean angular orientation for the arrays at $t = 120$ minutes for the square influx of microtubules averaged over 16 runs, along with an ending state sample of the influx experiment. 132
9.7 A closer look at the KDE with a Gaussian kernel and the Scott bandwidth [103] of the orientation PDF from sample 4 of the CLASP 30° experiment, along with the histogram it was estimated from. In this case, the resulting density is a close match for a qualitative representation of the shape of the histogram. 133
9.8 An example with three samples of an estimated orientation PDF from the CLASP 30° experiments, along with the ending state of sample 7 from the CLASP 30° experiment. 134
9.9 Two examples with three samples of estimated orientation PDFs from the CLASP 45° and 60° experiments 134
9.10 For the rectangular experiment with CIC Boundary, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean. 135
9.11 The plot contains the best-fit logarithmic curves ($f(t) = a\log(bt)+c$) for the 120 minute range of interest fitted to the local and global mean correlation over time for each of the rectangular simulation experiments. All simulations start empty and have zero correlation. 136
9.12 Histogram of the mean angular orientation for the arrays at $t = 120$ minutes for the rectangular case with a CIC boundary averaged over 16 runs, along with an ending state sample of the CIC experiment. 137
9.13 Histogram of the mean angular orientation for the arrays at $t = 120$ minutes for the rectangular case CLASP rule enabled and a crossover rate of $8,000$ averaged over 16 runs, along with an ending state sample of the CLASP+Crossover experiment. The ending state has network-like behavior similar to figure (8.11a) in Chapter 8. 138
9.14 Side-by-side comparison of estimations of the array orientation and a sample for the rectangular case with CLASP and an influx of microtubules entering the boundary. 139

9.15 Two examples with three samples of estimated orientation PDFs from the CLASP 30° and 45° experiments in rectangular domains. 140
9.16 An example with three samples of an estimated orientation PDF from the CLASP 60° experiment, along with the ending state of sample 3 from the CLASP 60° experiment. 141

ACKNOWLEDGMENTS

I would like to thank my advisor Eric Mjolsness for all his support, guidance, and patience as I explored the fascinating intersection between computational science and biology. Thanks to him, I've grown as a scientist, software engineer, and a writer. I would also like to thank my co-advisor and program director Jose Castillo, who brought me into the computational science PhD program. Additionally, I'd like to thank the other members of my committee. I'd like to thank Peter Blomgren for teaching me about numerically solving PDEs and saving the tough proofs for dark and stormy nights, Jun Allard for our discussion on microtubules, and Charless Fowlkes for teaching me the fundamentals of computer vision. I would also would also like to acknowledge valuable conversations with Jacques Dumais, Olivier Hamant, and Elliot Meyerowitz.

I would also like to thank my amazing partner, Hina. I am so thankful for her supporting me every step of the way and being there for me even in the hardest of times, and also being my sounding board and statistical wizard on call. I truly couldn't ask for a better partner and she deserves more thanks than I could ever express in words.

To my family, thank you for always being there even if most of the time you never quite knew what I was doing. If you'd like to find out, take a read, if you dare! So, thank you Heather and Josh, Holly and Hunter, Michelle and Josh, and all the nieces and nephews. A special thanks to Josh for all the memes you sent to get me through the rough days, you know which one you are! Also, thank you Dad for just being there, it meant a lot.

To my friends, thank you too. Without you, I wouldn't be where I am. Thanks to AJ for encouraging me to take on grad school, I'll never forget our whiteboard days at CMC. Thank you Tristan for always being around to remind me that sometimes I just need to take a break and do something fun, and than you Angie for always being so optimistic. Thank you Matt for all of our chats over coffee and Niloufar for helping me see things the simple way. To my friend's Eva and Jeremiah, thank you both for always being there through the years. Also, thank you Jeremiah for inspiring me to be the best scientific software writer I can be. Finally, to all my gym friends, thanks for keeping me in shape.

I'd also like to thank some things from media that have inspired me. To Jean Luc-Picard, thank for teaching me that space is the final frontier. To Pikmin, thank you for teaching me the valuable art of Dandori and keeping me organized.

Finally, this work was funded in part by U.S. NIH NIDA Brain Initiative grant 1RF1DA055668-01, U.S. NIH National Institute of Aging grant R56AG059602, Human Frontiers Science Program grant HFSP—RGP0023/2018, and the UCI Donald Bren School of Information and Computer Sciences. This work was supported in part by the UC Southern California Hub, with funding from the UC National Laboratories division of the University of California Office of the President.

VITA

Eric Medwedeff

EDUCATION

Doctor of Philosophy in Computational Science **2024**
University California Irvine *Irvine, CA*

San Diego State University *San Diego, CA*

Bachelor of Science in Mathematics **2017**

California State University San Bernardino *San Bernardino, CA*

Associates of Science in Mathematics **2015**
Copper Mountain College *Joshua Tree, CA*

RESEARCH EXPERIENCE

Graduate Research Assistant **2019–2024**
University of California, Irvine *Irvine, California*

Data Science Initiative Summer Intern **2021**

Lawrence Livermore National Lab *Livermore, California*

Parallel Computing Summer Intern **2020**

Los Alamos National Lab *Los Alamos, New Mexico*

Graduate Research Assistant **2018–2019**

San Diego State University *San Diego, California*

TEACHING EXPERIENCE

Teaching Assistant **2017–2018**
San Diego State University *San Diego, CA*

REFEREED PUBLICATIONS

Eric Medwedeff and Eric Mjolsness
Advances in the Simulation and Modeling of Complex Systems using Dynamical Graph Grammars
Available on arXiv (In Review Pending) **2024**

Eric Medwedeff and Eric Mjolsness
Approximate Simulations of Cortical Microtubule Models using Dynamical Graph Grammars
Special Issue Dedicated to the Fifteenth q-bio Conference, Physical Biology **2023**

Jacob Marks, Eric Medwedeff, Ondrej Certik, Robert Bird, Robert W. Robey
Improving Fortran Performance Portability
International Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science **2022**

Xiaobai Liu, Qian Xu, Grayson Adkins, Eric Medwedeff, Liang Lin, and Shuicheng Yan
Learning Semisupervised Multilabel Fully Convolutional Network for Hierarchical Object Parsing
IEEE Transactions on Neural Networks and Learning System **2020**

SOFTWARE

DGGML `https://github.com/emedwede/DGGML`
A C++ dynamical graph grammar modeling library.

YAGL `https://github.com/emedwede/YAGL`
Yet another C++ graph library.

CajeteCMA `https://github.com/emedwede/CajeteCMA`
C++ simulator for simulating the cortical microtubule array of plant cells using dynamical graph grammars.

ABSTRACT OF THE DISSERTATION

Approximate Simulations of Dynamical Graph Grammars
using the
Dynamical Graph Grammar Modeling Library

By

Eric Medwedeff

Joint Doctor of Philosophy in Computational Science

University of California, Irvine, July 2024

Professor Eric Mjolsness, Chair

Dynamical graph grammars (DGGs) are capable of modeling and simulating the dynamics of complex biological systems using an exact simulation algorithm derived from a master equation; however, the exact method is slow for large systems. To accelerate the simulations of DGGs we have developed an approximate simulation algorithm that is compatible with the DGG formalism. The approximate simulation algorithm uses a spatial decomposition of the domain at the level of the system's time-evolution operator, to gain efficiency at the cost of some rules or reactions firing out of order, which may introduce errors. The decomposition is coarsely partitioned by effective dimension (d=0 to 2 or 0 to 3), to expose the potential for exact parallelism between different subdomains within a dimension, where most computing will happen, and to confine errors to interactions between adjacent subdomains of different effective dimensions. Additional efficiency can be achieved through maintaining an incrementally updated match data structure for all possible rule matches. To demonstrate these principles we have developed the Dynamical Graph Grammar Modeling Library (DGGML), and two DGG models for the plant cell cortical microtubule array (CMA). In the first model, we find evidence indicating that the initial formulation of the approximate algorithm is substantially faster than the exact algorithm, and one experiment leads to net-

work formation in the long-time behavior, whereas another leads to a long-time behavior of local alignment. In the second model, we restrict ourselves to the CMA in the periclinal face of a plant cell and explore the effects that different face shapes and boundary conditions have on local and global alignment. In the case of a square face shape, we find the array orientation to be multi-modal, and in the case of a rectangular face shape, we find that different boundary conditions reorient the array mainly between the long and short axes. The periclinal CMA DGG demonstrates the flexibility and utility of DGGML and highlights its viability to be used as a computational means of testing, screening or inventing hypotheses to explain emergent phenomena.

Chapter 1

Introduction

1.1 Motivation

Static graphs are a fundamental concept in mathematics and computer science [36], and intuitively allow us to represent relationships between objects and concepts. Dynamic graphs, on the other hand, are graphs that can change over time and allow us to intuitively represent changing relationships. The theory of graph grammars found in [95] provides a comprehensive mathematical framework to understand how dynamic graphs become dynamic by means of expressive rewriting systems. A dynamic graph, and in turn a graph rewriting system can, be further equipped with a high-level language that maps graphs to a master equation resulting from an operator algebra framework, effectively enabling the dynamic graph to become dynamic through a stochastic rewriting process. Dynamical Graph Grammars (DGGs) [76] allow for an expressive and powerful way to declare a set of local rules to model a complex dynamic system with graphs.

DGGs also have a well-defined meaning. They map graph dynamics into a master equation, a set of first-order linear differential equations governing the time evolution of joint probability distributions of state variables of a dynamic system. Using operator algebra [75], DGGs can be simulated using an exact algorithm (Chapter 4, Algorithm 1) that subsumes Gillespie's Stochastic Simulation Algorithm (SSA) [48], which is closely related to the kinetic Monte Carlo algorithms of statistical physics [131]. As does the SSA, the exact algorithm becomes slow for large systems and signals a need to develop a faster and more scalable algorithm. Using operator splitting, an approximate faster and scalable algorithm can be derived for spatially embedded graphs (Chapter 4, Algorithm 4).

The original implementation of the DGG formalism is realized by the software Plenum [129] with powerful symbolic processing provided by Mathematica [128], but only uses the exact algorithm and does not use native graph data structures or graph algorithms. To allow for a more graph and performance-oriented implementation, we have developed the Dynamical Graph Grammar Modeling Library (DGGML) in the C++ programming language. DGGML trades the symbolic processing found in Plenum for native graph data structures and performance. DGGML also serves as a way to directly demonstrate the utility of the new approximate algorithm. The design and usage of DGGML discussed in this thesis also serves as a base or guide for future implementations of the DGG formalism as an independent language with its own compiler, along with informing any designer or user of the trade-offs between simulation algorithms.

To demonstrate the usage and utility of DGGML, particularly on the user side, we focus on two different biological models for the plant cell cortical microtubule array (CMA), where all graph rules are for graphs spatially embedded in Euclidean space. The first model (Chapter 8 and Appendix A) is implemented using the precursor [73] to DGGML, while the second model (Chapter 9, and Appendix B) is completely defined with the DGGML user-interface and simulated using the implementation of Algorithm 4 of Chapter 4 provided by DGGML. Both models are presented to showcase the evolution of DGGML and the flexible power of DGGs.

These models are also presented to demonstrate how DGGs can be used to search for corresponding wet-lab experiments that would distinguish between and test alternative hypotheses. In our case, these digital stand-ins enable our investigation of the CMA in plant cells. Within the CMA, the mechanisms and the general principles governing the organization of cortical microtubules into functional patterns have long been studied [123], but there are still many outstanding questions [39]. The inspiration for the models presented in this thesis comes from this yet-to-fully-be-explored frontier, and the simplified models we have developed using DGGs and simulated using DGGML for CMA dynamics come with the potential to be extended to include more complex dynamics and interactions working together at different scales of space.

Work has already been done to simulate the dynamics of MTs in plants [84], [24]. However, to our knowledge, there is no other known formalism that does it by using dynamic graphs. The only previous work is found in Plenum [129] and theoretically discussed in [76] and [130]. Finally, the algorithms, library, and models with corresponding simulations presented in this thesis are intended to fully demonstrate the synergy between the technical computing side of computational science and the mathematical modeling side of systems biology.

1.2 Overview of Contributions of the Dissertation

The dissertation makes several significant novel contributions to the research area of Dynamical Graph Grammars (DGGs) and makes efforts to make the entry point into this field clear. It begins by introducing key concepts such as stochastic chemical kinetics, the stochastic simulation algorithm (SSA), and graph theory fundamentals, laying the groundwork for developing an approximate algorithm to improve simulation performance. The iterative introduction of the DGG formalism from Stochastic Parameterized Grammars to Dynamic Grammars to DGGs keeps the historical evolution of the topic intact while building out the general formalism. The dissertation also details the original approximate hybrid parameterized ODE SSA algorithm (Algorithm 3), leveraging operator splitting and an expanded cell complex to decompose simulation spaces effectively while maintaining accuracy. We also present and even more efficient and improved version (Algorithm 4). The algorithm includes an incrementally updated match data structure composed of all rule instances and all possible LHS connected component instances, greatly reducing the need to recompute matches after every rule firing event. We also have contributed novel methods for mapping rules to geometric cells (geocells) in the form of the function φ.

The Dynamical Graph Grammar Modeling Library (DGGML) is presented as a realization of the DGG formalism and the approximate hybrid parameterized ODE SSA algorithm. Notable developments to this thesis made through the implementation of DGGML include Yet Another Graph Library (YAGL), the expanded cell complex (ECC), an incremental update mechanism for the collection of possible rule firings, a unique solution to the subgraph recognition problem for labeled graphs, and guidelines for grammar creation and language interfacing. We have also implemented a first-of-its-kind analysis framework for the left-hand side of DGG grammar rules. Collectively, these developments have enhanced DGGML's versatility and usability for dynamic system modeling.

Two practical applications of DGGs are presented as well. The first DGG model is for the plant cell cortical microtubule array (CMA) (Appendix A), which demonstrates the performance gained by using the approximate algorithm. Simulations of the CMA DGG had two outcomes: the formation of a highly connected network, and the local alignment of microtubules (MTs). The second model, the periclinal CMA (PCMA) (Appendix B), demonstrates DGGML in action, along with providing a unique experimental framework for analyzing array orientations. Through the use of the PCMA DGG, we were also able to demonstrate that the shape of the periclinal cell face and the dynamics that play out on the boundary may have an impact on the types of orientations observed in the long-time behavior. In the case of a square face, we discovered that the orientations could be multi-modal, regardless of the boundary. In the rectangular case, we found that a collision-induced catastrophe boundary led to alignment with the longest axis and that an influx of MTs on the boundary was enough to switch the axis of alignment to the shortest.

1.3 Chapter Summaries

This chapter (Chapter 1) introduces the motivation for the thesis, highlights key contributions, and provides chapter summaries. Chapter 2 introduces several concepts required to understand the DGG formalism and its origins, plus essential concepts for discussing the implementation of the library. Stochastic chemical kinetics and the stochastic simulation algorithm (SSA) are discussed [49] to support the need for the approximate algorithm [73] presented in this work, along with performance improvements. Some graph theory [36] is presented to make statements about graphs and related concepts more precise, since graphs are at the core of the DGG formalism [76]. Several methods for representing dynamic graphs and trade-offs are discussed, and this transitions directly into the topic of graph rewriting systems. Both dynamic graphs and rewriting systems play a key role in the design choices made to implement the Dynamical Graph Grammar Modeling Library (DGGML). Building upon the notion of graphs, extended objects [76] and cell complexes are introduced to clarify what types of objects are simulated and to provide a background for the expanded cell complex. Chapter 2 concludes with a formalization of DGGs as a declarative modeling language and claim 1 (model existence), followed by a brief overview of related biological modeling tools.

Chapter 3 introduces the DGG formalism iteratively. Dynamical Graph Grammars (DGGs) are a further refinement of the Dynamic Grammars, which generalized the Stochastic Parameterized Grammars by the inclusion of differential equation rules. DGGs include all the related formalisms of SPGs and DGs, along with an additional and expressive modeling language framework for graphs. In this dissertation, the DGG formalism contributes to the design and implementation of the Dynamical Graph Grammar Modeling Library (DGGML).

Chapter 4 presents the original hybrid parameterized ODE SSA (Algorithm 1) [75], along with contributing suggestions for a parallelized version (Algorithm 2) and most importantly introduces the approximate hybrid parameterized ODE SSA (Algorithm 4). The approximate algorithm is a major contribution, providing an operator splitting algorithm that imposes a domain decomposition using a novel expanded cell complex that corresponds to summing operators, over pre-expansion dimensions d, and cells c of each dimension. Two notable assumptions were made for the approximation of the exact algorithm: spatial locality of the rules and well-separatedness of the cells in the expanded cell complex used to decompose the simulation space into domains. We also introduce the match data structure at the heart of Algorithm 1 and several methods to map rules to geocells by means of the function φ.

Chapters 5, 6, and 7 detail the DGGML framework and its realization of Algorithm 4. Chapter 5 presents an overview of DGGML along with its essential components. Notable contributions include Yet Another Graph Library (YAGL), a dynamic graph library designed for DGGML, and several foundational elements such as the spatial variant node, subgraph specific pattern recognizer (SSPR), expanded cell complex (ECC), and the process for incrementally updating the set of rule matches after graph rewrites occur.

YAGL, which functions as a header-only library, complements DGGML by handling the requirements of a dynamic graph. The ECC, a crucial development, offers a specialized graph data structure essential to capture both geometric and topological aspects of the simulation space, a key component of the Algorithm 4. The SSPR embodies a heuristic approach to address the subgraph recognition problem for labeled graphs, significantly enhancing DGGML's versatility. The incorporation of an incremental update mechanism allows for partial invalidation of matched rule instances post-rule firing, facilitating more efficient system updates. The chapter also includes a section on graph transformations for a more comprehensive understanding of rewriting.

Chapter 6 provides general guidelines for creating a grammar, and demonstrates how these guidelines are applied in DGGML through interactions with library interfaces and how they inform modeling choices. Furthermore, it aims to demonstrate what the integration of the DGG language and formalism into another programming language looks like. Chapter 6 also describes the user interface for DGGML and contributes two examples. The first example presented describes how to transform a higher-level description of a stochastic rule using the DGG formalism into a format compatible with the programming interface exposed in DGGML. The same process is also applied to a deterministic solving rule. Chapter 6 contributes and opens up an accessible entry point into the design of DGG language interfaces, while offering insights into the considerations necessary when translating Dynamic Graph Grammars (DGGs) into a working implementation in a target language.

The insights are further carried into Chapter 7, which builds on Chapters 5 and 6 to explore the fundamental details of how DGGML internally analyzes and simulates a grammar. The analysis phase contributes a novel hierarchy for creating a set of pattern matchers for spatially embedded graphs with connected components used as fundamental pattern objects. The simulation phase takes the building blocks and puts them into action, completing the picture of how Algorithm 4 is implemented in the library.

Chapter 8 uses DGGs to discuss a model for the cortical microtubule array (CMA) in plant cells (Appendix A), which is a complex system with many subsystems that are well-suited to be modeled with DGGs. The chapter is taken from previously published preliminary work [73] using the approximate algorithm (Algorithm 4) implemented as a prototype simulator, where three simple experiments are run to test the viability of simulating the CMA and the algorithm. From these results, we find evidence indicating the initial formulation of the approximate algorithm is substantially faster than the exact algorithm, and one experiment leads to network formation in the long-time behavior, whereas another leads to local alignment.

Chapter 9 revisits the CMA using DGGML. A new model is developed (the periclinal CMA (PCMA) Appendix B) to investigate what effects the shape of the periclinal face and boundary conditions near the edges of the face have on the alignment orientation of the array. In particular, we focus on two face geometries: square and rectangular. For the boundaries, we focus on two cases: collision-induced catastrophe (CIC) and entry and exit of microtubules (MTs) mediated by cytoplasmic associated linker proteins (CLASP). In the case of the square geometry, we were able to determine that the orientation of the array was multimodal except when a high crossover rate led to network-like behavior. In the case of the rectangular domain, we found that changes to the boundary conditions reorient the array between long and short axes. Finally, Chapter 10 concludes the dissertation and provides direction for future work.

Chapter 2

Background

2.1 Introduction

Dynamical graph grammars (DGGs) offer a powerful formalism for modeling and analyzing complex systems, particularly those characterized by dynamic and evolving structures. Driven by a stochastic process defined by a master equation, DGGs are directly related to foundational work on stochastic chemical kinetics. However, DGGs are more expressive and form a declarative modeling language based on the rewriting of graphs. The graphs themselves are used to represent objects, relationships, and spaces. This chapter presents background information on stochastic chemical kinetics, graph theory, dynamic graphs and their representations, graph rewriting systems, extended objects and cell complex theory, grammars, and languages. Additionally, examples of modeling tools similar to the dynamical graph grammar modeling library (DGGML) are presented. The topics in this chapter form a foundation for the DGG formalism and the remainder of the thesis.

2.2 Stochastic Chemical Kinetics

The SSA is an example of *stochastic* modeling, as opposed to the *deterministic* modeling approach [68]. In a deterministic approach, the time-continuous processes are wholly predictable and can be governed by the *reaction-rate equations* [67], a set of coupled ordinary differential equations (ODEs). The stochastic approach is a type of random-walk process that is completely encoded in the *master equation*. The master equation itself is a high-dimensional linear differential equation, $P'(t) = W \cdot P(t)$, which governs the rate at which probability p flows through different states in the system. However, systems can become very large due to an exponential state-space explosion with respect to the number of biological variables, and the systems may have infinite-dimensional state spaces, making the analytical solution to the master equation computationally intractable or impossible.

Kinetic Monte Carlo methods have been used in different applications, such as Ising spin systems [16] and the work of Gillespie [50], where he uses the Monte Carlo method and kinetic theory to rigorously derive the exact stochastic simulation algorithm (SSA) for chemical kinetics. The derivation in chemical kinetics makes a case based on several assumptions about the systems, the most important being that the system contains a large number of molecules well-mixed at thermal equilibrium. After making key assumptions, it is necessary to set reaction rates - which can be difficult to determine. Three routes for determining rates are lab measurements, giant *ab initio* quantum mechanical calculations or machine learning generalizations thereof, and parameter optimizations in the context of system-level observations together with the use of other known reaction rates that are more easily measurable. Finally, an event is sampled from a conditional density function (CDF). The event that occurs or *"fires"* is effectively a *reaction*. The Monte Carlo procedure does not give the analytic solution to the master equation, but it does yield an unbiased sample trajectory of a system. It effectively provides a *realization* through numerical simulation.

As powerful as the exact SSA is, it is prohibitively slow, since each reaction event must be computed in order. Numerous methods have been proposed to speed up the exact SSA. τ-Leap [51] fires all reactions in a window of τ before updating propensity functions, saving computation at the cost of errors. Later, it was made even more efficient [21]. R-Leaping [8] lets a preselected number of reactions fire in a simulation step, again at some cost in accuracy. The Exact R-Leap, "ER-Leap" [79] modifies the R -Leaping algorithm to be exact and provides a substantial speed-up over SSA. ER-leap was later improved upon and parallelized in $HiER$-Leap [85]. More recently, S-Leap [70] was introduced as an adaptive, accelerated method that bridges the methods of τ-Leaping and R-Leaping. There are many other works on speeding up the original SSA as well.

This dissertation work builds on this rich history and complements it. The goal is not to just enable the simulation of a master equation for stochastic chemical kinetics. Instead, the goal is to enable the solving of a broader class of problems in biology and beyond, by representing the dynamics of spatially extended objects using graphs. The foundational work for the mathematical theory will be briefly discussed in the DGG formalism section and the curious reader may refer to [75, 80, 76] for more detailed information.

2.3 Graph Theory

The following graph theory and notation are presented as references. A *graph* (*undirected*) $G = (V, E)$ is a set of V vertices and a set of edges $E \subseteq \{\{u, v\} | u, v \in V\}$, each an unordered pair of V, where elements $u, v \in V$ are vertices. $V(G)$ is the set of vertices of graph G and $E(G)$ is the set of edges of graph G. Now, let $G = (V, E)$ and $G' = (V', E')$. There is a homomorphism from G to G' if there exists a mapping $f : V \longrightarrow V'$ that preserves the adjacency of vertices, i.e. $(v, v') \in E \implies (f(v), f(v')) \in E'$. This is called a *homomorphism* from G to G'. If the function f is bijective and its inverse is a homomorphism, then f is

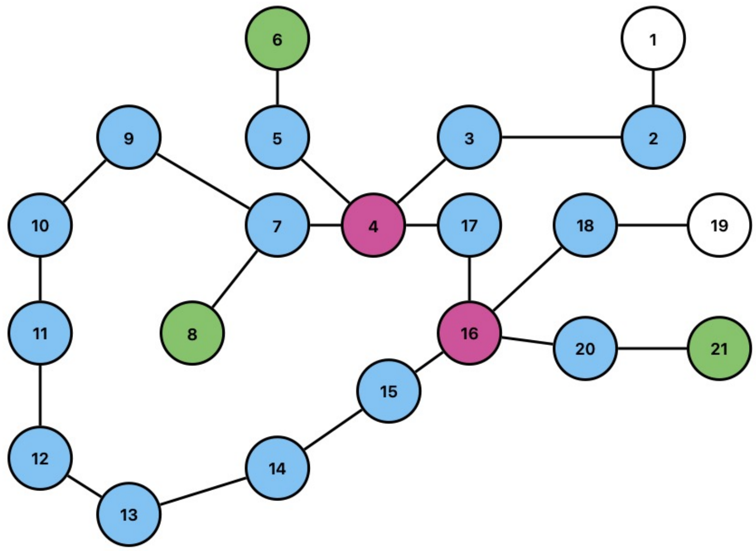


Figure 2.1: A visual example of a graph labeled by number and color.

also a [36] *isomorphism*, and graphs G and G' are *isomorphic*.

A *labeled graph* G is defined as $G = (V, E, \alpha)$, where (V, E) is a graph and $\alpha : V \longrightarrow L$ is a function that assigns labels to vertices. To each labeled graph, there corresponds an (unlabeled) graph (V, E) without the labels. A *label-preserving homomorphism of labeled graphs* is defined to be a graph homomorphism that preserves the labels exactly, without remapping them. A *match* is defined as an injective label-preserving graph homomorphism $G \hookrightarrow G'$. Informally, a match locates a "copy" of G as a subgraph inside G' for which all vertices, edges, and labels are preserved.

A labeled graph can be seen in figure 2.1. Here, the nodes are uniquely labeled using positive integers and the edges remain unlabeled. The discrete vertex labels have been mapped to a color set and visualized with those colors. In this case, the graph has no spatial embedding, so it could be visualized in many different ways.

If an isomorphism, $G \Longleftrightarrow G'$, exists then G and G' are isomorphic and $G \simeq G'$. A corollary of isomorphism is automorphism i.e. an isomorphism from the graph G to itself is an *automorphism*. Now, let $G \cup G' \equiv (V \cup V', E \cup E')$ and $G \cap G' \equiv (V \cap V', E \cap E')$. If $G \cap G' = G$, then G' is a *subgraph* of G can be written as $G' \subseteq G$. On the other hand, if $G \cap G' = \varnothing$, the two graphs are *disjoint*. Let H be a graph. If $G' \subseteq G$ and $H \simeq G'$, then H is *isomorphic to a subgraph* of G.

The set of *neighbors* for a vertex $v \in V$ can be denoted as $N_G(v)$. Sometimes this is referred to as the *frontier* of v in G. The *degree* of the vertex $d(v) \equiv |N_G(v)|$, and represents the cardinality of the set of neighboring vertices connected to v in the graph G. Any vertex of degree 0 is said to be *isolated*. Let $\delta(G) \equiv \min\{d(v)|v \in V\}$ and $\Delta(G) \equiv \max\{d(v)|v \in V\}$, where the former is the *minimum degree* of G and the latter is the *maximum degree* of G. Take the *average degree* to be: $d(G) \equiv \frac{1}{|V|}\sum_{v \in V} d(v)$. It follows then, $\delta(G) \leq d(G) \leq \Delta(G)$.

We let a *path graph* be of the form $P = (V, E)$ and the vertex set $V = \{x_0, x_1, \ldots, x_k\}$ and the edge set $E = \{(x_0, x_1), (x_1, x_2), \ldots, (x_{k-1}, x_k)\}$, where all vertices x_i are distinct. The vertex x_0 is the start of the path and the vertex x_k is the end of the path. So, the graph P forms a path between x_0 and x_k. Further, k is the number of edges, and a graph of *path length* k can be written as P^k. If however $E = \{(x_0, x_1), (x_1, x_2), \ldots, (x_{k-1}, x_k), (x_1, x_0)\}$ has all vertices distinct except for $x_0 = x_k$, this is called a *cycle*. Any edge $e \in G$ that joins two vertices, (x_i, x_j), of a cycle but is not itself contained in the cycle is a *chord* of G. A cycle without a chord is a *circle*. A graph with cycles is *cyclic* and a graph without cycles is *acyclic*.

A graph is *connected* if any two of its vertices can be reached by a path $P^k \subseteq G$. The negation of connected is *disconnected*. A *component* of G is a *maximally connected subgraph*. The set of connected components of a graph G is $N_C(G)$ and the cardinality of the set, i.e. number of connected components, is $|N_C(G)|$. So, $|N_C(G)| \leq 1$ if G is connected and disconnected if $|N_C(G)| > 1$. If $|N_C(G)| = |V|$, then it can be said that the graph is *fully disconnected*. An interesting example of this would be a set of n particles represented as vertices, each of which has no connections to any other.

An acyclic graph is also called a *forest*. A forest that is connected is a *tree*. A forest is then a graph with components that are trees. A *directed graph* (*digraph*) is a graph where edges have a direction. For a tree, it is sometimes useful to treat one vertex of the tree as unique. The selected vertex is the *root*. When the root is fixed, it is a *rooted tree*. Now, let G be a graph and T be a tree. T is a *spanning tree* of G if $T \subseteq G$ and T includes all of the vertices of G. If one vertex of T is selected to be considered special and kept fixed, then it is a *rooted spanning tree*. Rooted spanning trees can be used to construct depth-first search trees [36]. This fact is used in the heuristic pattern recognition and matching code derived in Chapter 5. Additionally, in the context of this thesis, a *spatially embedded graph* is a labeled graph where the label map α maps vertex v to a feature vector belonging to the d-dimensional real number space, $\mathbb{R}^d$.

A *dynamic graph*, $G(t)$ is a graph that changes over time. The change can either be in the form of vertex/edge creation or destruction, the change of label parameters, or both. Mathematically, $G(t) = (V(t), E(t), \alpha_t)$, where $\alpha_t : V(t) \longrightarrow L$.

2.4 Representing Dynamic Graphs

Dynamic graphs are graphs that change over time, in contrast to static graphs that remain fixed. Two primary methods for representing static graphs are adjacency matrices and adjacency lists. The adjacency matrix, despite its higher space complexity of $\mathcal{O}(|V|^2)$, where $|V|$ denotes the number of vertices, offers quick $O(1)$ lookup times, making it suitable for dense static graphs. In contrast, adjacency lists, which store only vertex connections, are more space efficient, with a space complexity of $\mathcal{O}(|V|+|E|)$ where $|E|$ represents the number of edges. The space efficiency is particularly valuable for sparse graphs. However, their lookup speeds are comparatively slower, with a time complexity ranging from $\mathcal{O}(1)$ to $\mathcal{O}(|V|)$, depending on the data structure used (e.g., arrays, linked lists). These differences between design choices highlight the careful consideration that must be made between the trade-offs of space efficiency and lookup time complexity, especially in dynamic graph scenarios where frequent rewrites and alterations to the node and edges occur.

Variations of adjacency matrices and lists, including Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Coordinate List (COO), have been developed to optimize efficiency for specific use cases. CSR and CSC formats are ubiquitous in numerical computing, optimizing the data's structure for matrix-vector operations such as matrix multiplication. On the flip side, the COO format excels in random access times and offers flexibility for sorting algorithms, making it advantageous for scenarios where a specific edge or vertex order is required. COO effectively stores the contents as a list of tuples in the format (row, column, value).

While these representations excel in optimizing operations on static graph operations, recent efforts have concentrated on accelerating these operations and crafting algorithms that harness GPU computing power for dynamic graphs. Recent work has been done to aid in this effort by designing algorithms that run on the GPU [124, 122, 20].

For dynamic graphs, a specialized combination of approaches is required. In this thesis, a dynamic graph representation defined explicitly for the CPU has been developed and implemented in Yet Another Graph Library (YAGL) with more details found in Chapter 5. A potential future improvement for YAGL would be to extend the library to work for GPUs as well. This type of extension would ensures adaptability and scalability across diverse computing architectures, effectively addressing the evolving use cases of dynamic graphs.

2.5 Graph Rewriting Systems

Generalized graph rewriting systems [56], also known as graph transformation systems, represent another powerful formalism for modeling and analyzing complex systems, particularly those with inherently dynamic and evolving structures. At their core, graph rewriting systems describe transformations that occur on graphs, where nodes represent objects, and edges denote relationships or connections between these objects. These transformations capture the dynamic behavior of systems by specifying rules that dictate how the graph can be modified over time, further motivating the need for a dynamic graph and illuminating the relationship between rewriting systems and DGGs.

In a graph rewriting system, a rule typically consists of a pattern graph (in the DGG formalism, a left-hand side graph). The pattern graph defines the structure to be matched within the existing graph (in the DGG formalism, the system graph). A rule also has a replacement graph, which specifies the new structure to be generated upon successful application of the rule or, in the DGG case, a firing of a rule. In a general rewriting system, when a rule's pattern matches a subgraph of the current graph, the corresponding portion of the graph is "rewritten" (as defined in Chapter 5) according to the rule, resulting in a modified graph. This process of applying rules iteratively enables the system to transition from one state to another, reflecting changes due to whatever process is driving the underlying system dynamics.

Graph rewriting systems are peculiar because they tend to hide in plain sight and have a rich field of applications in computer science, biology, chemistry, and other fields. In the computer science world, they can be used for modeling and analyzing concurrent and distributed systems [34], formal verification [127], programming [7], query languages [94], and the emerging plethora of graph databases [71]. In biology and chemistry, graph rewriting systems can be used to model biochemical reaction networks [105], enabling researchers to simulate and study complex biological processes, as is made evident in this dissertation.

One of the key advantages of graph rewriting systems lies in their expressive power and flexibility, allowing for the representation of a broad range of dynamics, different systems, and structural transformations. The graph rewriting framework can be further specialized, by defining a graph rewriting system where rewrites are driven by a stochastic simulation algorithm derived from the master equation, $P'(t) = W \cdot P(t)$. From the specialization, a formal mathematical framework is available by means of the operator algebra in the DGG formalism [76, 77]. Using the mathematical framework of the DGG formalism, automated theorem provers [31] could take advantage of the operator algebra to derive new algorithms or mathematically bound errors in the approximate algorithm discussed in Chapter 4, effectively providing a new means of analysis not offered by traditional graph rewriting systems. A definitive source for graph rewriting systems can be found here [95]. While it is very powerful, it takes a dense category theory approach and can be inaccessible to newcomers. However, for the most curious of curious readers, it is highly worth at least browsing.

2.6 Extended Objects and Cell Complex Theory

Declarative modeling of complex systems requires a way to describe non-point-like *extended objects* and the space in which they are embedded [76]. In the context of biological systems, extended objects include polymer networks in the cytoskeleton and multicellular tissues. The embedding space could be the surface or interior of a cell or a whole organism. In this section, a mix of standard and nonstandard definitions is used in the construction of extended objects and cell complexes.

Graphs augmented with labels are expressive mathematical objects capable of a high level of abstraction, and these are used for the representations of extended objects. As defined in

[76], *numbered graphs* are special cases of labeled graphs that have unique consecutive non-negative integer labels for vertices. If the graph in figure 2.1 did not have colors assigned to the nodes, it would be a numbered graph. A *graded graph*, on the other hand, is a graph where vertices are labeled non-uniquely with a level number, associated with a spatial resolution which can only differ by $\{0, \pm 1\}$ between neighbors. A *stratified graph* labels the vertices by a non-negative integer "dimension" of the stratum to which they belong. *Graded stratified graphs* have both dimension and level number vertex labels with suitable constraints.

Continuing as in [76], a special case of stratified graphs is the *abstract cell complex*. The abstract cell complex is a graph that is used to represent the topology of a space in the manner of a CW cell complex [55]. It has further constraints on the dimension labels. A *graded abstract cell complex* can represent the topological properties of the space with the addition of level numbers associated with spatial resolution.

In topology, mappings that preserve all the topological properties of a given space are *homeomorphisms*. They are continuous functions in both directions. A *manifold* is a mathematical object (topological space) that is locally *homeomorphic* to Euclidean space "near" (within an open set including) each point [81] [64]. And so, a *d-manifold* is a topological space that locally looks like d-dimensional Euclidean space [18].

Now, let $k \in \mathbb{Z}^+$, then for any positive integer $k \geq 0$, denote an *open unit k-ball* by $\mathbb{B}^k$ and a *closed unit k-ball* as $\overline{\mathbb{B}^k}$. A mathematical *k-cell* is a topological object that is homeomorphic to $\overline{\mathbb{B}^k}$. Further, the k-cell is defined by all the $k-1$-cells $\ldots$ to 0-cells that compose its boundary. For example, $\partial\mathbb{B}^3$ includes the 2-cells, 1-cells and 0-cells. For any k-cell, ∂ is the operator that returns the k-cell's boundary if it exists.

A *cell complex* $\mathcal{C}$ is a collection of mathematical n-dimensional n-cells, along with all of the lower-dimensional cells that make up their boundaries, and so on iteratively down to the 0-cells. For example, if a cell complex is built to represent the topology of a square as in

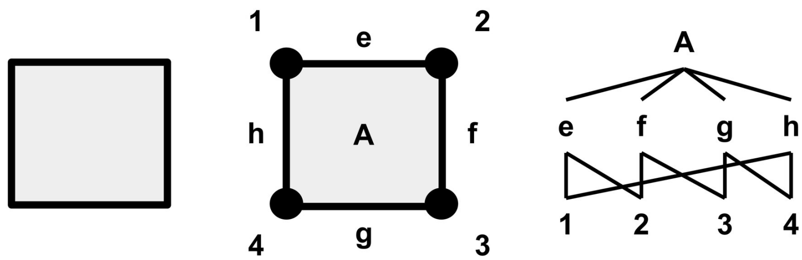


Figure 2.2: Left: A square. Center: The 2-cell, 1-cells, and 0-cells are labeled for the square. Right: The incidence (relationship) graph for the cell complex of a square.

figure 2.2, the square itself is $\{A\}$, the 2-cell. The four 1-cells are the edges, $\{e, f, g, h\}$, and the four 0-cells are corners, $\{1, 2, 3, 4\}$. In addition, the corners are contained in the boundaries of the 2-cells and the boundaries of the 1-cells.

In figure 2.2, we have defined the topology of a square, but we have not provided a geometric interpretation. In [110] they say *topological collection* is a cell complex where values (e.g., they could be real-valued position vectors, an integer label, etc.) are associated with each cell. Take again the example of a square. If 0-cells are assigned points $(x, y) \in \mathbb{R}^2$, 1-cells are mapped to the line segments between two such points, and the 2-cell given a continuous mapping to the region in $\mathbb{R}^2$ associated with the interior of the square, then there is now a *spatial embedding* [59].

In the graph theory section, it was observed that graphs are mathematical objects that can be used to describe relationships. Labeled graphs add additional expressiveness in the form of features requiring "vector labels" i.e. the cross product of feature spaces. Depending on the use case, a graph is itself a cell complex of dimension one with restrictions [47]. On the right in figure 2.2, the incidence graph is depicted for the previous square example. An *incidence graph* represents the boundary relationships between k-cells. Note that the incidence graph in figure 2.2 already has vertices named with unique labels. So, it naturally follows from the notion of a topological collection that a spatially embedded cell complex graph can be defined by providing additional labels, one of which is a spatial embedding. Therefore:

Definition 2.1 (Spatially Embedded Cell Complex Graph)
Let $G_C = (V, E, \alpha)$ be a labeled graph where:

1. *$V \equiv$ k-cells*

2. *$E \equiv$ k-cell boundary relationships*

3. *$\alpha \equiv$ {unique names, k-cell dimension, spatial embedding}, where α is a mapping to a cross product of feature spaces.*

Then, G_C is known as a Spatially Embedded Cell Complex Graph.

Further discussion of the theory of cell complexes can be found in [17, 18, 64, 81, 55].

2.7 Grammars and Languages

In any formal language, grammars describe the syntactical structure of an acceptable sentence and allow for varying levels of expressiveness. Linguists introduced generic grammars as a way to study formal languages [26, 27]. Grammars themselves can be classified into a formal language hierarchy, with one of the most well-known being the Chomsky hierarchy. The hierarchy classifies languages into the categories of recursively enumerable, context-sensitive, context-free, and regular, also known as type 0, 1, 2, and 3. Type 0 is the most expressive and type 3 is the least.

Stochastic grammars are built on top of generative grammars and are grammars that associate probabilities with rules. Another related, but orthogonal formalism, L-systems, was introduced [87] to model biological processes. L-systems represent a different kind of formal language that can translate generated strings into geometric structures; however, they remain deterministic. Stochastic parameterized grammars (SPGs) [80] and dynamic grammars (DGs) [80] were inspired by these notions and lead to the definition of dynamic graph grammars [76]; however, only in SPGs and DGs is there a linear summation $W = \sum_r W_r$ of rule operators W_r to get grammar operator W, in the master equation $P'(t) = W \cdot P(t)$.

Along with grammars, programming languages can be categorized. Computer programming languages are broadly divided into two categories: *declarative* and *imperative*. A declarative programming language describes "what is" the goal to be reached. Some examples of declarative languages are HTML, Haskell, YACC, and Make. An imperative programming language describes "how "the goal is to be reached. Some well-known examples are C++, Python, and Java. However, it is worth noting that the listed examples broadly fall into the mentioned categories, but some also have capabilities to be multi-paradigm as well. The DGG formalism falls into the category of a declarative modeling language, but in this work, its implementation is written in C++, a language that can be used imperatively. Understanding the context justifies the following definition and its subsequent claim:

Definition 2.2 (Declarative Modeling Language [76])
Let L be a declarative modeling language. L is a formal language with:

1. *a compositional map $\Psi : L \longrightarrow S$ that maps all syntactically valid models $M \in L$ into some space S of dynamical systems*

2. *Conditionally valid or conditionally approximate valid families of Abstract Syntax Tree (AST) Transformations*

Given definition 2.2, it can be proposed:

Claim 1 (Model Existence)

Given a Dynamical Graph Grammar modeling language L_{DGG}, and a model $M_{DGG} \in L_{DGG}$, there exists some implementation map ℓ to the imperative supporting language C++, that implements a mapping Ψ, to a valid dynamical system in S.

In this thesis and the following chapters, a mapping is exhibited to support Claim 1. A similar claim for Mathematica is supported by the thesis in which Plenum is presented [129]. The key difference is that the dynamical graph grammar modeling library (DGGML) in this work provides improved scalability due to efficiency in both serial and in principle, parallel.

2.8 Related Work

The dynamical graph grammar modeling library (DGGML) developed for this thesis draws direct inspiration from Plenum [129]. Plenum implements dynamical graph grammars within the Mathematica environment and leverages the power of the built-in symbolic programming capabilities. However, despite its mathematically friendly and expressive interface, Plenum encounters scalability challenges due to its reliance on exact algorithms, making it less suitable for larger and more complex systems. Additionally, while Plenum accommodates graphs indirectly via unique object identifiers (OIDs), it does not treat graphs as native data structures, which also causes performance issues. Related modeling libraries similar to those used in this dissertation include MGS [47], MCell [111], and PyCellerator [105].

MGS is a declarative spatial computing programming language for cell complexes with applications in biology such as neurulation [110], where it is the process by which the neural tube is formed, for example, in vertebrate development. Other works such as [64] by Lane used cell complexes for the developmental modeling of plants and simulating cell division. MCell is used for the simulation of cellular signaling and focuses on the complex 3D subcellular microenvironment in and around living cells. It is designed around the assumption that at small subcellular scales, stochastic behavior dominates. Therefore, MCell uses Monte Carlo algorithms to track the stochastic behavior of discrete molecules in space and time as they diffuse and interact with other molecules. PyCellerator, on the other hand, is a computational framework for modeling biochemical reaction networks using a reaction-like arrow-based input language (a subset of Cellerator [104]). Although these tools share similarities with DGGML and are worth independent exploration, this dissertation concerns the simulation of spatially embedded dynamical graph grammars and applications.

Chapter 3

Dynamical Graph Grammar Formalism

3.1 Introduction

Dynamical Graph Grammars are a further refinement of Dynamical Grammars (DGs) [80], which generalized Stochastic Parameterized Grammars (SPGs) [80] by the inclusion of differential equation rules. SPGs function to unify the formalism of generative grammars, stochastic processes, and dynamic systems. While SPGs can be applied to graphs DGGs include all the related formalisms of SPGs and DGs, along with an additional and expressive modeling language framework for graphs. The semantics of the DGG formalism starts with DGG models M_{DGG} in language L_{DGG} and using a compositional map Ψ_{DGG} maps the declarative grammar rules in the model to a valid dynamical system expressed by a master equation. Consequently, a DGG can be seen as a graph rewriting system, where rewriting events comprise a stochastic process.

3.2 Master Equation

The master equation represents the time evolution of a continuous-time Markov process. It can be written in the form:

$$\frac{d}{dt}P(t) = W \cdot P(t) \ , \tag{3.1}$$

with the equation having the formal (but usually not practical) solution:

$$P(t) = e^{tW} \cdot P(0). \tag{3.2}$$

W is called the model system's "time-evolution operator", since it entirely specifies (in a probabilistic way, which can specialize to deterministic dynamics if need be) how the model evolves in time.

3.3 Operator Process Maps

Let $\Psi(M) = W(M)$ be a semantic map over DGG models comprising rules indexed by r, and $\hat{W}_r \equiv \hat{W}_{LHS_r \to RHS_r}$ be an operator that specifies the non-negative flow of probability between states under each rule r. Then Ψ is "compositional" if it sums the operators W_r over rules thusly:

$$W = \sum_r W_r \tag{3.3a}$$

$$W_r \equiv \hat{W}_r - D_r \tag{3.3b}$$

$$D_r \equiv \mathrm{diag}(\mathbf{1} \cdot \hat{W}_r) \tag{3.3c}$$

where equation (3.3a) states that rule operators sum up to the grammar operator, equation (3.3b) states rules conserve probability, and equation (3.3c) represents the summed conditional probability outflow per state. The operators for rules indexed by $r \in M$ map to the operator sum and the dynamics can be defined under the ME.

3.4 Chemical Reaction Rules

The formalism for DGG rules starts with understanding pure chemical reaction notation and contextualizes what we mean by declarative modeling. Pure reaction rules can themselves be represented in a production rule notation. For example, if we have a pure reaction system, the standard chemical reaction notation [74, 76] would be of the form:

$$\sum_{\alpha=1}^{A_{max}} m_{\alpha}^{(r)} A_{\alpha} \xrightarrow{k_{(r)}} \sum_{\beta=1}^{A_{max}} n_{\beta}^{(r)} A_{\beta} \tag{3.4}$$

Here we use r to represent the r-th reaction channel. We have $m_{\alpha}^{(r)}$ and $n_{\beta}^{(r)}$ as the left and right-hand side nonnegative integer-valued stoichiometries. We do this because we cannot physically have negative chemical reactants. Finally, we let $k_{(r)}$ be the forward reaction rate function.

We can write equation 3.4 in a more compact form [76] using multi-set notation denoted as $\{\cdot\}_*$, where $\cdot$ is a mathematical placeholder and $*$ symbolically indicates multi-set. So:

$$\{m_{\alpha}^{(r)} A_{\alpha}\}_* \xrightarrow{k_{(r)}} \{n_{\beta}^{(r)} A_{\beta}\}_* \tag{3.5}$$

Either way, a simple example of the notation in action is the reaction rule for the formation of water:

$$2H_2 + O_2 \xrightarrow{k_r} 2H_2O \tag{3.6}$$

In equation 3.6 the left-hand side of the equation represents the reactants, consisting of two molecules of hydrogen gas H_2 and one molecule of oxygen gas O_2. The right-hand side represents the products, consisting of two molecules of water H_2O. The coefficients in the equation indicate the stoichiometric ratios of reactants and products involved in the reaction. The reaction occurs with a rate of $k_r > 0$. We can make this more generic and summarize the resulting process as $LHS_r \longrightarrow RHS_r$ for rule r. Consequently, this understanding allows us to encode this formal description into the master equation and derive Gillespie's SSA [48] for a well-mixed system using operator algebra [75].

3.5 Stochastic Parameterized Grammar Rules

Parameterized grammar rules extend the pure reaction rules to an additional parameterized space and allow for a more expressive form of modeling. This gives rise to the SPG. We can also go even further than the Gillespie SSA, and do away with the assumption that we will always have a well-mixed system by adding spatial location parameters $\vec{x}$. The probability space for the SPG was defined in [129]. A form of a stochastic parameterized rule is:

$$\begin{aligned} &\{\tau_{\alpha(p)}[x_p] | p \in L_r\}_* \longrightarrow \{\tau_{\beta(q)}[x_q] | q \in R_r\}_* \\ &\quad \textbf{with} \quad \rho_r([x_p],[y_q]) \end{aligned} \tag{3.7}$$

where $\tau_{\alpha(p)}[x_p]$ and $\tau_{\beta(q)}[x_q]$ are the object types accompanied by parameters x_p and x_q. Note that x_p and x_q may be vectors. Again, r is the rule index, and L_r, R_r are the left and right-hand side argument list indexed sets. So, p and q represent the position of $\tau_{\alpha(p)}[x_p]$ and $\tau_{\beta(q)}[x_q]$ in their respective argument lists. Finally, in the *with* clause $\rho_r([x_p],[y_q])$ is the reaction rate function of both the incoming and outgoing parameters. $\rho_r([x_p],[y_q]) \longrightarrow \mathbb{R}^+$ is a non-negative propensity rate function. If $\rho_r([x_p],[y_q])$ is integrable over output parameters it can be decomposed into a rate function over input parameters and a conditional probability over the output parameters:

$$
\begin{aligned}
&\rho_r([x_p]) \equiv \int \rho_r([x_p],[y_q])\,\Delta[y_q] \\
&P([y_q] \mid [x_p]) \equiv \frac{\rho_r([x_p],[y_q])}{\rho_r([x_p])} \\
&\rho_r([x_p],[y_q]) \equiv \rho_r([x_p]) * P([y_q] \mid [x_q])
\end{aligned}
\tag{3.8}
$$

For clarity, grammar rules will generally be written decomposed in this manner.

An example of a parameterized rule is cell division:

$$
\begin{aligned}
&\mathrm{Cell}_1[x_1,R_1] \longrightarrow \{\mathrm{Cell}_1[x_1',R_1'],\mathrm{Cell}_2[x_2,R_2]\} \\
&\quad \textbf{with} \quad \rho([x_1,R_1,x_1',R_1',x_2,R_2]) \ .
\end{aligned}
\tag{3.9}
$$

On the left-hand side, a single cell is parameterized by position x, where x_i is the position for cell i. It has a radius of R, where R_i is the radius of cell i. Upon selection of a firing rule, indicative of a reaction event, two cells are produced on the right-hand side replacing the one on the left. The unfactorized propensity function, dependent on input parameters of the left and right, weights the likelihood of the reaction occurring. Specifically, it can be defined such that cell division only occurs when the radius R_1 of the initial cell surpasses a predetermined threshold.

Effectively, parameterized rules are encoded into the master equation and we can derive a simulation algorithm [75]. We can elevate these parameterized rules to include graphs by adding unique discrete object IDs [130] as parameters.

3.6 Dynamical Grammars Rules

Using operator algebra we can derive an exact time warping simulation algorithm [75] (Algorithm 1), and add in differential equation rules:

$$\begin{aligned} &\{\tau_{\alpha(p)}[x_p] | p \in L_r = R_r\}_* \longrightarrow \{\tau_{\beta(q)}[x_q] | q \in R_r = L_r\}_* \\ &\quad \textbf{solving} \quad \{\frac{dx_{p,j}}{dt} = v_{p,j}([x_k]) | p, j\} \ . \end{aligned} \tag{3.10}$$

Here, everything remains the same regarding notation, except the left-hand side and right-hand side do not change in number or object type, but the parameters can evolve by solving a differential equation in the new *solving* clause. When we combine these differential rules with the parameterized rules, we get Dynamical Grammars [85].

An example of a dynamic grammar rule is a cell growth rule:

$$\begin{aligned} &\text{Cell}_1[x_1, r_1] \longrightarrow \text{Cell}_1[x_1, R_1] \\ &\quad \textbf{solving} \quad \frac{dR_1}{dt} = k \ . \end{aligned} \tag{3.11}$$

Similar to equation 3.9, a single cell is parameterized by position x on the left-hand side and radius R. Instead of two new cells being produced, a firing of this rule consists of an update to the parameter space by solving an ordinary differential equation. In this case, the equation indicates the radius grows at a constant rate of k as long as the cell exists.

3.7 Dynamical Graph Grammar Rules

Dynamical grammars [130] include all the previous formalism and can handle graph representations, for example by using unique object ID (OID) parameters. However, using OIDs to represent graphs decreases readability and natural expressiveness. Alternatively, using a formal graph notation [76], DGGs can be represented using a different form:

$$G\langle\!\langle \lambda \rangle\!\rangle \longrightarrow G'\langle\!\langle \lambda' \rangle\!\rangle \quad \textbf{with } \rho_r \ \textit{or} \ \textbf{solving } \dot{x} = v. \tag{3.12}$$

Here $G\langle\!\langle \lambda \rangle\!\rangle$ is the left-hand side labeled graph with label vector λ and $G'\langle\!\langle \lambda' \rangle\!\rangle$ is the right-hand side labeled graph with label vector λ'. G and G' without their label vectors λ and λ' are numbered graphs so that the assignment of label component λ_i to graph node member i is unambiguously specified. We have the usual *solving* and *with* clauses.

The following is a simplified example of a stochastic dynamic graph grammar rule from the cortical microtubule array (CMA) grammar discussed further in Appendix A:

Stochastic Retraction:

$$\begin{aligned}
&(\,\blacksquare_1 - \bigcirc_2 - \bigcirc_3)\,\langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3) \rangle\!\rangle \\
&\longrightarrow (\,\blacksquare_1 - \bigcirc_3)\,\langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), \varnothing, (\boldsymbol{x}_3, \boldsymbol{u}_3) \rangle\!\rangle \\
&\qquad \textbf{with } \ H(\cdot)
\end{aligned} \tag{3.13}$$

Equation 3.13 represents a discrete rule used to shorten a microtubule (MT). Specifically within the context of the CMA DGG (more details in Chapter 8 and Appendix A), this rule removes the middle open circle node (node 2) and its corresponding edges. A new edge is created between node 1 and 3. When this rule is combined with the deterministic retraction rule in equation A.3 we get the effect of retraction. A Heaviside function, denoted as H with a placeholder input, signifies the instantaneous propensity activation once a threshold is crossed. One choice for input in H is to compute the distance between the filled square (node 1) and the middle open circle (node 2) and subtract a threshold.

The following is another stochastic rule taken from a morphodynamic grammar for dendritic spine heads [62]:

Actin Arp2/3 Branching:

$$
\begin{aligned}
&\left(\bigcirc_1 - \bigcirc_2 - \bigcirc_3\right)\ \langle\!\langle(\boldsymbol{x}_1,\boldsymbol{\theta}_1),(\boldsymbol{x}_2,\boldsymbol{\theta}_2),(\boldsymbol{x}_3,\boldsymbol{\theta}_3)\rangle\!\rangle\\
&\longrightarrow \begin{pmatrix}\bigcirc_1 - \bigcirc_2 - \bigcirc_3 \\ \qquad\qquad\quad \diagdown \\ \qquad\qquad\qquad\quad \bullet_4\end{pmatrix}\ \langle\!\langle(\boldsymbol{x}_1,\boldsymbol{\theta}_1),(\boldsymbol{x}_2,\boldsymbol{\theta}_2),(\boldsymbol{x}_3,\boldsymbol{\theta}_3),(\boldsymbol{x}_4,\boldsymbol{\theta}_4)\rangle\!\rangle\\
&\quad \textbf{with}\ \ k_{\text{branch}}\ N_{(\text{Arp, free})}\\
&\quad \textbf{where}\ \begin{cases}\text{branchSide} \sim \mathcal{U}(\pm 1)\\[1ex] \theta_{\text{branch}} \sim \mathcal{N}(\frac{70}{180}\pi \times \text{branchSide}, \sigma_\theta)\\[1ex] \boldsymbol{x}_4 = \boldsymbol{x}_2 + \begin{bmatrix}\cos(\theta_{branch}) & \sin(\theta_{branch})\\ -\sin(\theta_{branch}) & \cos(\theta_{branch})\end{bmatrix}(\boldsymbol{x}_2 - \boldsymbol{x}_1)\end{cases}
\end{aligned}
\tag{3.14}
$$

The rule in equation 3.14 is a branching rule for an actin polymer within the spinehead. Effectively, based on the branching factor k_{branch} and the Arp-related factor, the polymer graph depicted in the LHS can branch. The graph with a branch is produced on the RHS. The branching angle is sampled from a normal distribution, and the resulting newly created graph node is rotated and set a distance away. The rule in equation 3.14 provides an example of how the **where** clause, not shown in the purely destructive retraction rule of equation 3.13, works.

Switching back to the CMA grammar (discussed further in Appendix A and B), the following rule is a simplified example of a dynamic graph grammar ordinary differential equation rule used to model retraction of the depolymerizing end of a microtubule:

Deterministic Retraction:

$$
\begin{aligned}
&(\blacksquare_1 - \bigcirc_2)\ \langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2)\rangle\!\rangle\\
&\longrightarrow (\blacksquare_1 - \bigcirc_2)\ \langle\!\langle(\boldsymbol{x}_1 + d\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2)\rangle\!\rangle\\
&\quad \textbf{solving}\ \ d\boldsymbol{x}_1/dt = \hat{\rho}_{\text{retract}}(\cdot)\boldsymbol{u}_1
\end{aligned}
\tag{3.15}
$$

The rule in equation 3.15 is a deterministic *solving* rule used to shorten an MT. In the context of the CMA DGG, this rule exclusively updates the position parameter, x_1, associated with the filled square (node 1). The function $\hat{\rho}_{\text{retract}}$ can be any function and directly determines the growth rate along the unit vector direction. Collectively, a set of deterministic and stochastic rules r, where each is defined over a probability density function, constitutes the grammar for a model M. For additional examples of graph grammar rules, please refer to Appendix A and B.

Chapter 4

Simulation Algorithms

4.1 Introduction

In this chapter, a brief description of the exact hybrid parameterized ODE SSA-like algorithm is provided, along with suggestions for the appropriate context to use it. A parallel version of it is also proposed and briefly discussed. This is followed by the core of this chapter, which is about the approximate algorithm. An overview of how to construct the approximate algorithm and the pieces required are detailed. Additionally, different strategies for parallel implementations are discussed.

4.2 Exact Hybrid Parameterized ODE SSA

The exact algorithm simulates a single trajectory of a continuous time stochastic process. Propensity functions are factored into a product of the rate function and a distribution of output parameters conditioned on input parameters, as in equation 3.8. While the simulation time is less than the maximum, we sample the time until the next reaction and modify the system when such a reaction occurs. The process is very similar to the standard SSA but with the inclusion of parameter update sampling and ODE solving, including the time-warping equation. The warp equation is an ODE to keep track of the time until the next event and must be solved as part of the system of ODEs governing the time evolution of the parameters. When a reaction does occur, the state of the system is modified according to the rule instance selected and the parameters are sampled from the conditional distribution of the factored propensity function.

Algorithm 1: Exact Hybrid Parameterized SSA/ODE Algorithm [75]

---

factor $\rho_r([x_p],[y_q]) = \rho_r([x_p]) * P([y_q] \,|\, [x_p])$;
while $t \leq t_{max}$ **do**
  initialize SSA propensities as $\rho_r([x_p])$;
  initialize $\rho^{(total)} := \sum_r \rho_r([x_p])$;
  initialize $\tau := 0$;
  draw effective waiting time τ_{max} from $\exp(-\tau_{max})$;
  while $\tau < \tau_{max}$ **do**
    solve ODE system, plus an extra ODE updating τ;
      $\frac{d\tau}{dt} = \rho^{(total)}(t)$;
  draw reaction r from distribution $\rho_r([x_p])/\rho^{(total)}$;
  draw $[y_q]$ from $P([y_q] \,|\, [x_p])$ and *execute* reaction r;

The exact algorithm accommodates the expressive nature of DGGs, but it exhibits diminished performance when applied to large systems. Plenum [129], in particular, is slowed even further by a symbolic implementation entirely written in a computer algebra system. In the development of Plenum, the internal implementation of the exact algorithm and internal algorithmic design considerations were made to fit within the framework of Mathematica, effectively trading performance for expressiveness. Grammar rules were defined symbolically with parameterized objects using object IDs (OIDs). To find matches of left-hand side (LHS) grammar rules, Plenum made use of Mathematica's pattern matching capabilities to find LHS rule matches by considering the LHS pattern as a tuple of object combinations. The matching tuple is generated by searching a pool of objects and eliminating invalid matches by using constraints and constraint-solving [32, 33]. Plenum handled discrete events and continuous rules by integrating rate functions and solving differential equations using Mathematica's numerical ODE solver. In this context, the implementation of Algorithm 1 is realized in an expressive and symbolic framework. In contrast, DGGML is less expressive, but it realizes the exact algorithm with far less overhead and with native graph support.

In terms of performance, for small non-spatial systems, the exact algorithm is a fitting and straightforward solution. Moreover, the exact algorithm effectively extends to ensemble simulations of non-spatial systems, where simulations may be executed concurrently. This approach is useful in broad applications, such as conducting parameter sweeps and computing the statistical properties, including mean and variance, across numerous runs.

4.3 Parallel Exact Hybrid Parameterized ODE SSA

When the system in question is sufficiently complex, the exact algorithm can be sped up by introducing parallelism at key points, as demonstrated in Algorithm 2. While reactions (rules) still must be fired in order to maintain accuracy, a parallel solution can effectively leverage modern multi-core architectures, significantly expediting the simulation of a single trajectory. This optimization not only improves computational efficiency but also increases the algorithm's scalability, particularly when confronted with medium-sized systems where fewer simulations and massive ensembles are not required.

In Algorithm 2, there are three distinct points of parallelization. The first point is the propensity computation. Every iteration each propensity for each matching rule instance found in the system must be computed. While there are methods [129] to reduce how often the propensities are recomputed, computing them can not be entirely avoided. A simple solution to speed up this bottleneck is introducing a parallel for loop. Since computing each propensity is independent of any other and the parameters are read-only operations, the

Algorithm 2: Parallel Exact Hybrid Parameterized SSA/ODE Algorithm

factor $\rho_r([x_p],[y_q]) = \rho_r([x_p]) * P([y_q] \,|\, [x_p])$;
while $t \le t_{max}$ **do**
  ParFor *initialize* SSA propensities as $\rho_r([x_p])$;
  ParReduce *initialize* $\rho^{(total)} := \sum_r \rho_r([x_p])$;
  initialize $\tau := 0$;
  draw effective waiting time τ_{max} from $\exp(-\tau_{max})$;
  while $\tau < \tau_{max}$ **do**
    ParFor **solve** ODE system, plus an extra ODE updating τ;
      $\frac{d\tau}{dt} = \rho^{(total)}(t)$;
  draw reaction r from distribution $\rho_r([x_p])/\rho^{(total)}$;
  draw $[y_q]$ from $P([y_q] \,|\, [x_p])$ and *execute* reaction r;

operation is well suited for parallelization.

The second point of parallelization is the computation of the total sum of propensities. In serial computing, the computational complexity of the sum would be $\mathcal{O}(n)$, where n is the number of propensities to the sum. The parallel construct analog to the summation is a parallel reduce, which can be run $\mathcal{O}(\frac{n}{p} + \log n)$, where n is the same, and p is the number of processors.

The third point concerns the performance of Ordinary Differential Equation (ODE) systems, which represents a critical computational task within many scientific simulations. The efficacy of parallelization strategies is dependent upon the specific characteristics of the system under consideration. However, the sequential solving of sizable ODE systems serves as a notable bottleneck, and warrants optimization efforts. A pragmatic approach to mitigate this bottleneck is to leverage solvers built with parallel capabilities [58, 2]. Such solvers offer promising routes for improving computational efficiency and scaling computational resources effectively in the context of ODE system solving.

Algorithm 2 becomes useful when testing more complex grammars or when quicker results and analysis of a single trajectory are appropriate. It is important to note though, that parallel algorithms have substantially more overhead (e.g. thread creation) compared to serial algorithms. This overhead is particularly noticeable in multi-CPU compute nodes where maintaining cache coherency imposes additional computational costs. Generally, the effectiveness of this method should be seen when the number of propensities/ODEs exceeds several orders of magnitude and/or the propensity/ODE calculations exhibit considerable complexity. Finally, depending on the available resources, the ODE solving process could potentially be offloaded to a GPU accelerator for further performance gains. In the case of GPU offloading the problem size, throughput, and latency should be carefully considered.

In DGGML, which is discussed in further detail in Chapters 5 through 7, Algorithm 4 from section 4.4 is used; however, Algorithm 2 can be implemented within an expanded cell in Algorithm 4, with the constraint that all expanded cells are processed in serial. The system would need to be decomposed into expanded cells that are large enough to maximize the multi-core performance of the hardware, but small enough so that the domain can be decomposed into expanded cells that will run in a reasonable amount of time. In the case where no domain decomposition exists in the approximate algorithm, we have exactly the exact parallel algorithm.

Within DGGML, the aforementioned points of parallelization can be made within the algorithm run for an individual geocell. Since DGGML already makes use of SUNDIALS [58], a future upgrade to enable parallelization on the CPU or GPU is built in [46]. The primary consideration before implementing this algorithm is to assess if the problem size justifies parallelization benefits over running a serial version of Algorithm 4 in ensemble mode.

4.4 Approximating the Exact Hybrid Parameterized ODE SSA

The foregoing exact algorithm is powerful and works for multiple rules of different forms [75]; however, it is prohibitively slow for large systems. A single run of the exact algorithm yields only one trajectory. In practice, thousands or more may be needed to be run to compute meaningful statistics or to recover outcome density functions. Algorithm 1 will quickly become computationally intractable, since it must compute the next event to occur based on every potential rule firing in the system. To address this problem we introduce Algorithm 3 (the original approximate hybrid parameterized ODE SSA) and Algorithm 4 (the improved approximate hybrid parameterized ODE SSA) below.

We make two key assumptions in our approximation of the exact algorithm: spatial locality of the rules, and well-separatedness of cells of the same dimensionality within the cell complex used to decompose the simulation space into domains. Consider the spatial locality constraint for graphs. The system state comprises extended objects taking the form of labeled graphs. Each of the nodes in a graph is labeled with a vector-valued position parameter. Additional parameters are allowed that have no spatial constraint. Our graph grammar rules are made spatially local by virtue of their propensity functions. We use spatial locality to define local neighborhoods of rule firings. Any rule instantiations outside this neighborhood have zero or near-zero propensity that decreases rapidly, for example exponentially, with distance. Hence any two objects in the system that are too far apart have a very small chance of reacting, and their potential interactions can be ignored at a small approximate error cost.

Spatial locality also allows us to decompose the domain of the simulation space into smaller, well-separated geometric cells. In the context of the simulation algorithm, a "cell" refers to a computational spatial domain, which differs from the biological notion of a cell. Such a geometric cell or "geocell" is a cell of an expanded cell complex (ECC), labeled by the dimension of the corresponding (geo)-cell in the unexpanded cell complex. Lower dimensional cells of the cell complex are expanded in their perpendicular directions to be wide enough to keep rule instances from spanning multiple same-dimensional geocells. An example can be seen in figure 5.6. By setting these geocells to be wide enough (at least several factors larger than the exponential "fall off distance"), we can logically map rule instances to well-separated geocells.

The operator W in equation 3.1 assumes a state space and specifies the probability flow on that space for all of the extended objects in our system. Considering equation (3.3a), $W = \sum W_r$, the method we propose to approximate e^{tW} is an operator splitting algorithm that imposes a domain decomposition by means of an expanded cell complex that corresponds to summing operators, $W = \sum_{(d)} W_{(d)} = \sum_{(d,c)} W_{(c,d)}$, over pre-expansion dimensions d, and cells c of each dimension:

$$e^{tW} \approx \left(\prod_{d\downarrow} e^{\frac{t}{n} W_{(d)}} \right)^{n\to\infty} \tag{4.1a}$$

$$e^{t'W_{(d)}} = \prod_{c\subset d} e^{t'W_{(c,d)}} \quad \text{where} \quad \left[W_{(c,d)}, W_{(c',d)}\right] \approx 0 \quad \text{and} \quad t' \equiv \frac{t}{n} \tag{4.1b}$$

$$W_{(c,d)} = \sum_r W_{r,c} \equiv \sum_r \sum_{\left\{ \substack{R \mid \varphi(R)=c, \\ R \text{ instantiates } r} \right\}} W_r(\, R \mid c, d) \tag{4.1c}$$

Sub-equation (4.1a) is a first-order operator splitting, by solution phases of fixed cell dimension, where $d\downarrow$ means we multiply from right to left in order of highest dimension to lowest. It incurs an approximate error of $\mathcal{O}((t/n)^2)$. Sub-equation (4.1b) is an even stronger refinement of sub-equation (4.1a) because it uses the fact that the resulting cells c of fixed dimension d are all well-separated geometrically with enough margin (due to the expanded regions of dimension $d' \neq d$ [89]) so that rule (reaction) instances R, and R' commute to high accuracy if they are assigned to different cells c, c' of the same dimensionality, by some rule (reaction) instance allocation function φ. The commutators of equation (4.1b) can be calculated as derived in [77], but they will inherit the product of two exponential falloffs with separation that we assumed for the rule propensities (c.f. [77], equation 12 therein), which is an even faster exponential falloff. Hence the dynamics $e^{tW_{(c,d)}}$ of different cells c, c' of the same original dimension d and can be simulated in any order, or in parallel, at little cost in accuracy.

The operator splitting and the function φ introduce a major opportunity for parallel computing because the exponentials $e^{tW_{(c,d)}}$ defined in each cell c of a given dimensionality d can all be sampled independently of one another. This potential parallelism includes the possibly heavy computation of solving ODEs specific to cell c. Sub-equation (4.1c) then defines the geocell-specific operator for the process to be simulated by Algorithm 1 [75], specialized to the case of graphs. The resulting parallel algorithm is outlined in Algorithm 3.

Algorithm 3: Original Approximate Spatially Embedded Hybrid Parameterized SSA/ODE Algorithm

```
while t ≤ t_max do
    foreach dimension d ∈ {D_max, D_max − 1, . . . , 0} do
        using function φ map rule instances to the geocells of the expanded cell
          complex;
        ParFor expanded geocell c_i ∈ ExpandedCellComplex(d) do
            run Exact Hybrid Parameterized SSA/ODE algorithm for Δt in c_i;
    t += Δt;
```

Algorithm 3 is the original version of the approximate algorithm and is ran in serial (no ParFor) for the model discussed in Chapter 8 [73]. On the highest level, it specifies that φ must be used to map reactions to geocells and those have the potential to be processed in parallel and fully encapsulates sub-equations (4.1b) and (4.1c). However, it leaves the details of how to keep the state of rule matches in the simulated system consistent across geocells after a rule fires and rewrite occurs unspecified.

Algorithm 4, which is the algorithm used in DGGML and the model in Chapter 9, improves upon the original by filling in the details on how to keep the state of rule matches in the simulated system consistent across geocells after a rule fires and rewrite occurs. To keep a consistent state of rule matches, we must specify how rule matches are represented. The *match data structure*, which is initialized at the beginning of Algorithm 4, is a data structure composed of all possible rule instances matching the grammar rules and all possible corresponding left-hand side (LHS) connected component instances. The match data structure is then the representation of all current rule instances and must be updated every time a rule fires and the system graph is rewritten. To make this work we take advantage of an incremental updating procedure (Chapter 5, section 5.6) within a geocell, which allows us not to have to recompute all new matches after distant events occur in the system.

Algorithm 4: Improved Approximate Spatially Embedded Hybrid Parameterized SSA/ODE Algorithm

```
initialize the match data structure with all rule instances;
while t_global ≤ t_max do
    foreach dimension d ∈ {D_max, D_max − 1, ..., 0} do
        using function φ map rule instances to the geocells of the expanded cell
          complex;
        ParFor expanded geocell c_i ∈ ExpandedCellComplex(d) do
            t_local = t_global;
            factor ρ_r([x_p], [y_q]) = ρ_r([x_p]) * P([y_q] | [x_p]);
            while t_local ≤ t_global + Δt_local do
                initialize SSA propensities as ρ_r([x_p]);
                initialize ρ^(total) := Σ_r ρ_r([x_p]);
                initialize τ := 0;
                draw effective waiting time τ_max from exp(−τ_max);
                while τ < τ_max and t_local ≤ t_global + Δt_local do
                    solve ODE system, plus an extra ODE updating τ;
                      dτ/dt_local = ρ^(total)(t_local);
                draw rule instance r from distribution ρ_r([x_p])/ρ^(total);
                draw [y_q] from P([y_q] | [x_p]) and execute rule instance r;
                incrementally update match data structure;
        synchronize and remove invalid rule instances from data structure of matches;
    recompute rule level matches (as needed);
    t += Δt_global;
```

Another detail filled in with Algorithm 4, is the point of synchronization and rule invalidation that comes after processing all expanded cells of a given dimension. In DGGML, after φ maps all rules to a cell c_i of dimension d, the exact algorithm running within c_i is responsible for incrementally updating its state and keeping track of any rule invalidation that may share objects with rules belonging to neighboring cells of differing dimensions. Hence, the need for the *synchronization* point in Algorithm 4. Synchronizing for consistency is a consequence of the commutator approximation in equation (4.1b). When running in serial the synchronization point still exists, and functions as a logical point in the algorithm for inserting any custom code for correcting and creating a consistent state of rule matches and potentially measuring any errors. Algorithm 4 also has the potential to be made parallel in a lock-free way thanks to the synchronization point because it leaves resolution of the global state until after the fact, making it a huge improvement over Algorithm 3.

Algorithm 4 also greatly improves upon Algorithm 3 by introducing the aformentioned match data structure with an incremental update process to ensure that distant rule firings do not require the matches to be recomputed and that full system matches are only ever recomputed as needed. The primary need for recomputation even after synchronization is because spatially embedded and spatially local graph grammar rules are composed of distinct connected components (*motifs*) as discussed in Chapter 7 section 7.2. By virtue of a **solving** rule, these components may move in or out of the propensity "fall-off" distance. While these connected components themselves can always be incrementally updated i.e. the set of connected component matches is *fully online*, the rules instances that are comprised of them can only be incrementally updated for a short period of time. As a result, they need to be recomputed as needed, making the match data structure *semi-online*.

For further details of the incremental update and invalidation procedures see Chapter 5 and for the implementation details of Algorithm 4 in DGGML see Chapter 7. Additionally, it can be seen that without domain subdivisions, Algorithm 3 reduces to the exact algorithm (Algorithm 1) and Algorithm 4 reduces to a version of the exact algorithm but with an incremental update. A more complete mathematical treatment of the approximate algorithm using DGG commutators computed as in [77] to bound operator splitting errors is a topic for future work. Algorithms 3 and 4 are also specifically designed for spatially embedded graphs, but there are situations where they could be used in other non-spatial instances if we had a way to reasonably measure locality in the parameter space and that is also another topic for future work. For a more detailed derivation of the theory behind the approximate algorithms, please refer to Appendix C. For DGGML Algorithm 4 is used, but we present Algorithm 3 as an intermediate step in defining the algorithm used in DGGML.

4.5 Phi Function

The process by which rules get mapped to a particular dimensional cell in Algorithm 4 hinges on the choice of the φ function (Appendix C). This function plays a pivotal role in establishing a suitable mapping from reactions (rules) to geocells, ensuring that each reaction corresponds precisely to one geocell. This approach guarantees complete accountability for all rules. Because φ assigns a geometric cell to every match, φ serves to partition the set of matches. Various implementation choices exist for defining φ, each presenting distinct trade-offs, although we will focus on two primary approaches to demonstrate how they work within DGGML.

For the remainder of the section, assume that the following is a match of the left-hand side of a graph grammar rule, and the associated parameters and unique IDs are as follows:

$$(\blacksquare_{10} \longrightarrow \bigcirc_{11} \longrightarrow \bigcirc_{12})\langle\!\langle(\boldsymbol{x}_{10},\boldsymbol{u}_{10}),(\boldsymbol{x}_{11},\boldsymbol{u}_{11}),(\boldsymbol{x}_{12},\boldsymbol{u}_{12})\rangle\!\rangle$$

Here, the match has a closed square type with integer ID 10 and two open circle types with integer IDs 11 and 12. Each of the graph grammar rules has an associated position vector

x and a unit vector u. For this section, only the position parameter is used. For further details on the semantics of DGG rules, please refer to Chapter 3. Now, let $\mathrm{c}_{\mathrm{id}}(x_i)$ be a function that returns the unique cell id of which an object at position x_i belongs and let the function $\dim(\mathrm{c}_{\mathrm{id}}(x_i))$ be a function that returns the dimension of the respective cell. For example, assume that $\dim(\mathrm{c}_{\mathrm{id}}(x_{10})) = 2$, $\dim(\mathrm{c}_{\mathrm{id}}(x_{11})) = 1$, and $\dim(\mathrm{c}_{\mathrm{id}}(x_{12})) = 1$. Also, assume $\mathrm{c}_{\mathrm{id}}(x_{10}) = 4$, $\mathrm{c}_{\mathrm{id}}(x_{11}) = 5$, and $\mathrm{c}_{\mathrm{id}}(x_{12}) = 5$. Therefore, x_{10} belongs to a cell 4 of dimension 2 and the other nodes belong to cell 5 of dimension 1. Also, keep in mind that the spatial-locality and well-separated constraints ensure that all matched objects in the rule will not belong to more than one cell of the same dimension. Thus, restricted to a given match, the $\dim(\cdot)$ function has an inverse $\dim^{-1}(\cdot)$.

The first approach we discuss uses the single-point anchored φ function, which involves selecting an "anchor node" from the left-hand side and determining the geocell containing its spatial coordinates. The selection process may be randomized or guided by a user-defined heuristic such as a root of a spanning tree. Using the section example, we could pick the root to be node 10. In this way, we simply have $\varphi(x_{10}) = \mathrm{c}_{\mathrm{id}}(x_{10}) = 4$, which is a cell of dimension 2. Hence, a single-point anchored mapping function maps node 10 to cell 4. This method boosts computational efficiency by necessitating only a single point for lookup. However, a trade-off arises wherein automorphisms of the same rule match may be assigned to different dimensions. When an invalidation occurs, each geocell containing the invalidate rule must be updated. Invalidations are discussed further in Chapter 5.

Alternatively, the minimum dimensions function φ can be defined to map the rule match to the minimum dimension of the geocells to which each node maps. Using the section example, it would be $\varphi(x_{10}, x_{11}, x_{12}) = \dim^{-1}(\min[\dim(c_{id}(x_{10})), \dim(c_{id}(x_{11})), \dim(c_{id}(x_{12}))]) = 5$, which is a cell of dimension $\min[\dim(\dots)] = 1$ where $\dim^{-1}$ exists by well-separatedness. Additionally, this function can be further extended to consider all rules in which any node of the current rule participates, subsequently determining the dimensions of the corresponding geocells as discussed in Appendix C. In either scenario, each instance of the φ function calculates the minimum dimension for any match and assigns the rule to the corresponding geocell. Leveraging the well-separated property ensures that no node in the rule or its associated rules can be assigned to multiple geocells of identical dimensions. The minimum dimension φ ensures that all automorphisms are consistently assigned to the same dimension because the min function returns the same result, regardless of the ordering of the input. Computationally, the cost of using the minimum dimension φ means checking the spatial coordinates of each node on the left-hand side and possibly all the nodes of the rules it participates in.

4.6 Conclusion

The original simulation algorithm in [75] can be sped up by processing the reactions out-of-order at the cost of accuracy. The approximate hybrid ODE SSA is an operator splitting algorithm that imposes a domain decomposition by means of an expanded cell complex that corresponds to summing operators, over pre-expansion dimensions d, and without loss of accuracy over cells c of each dimension. This algorithm is specifically designed for spatially embedded graphs, but there are situations in which it could be used in other non-spatial instances if we had a way to reasonably measure locality in the parameter space. Two key assumptions were made in our approximation of the exact algorithm: spatial locality of the rules, and well-separatedness of the cell complex used to decompose the simulation space into domains. Furthermore, we proposed parallelization points for Algorithm 1 in Algorithm 2 and proposed a clear parallelization point for Algorithm 4. Also, we proposed an incremental match data structure composed of all rule instances and all possible LHS connected component instances, greatly reducing the need to recompute matches after every rule firing event. In addition, the choices of the φ functions were briefly discussed. In

Appendix C we provide more work on the algorithm and justification that can lead to a formal proof of an error bound in future work. In this thesis, the performance difference between using the exact and approximate algorithm is demonstrated in Chapter 8.

Chapter 5

Overview and Building Blocks of DGGML

5.1 Overview of the Design

The initial implementation of the approximate simulation algorithm was specific to a plant cell cortical microtubule array model [73] and not generalized to work with different model types. Since then it has been reworked into the modeling library DGGML, to enable reuse and generic construction of simulations for different grammars. Further, the modeling library is written in C++17 with minimal dependencies, making it portable and accessible. A conceptual overview of the library is shown in figure 5.1.

In figure 5.1, the library is structured into seven distinct categories: core, grammar, geometry and topology, state monitoring, differential equations, utility, and dependency. The core group contains essential functionality required for simulating a grammar. Within the grammar category are components necessary to define a grammar as a model and to initialize data structures for interaction with the core components. The geometry and topology com-

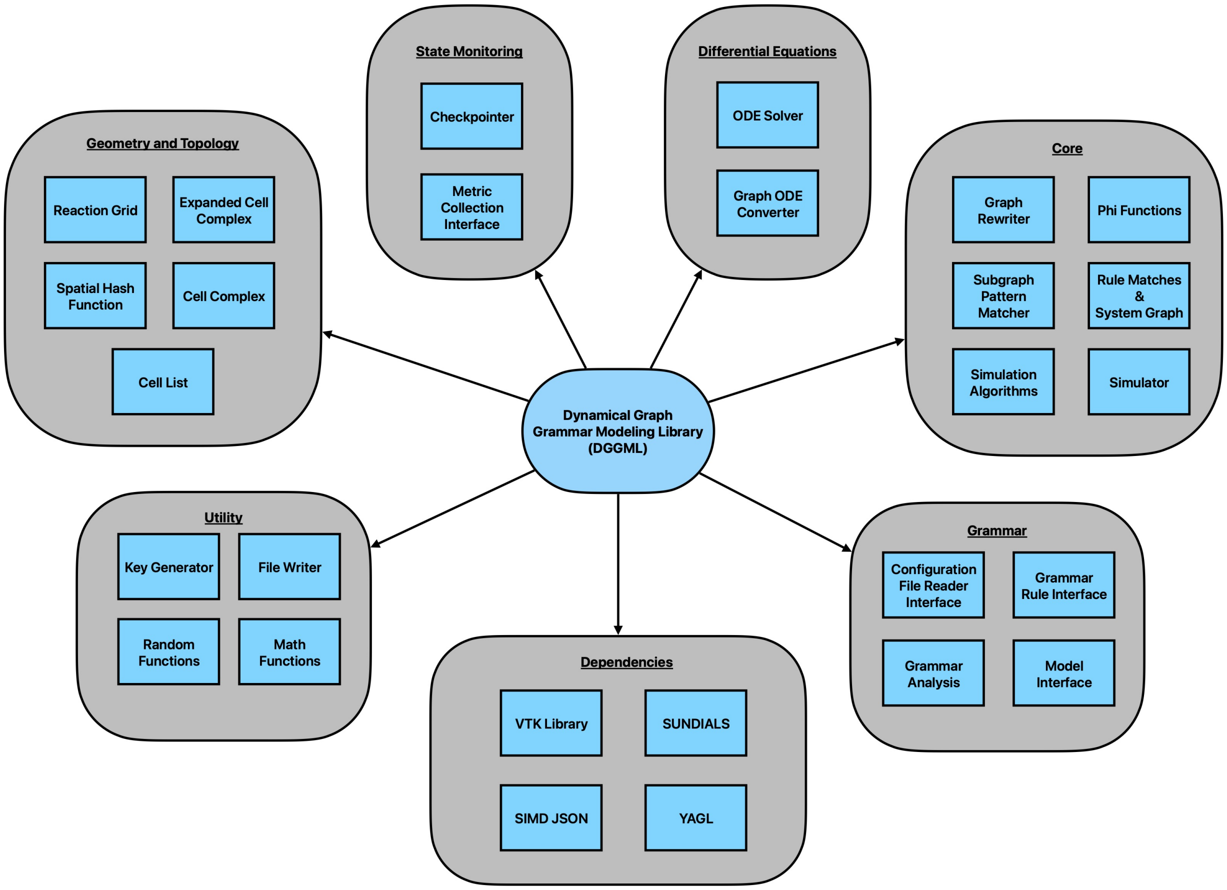


Figure 5.1: Conceptual overview of library design and organization. The library organization is grouped by components falling into the core, grammar, geometry and topology, state monitoring, differential equations, utility, and dependency categories. The hierarchy of the grouping is visualized as a central blue squircle (square with rounded corners) designating the library itself surrounded by exterior grey squircles, each representing a group of components, where the black directional arrows depict compositional relationships. Within these exterior squircles, blue rectangular boxes represent individual components. We have acronyms ODE (ordinary differential equations), VTK (visualization toolkit), SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers), SIMD JSON (single input, multiple data JavaScript object notation), and YAGL (yet another graph library).

ponents facilitate spatial organization, partitioning, and search queries. State monitoring components track and collect information on simulation progress. While differential equations could be grouped within the core, their potential for extensibility to accommodate more complex solvers favors a separate categorization. The utility grouping contains components that are used by the core and other categories. Finally, the dependencies are used throughout the library, but are the components that could be replaced by alternatives or a custom version of the type of component could be specifically written and included as part of the library itself. Overall, amongst the seven groups, twenty-seven key components work and interact with each other in different ways. In the remainder of the DGGML chapters, figure 5.1 serves as a conceptual point of reference. The relationships of components within and between groups, along with the necessary details of components, will be exposed through discussions on building blocks, and in later chapters by specifying how to define a grammar, how grammar analysis works, and how the grammar simulation takes place.

Finally, in this thesis, the implementation of the Dynamical Graph Grammar Modeling Library (DGGML) and CajeteCMA (used in Chapter 8) natively supports undirected graph grammar rules. However, other implementations could include native support for directed graphs. In a directed graph implementation, undirected links can be simulated by a pair of directed links in opposite directions. In an undirected graph implementation, directed links can be simulated by replacing each node with a triple of labeled nodes: a front door node for all incoming edges, connected undirectedly to a central node bearing all the original labels, and a back door node for all outgoing edges. Furthermore, the user-defined node type space or label space can be combined with a new 3-valued field for (in, central, out) labels. This dual encoding ensures the flexibility to represent both directed and undirected connections as needed.

5.2 Yet Another Graph Library

The DGG simulator relies on graphs as its core data structure. While the DGG formalism specifies graph rewrite types declaratively, it does not detail how these rewrites should be implemented in code or how graphs should be represented computationally. To address this gap, a dynamic graph library, Yet Another Graph Library (YAGL), was developed. YAGL serves as a versatile, header-only (i.e. can be included in any project without compiling first) library designed to accommodate various graph types. Unlike the commonly used monolithic and powerful Boost Graph Library (BGL) [108], YAGL was designed for its compatibility and flexibility when used in conjunction with modern C++17 features. This deliberate choice allows for greater adaptability and the possibility of future extension without reliance on external dependencies.

Internally, the Yet Another Graph Library (YAGL) graph, as initially described in [73], defines lists for nodes, edges, and adjacency. Nodes and edges are implemented as generic key-value pairs, allowing for versatility in the types of data they can store. This flexibility is achieved through the use of static polymorphism and template meta-programming techniques. In the YAGL implementation, nodes are stored in a C++ unordered map, functioning as a hash table for efficient constant-time lookup. Additionally, the edge list is maintained as an unordered multi-map to enable future support for multiple edges between nodes and to put no constraints on how the stored edges are to be ordered in the list, although this extra generality currently remains unused within the DGGML context. The adjacency list, which can be generated by sorting an edge list or independently constructed as in the case of the current version of YAGL, is for representing the incoming and outgoing edges for the graph. Fundamentally the adjacency list differs from an edge list because it does not store any data that could be associated with an edge. The adjacency list is realized as an unordered map where the key is the node key and the value is a pair of unordered sets of keys. The unordered map allows for constant time lookup of adjacency. The first key set is for outgoing edges, and the second set is for incoming edges. These two sets provide a comprehensive representation of the graph's connectivity and allow for quick lookup and dynamic insertion or removal.

The choice of an unordered map (hash table) for the node list in the graph structure is largely inspired by the library-sort algorithm [13]. A similar design is adopted for the adjacency list, with node data fully decoupled from relationship information. This design results in an average lookup time complexity of $\mathcal{O}(1)$, making it highly efficient, particularly for smaller, sparsely connected graphs containing approximately 10^5 nodes. The design choices for the node list and adjacency list exhibit significant advantages in terms of lookup efficiency and graph rewrite flexibility when compared to alternative representations such as linked lists or inflexible matrix-based implementations of the adjacency list. Additionally, YAGL includes graph rewrite operations such as adding or removing nodes/edges, along with other useful algorithms such as identifying connected components, various versions of depth-first and breadth-first searches, generating rooted spanning trees, determining graph isomorphism, and at a higher level of functionality providing a generalized solution to the injective homomorphism algorithm for subgraph specific pattern recognition.

In the context of DGGML, which leverages spatial graphs, node types in the graph definition provided by YAGL are implemented using a specialized spatial variadic templated variant node type. The data structure for the specialized node type has a field for the position type, which is what provides the specialized node type with the position (spatial) data. The variadic template referred to in the "spatial variadic templated variant node" ensures a variable number of types can be used to define a single node type, hence the terminology variadic. A variant, on the other hand, functions like a sum type. Sum types are a concept from type theory that represents a type that can take on different types. So here, variant means the type can change whereas variadic means the variant can be constructed out of a variable number of types and be changed into any one of them at any time in the program. This feature is used during graph rewrites to change a node on the left-hand side of a grammar rule to a node of a different type on the right-hand side of a grammar rule.

Essentially, graph nodes can be any type built out of any number of types, and will always have a position in simulated space. This design enables the exposure of spatial positions to the algorithm while concealing user-defined values within the variant. Notably, in C++, the 'variant' is a C++17 feature that serves as a mechanism for accommodating values of differing types. It also ensures type safety through compile-time checks and automatically manages memory reallocation when the type is changed.

5.3 Subgraph Specific Pattern Recognizer

A key part of any algorithmic implementation of the DGG formalism detailed in Chapter 3 is finding all the applicable instances of left-hand side (LHS) grammar rule matches in the system graph. Here, the system graph is our search space, and the LHS graph is the target graph. The process of recognizing a single instance of the target graph in the search space is what we mean by *graph pattern recognition*. Finding every valid instance of the target graph is *matching*.

Graph pattern matching algorithms can be categorized as either *exact* or *inexact*, with our focus in this work being exclusively on exact matching algorithms. These algorithms fall into three main categories [22]: tree search [117, 30], constraint programming [118, 33], and graph indexing [90]. The previous DG/DGG implementation by Yosiphon [129] was based on constraint programming for paramaterized objects [32]. Tree search algorithms, such as backtracking via depth-first search (DFS), exhaustively explore the search space to identify all potential solutions. In contrast, constraint programming strategically eliminates non-viable solutions early and may have a different search strategy. Graph indexing, optimized for static graphs, finds utility primarily in graph databases due to its efficiency in data retrieval and manipulation. Although graph pattern matching problems are well-studied, solutions often rely on heuristic approaches due to the NP-completeness of (sub-)graph isomorphism problems, as referenced in [117, 118, 41, 29, 30, 63, 22, 90, 118].

Our approach to addressing pattern recognition makes the design choice to do recognition by (sub)-graph isomorphism, and we implement a hybrid heuristic [96] tree search method with backtracking and constraints. Backtracking, often implemented recursively within a tree structure, gradually constructs the solution set while exploring the state space. It offers the flexibility to also be formulated iteratively, leveraging the known structure of the search tree for efficient exploration. Representing the state space also as a tree, each branch signifies a partial solution path, and every leaf node represents a variable to be matched. While a manually generated nested search loop was used iteratively in [73], our generalized approach in DGGML incorporates the recursive strategy for versatility. However, an iterative solution could have been used and the following discussion primarily focuses on the iterative handwritten version that runs in polynomial time, since the same logic applies to a generalized recursive version.

Expanding the discussion, we integrate the search process into a subgraph-specific pattern recognizer (SSPR), tasked with identifying all matching left-hand sides (LHS) of each grammar rule. A recognizer identifies labeled subgraphs that match one-to-one with a given LHS-labeled graph. While the SSPR in [73] includes a specific search code for each LHS grammar rule in the model and its respective connected components patterns, the subgraph pattern recognizer (SPR) in DGGML dynamically handles patterns generated from the user-defined grammar analysis (Chapter 7, section 7.2) at runtime. When the SPR in DGGML is ran on a specific pattern, it functions as an SSPR.

To develop an SSPR, we must determine methods to identify all relevant matches of a given LHS grammar rule in the system graph, denoted as G_{SYS}. The LHS of any rule is represented as G_{LHS}, and the system graph serves as our search space, with the left-hand side pattern as our target. *Subgraph pattern recognition* refers to recognizing a single instance of the target graph in the search space, while *matching* involves finding every valid instance of the target graph G_{LHS}. Chapter 7 elaborates on this concept during the grammar analysis phase, focusing on finding matches of connected components and utilizing those matches to identify patterns created from combinations of the connected component matches.

Since it would be expensive to directly search for all possible functions $f : G_{LHS} \longrightarrow G_{SYS}$, where f is an injective graph homomorphism, we need to apply some heuristic filter. For simplicity, assume G_{LHS} is a single connected component in this case. If there were more than one, each LHS would have multiple rooted spanning trees, and we would need to apply this process for each of them. This is specifically what we refer to as multicomponent matching, and address this with the "cell list" in Chapter 7. Let G_T be a rooted spanning tree (RST) of target graph G_{LHS} and let h be its inclusion map into into G_{LHS}. If G_T and f exist, then there must exist some injective graph homomorphism $g : G_T \longrightarrow G_{SYS}$. We can represent the entire process using the commutative diagram of injective graph homomorphisms as in equation 5.1:

$$\begin{tikzcd} G_{LHS} \arrow[r, "f"] & G_{SYS} \\ G_T \arrow[u, hook, "\subseteq", "h"'] \arrow[ur, hook, "g"'] & \end{tikzcd} \tag{5.1}$$

We can demonstrate what we mean by finding the mappings f and g. In figure 5.2 we have a visualization of the graph to be searched and in figure (5.3a) example of a target pattern. Figure (5.2) is G_{SYS} and figure (5.3a) is G_{LHS} also known as the target graph/pattern.

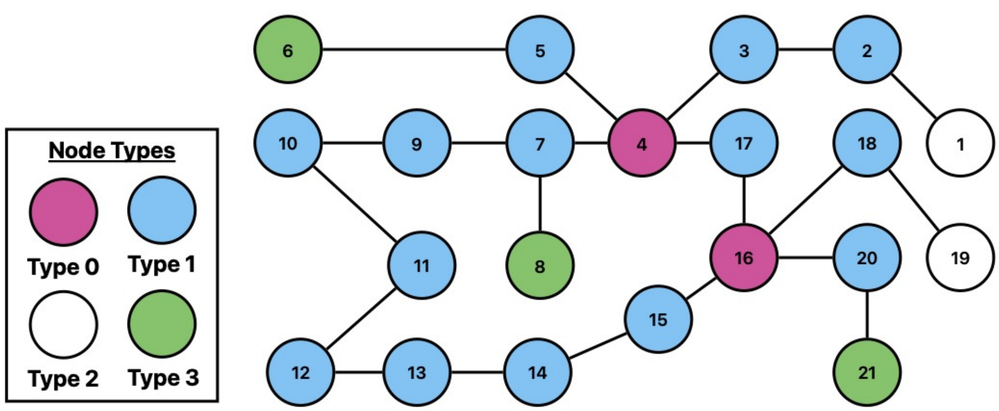
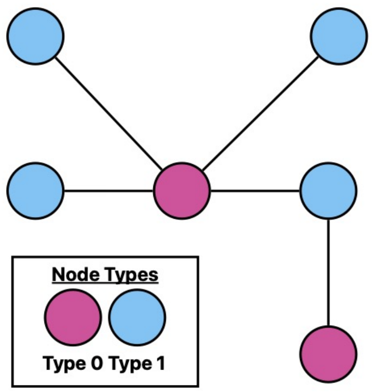
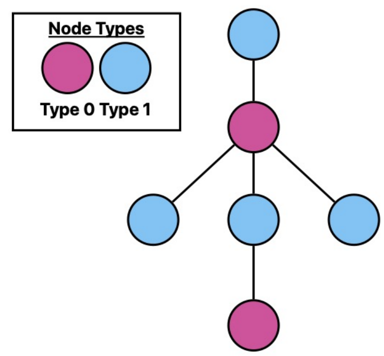


Figure 5.2: The graph to be searched.

Instead of trying to find the direct match for the target graph, we first find all matches g for some rooted spanning tree $G_T \subseteq G_{LHS}$, and then filter to "lift" g to f, if possible. In the process of filtering all the relevant additional constraints are applied. We can see an example of a valid rooted spanning tree in figure (5.3b). The height of the tree directly impacts how deep we need to search the system graph for valid patterns.



(a) The graph target pattern to find.



(b) Tree transformation of the pattern graph in figure (5.3a) with a tree height of 3.

Figure 5.3: A side-by-side view of the target graph to find one of its transformations.

In figure 5.4 we can see what an algorithm searching the graph using the transformation of the target pattern seen in figure (5.3b) does when we pick the starting node (node 5) as depicted and start our rooted search. The rooted tree directly corresponds to our search path. The shorter the tree, the less deeply we need to search and the more search branches that can be pruned. Starting on node five in the search graph as seen in figure 5.2 we search and find two matches as can be seen in figure 5.4. A search started from every node in the target graph would yield all possible matches. In this case, there are twelve matches (we include all valid permutations of a target match).

We can see the pseudo-code in Algorithm 5. For this particular graph, we have asymptotic complexity on the order of $O(N^6)$. However, that is for the worst case and without filtering. For a sparse graph, we expect it to be more likely on the order of $O(N^2)$. Using the logic

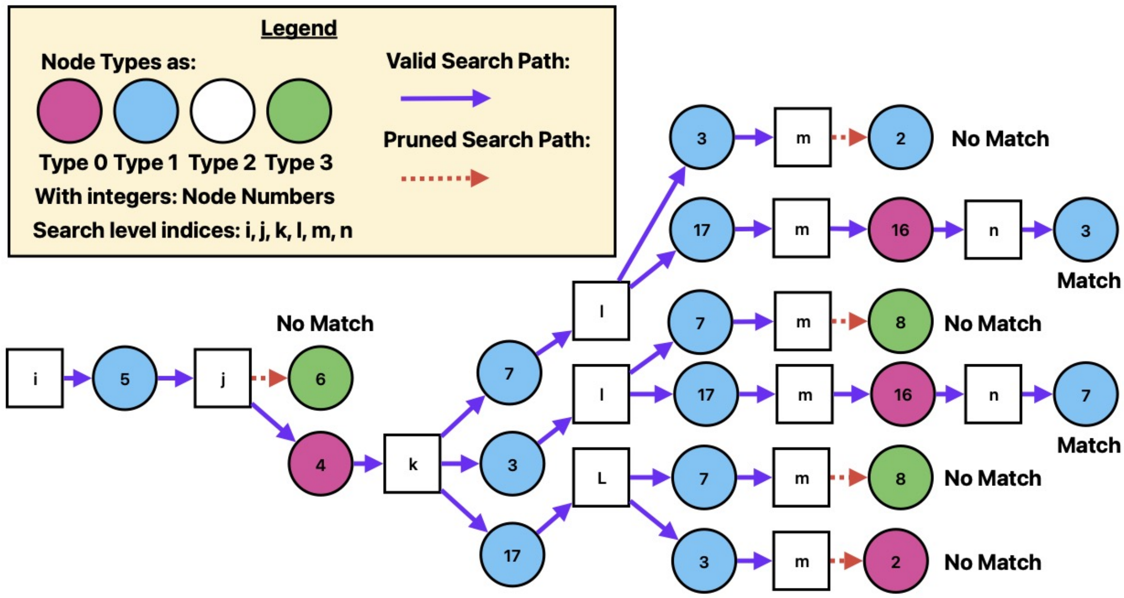


Figure 5.4: Example of how a search graph works to find an optimal path.

for an iterative SSPR, it is possible to write a generic solution recursively.

The generalized version implemented in DGGML directly creates a runtime equivalent [119] of Algorithm 5. Within DGGML, the SSPR algorithm takes user-constructed graphs, generates a rooted spanning tree for each of its connected components, and conducts a search for the component match, ensuring an identical systematic exploration of the search space as in Algorithm 5. This approach to SSPR design not only enables efficient pattern recognition but also sets the stage for potential parallelization in both specific and generalized cases.

For example, one strategy for parallelizing the SSPR is to parallelize over the outer loop of graph nodes for a search pattern. This parallelization process can be replicated for each left-hand side (LHS) graph to be searched. By distributing the workload across multiple processes, parallelization can significantly reduce the overall search time for large graphs. A further optimization could be to combine searches for multiple graphs into one by merging their RSTs and setting additional ways to reach acceptance states.

Algorithm 5: Heuristic Pattern Matching Example for the Pattern Graph in figure 5.3b

```
match = {∅};
for i ∈ G(V) do
    if λ_i = type 1 then
        continue;
    for j ∈ nbrs(i) do
        if λ_j = type 0 then
            continue;
        for k ∈ nbrs(j) ∖ {i} do
            if λ_k = type 1 then
                continue;
            for l ∈ nbrs(j) ∖ {i, k} do
                if λ_l = type 1 then
                    continue;
                for m ∈ nbrs(l) ∖ {j} do
                    if λ_m = type 0 then
                        continue;
                    for n ∈ nbrs(j) ∖ {i, k, l} do
                        if λ_n = type 1 then
                            continue;
                        match.append(i, j, k, l, m, n);
```

In terms of optimization, certain LHS graph rules might share isomorphic connected components and based on this observation, a more efficient approach to utilize the SSPR is discussed in the grammar analysis section (Chapter 7, section 7.2). Given DGGML's focus on spatially embedded graphs, connected component matches of an LHS can be treated as separate nearby graphs to be searched for. This observation eliminates the need for graph rules with multiple connected components to search extensively for each component in order. In Chapter 7, the simulation section introduces a "cell list" as a data structure to expedite this search process.

The SSPR plays a crucial role in efficiently identifying relevant matches of LHS grammar rules within the system graph in DGGML. By using a generalized algorithm based on the iterative approach to dynamically search for specific patterns generated from the user-defined grammar analysis (Chapter 7, section 7.2) at runtime, the SSPR offers powerful graph pattern recognition and matching capabilities. Making use of a heuristic tree search method with backtracking and constraints to address the NP-complete (sub-)graph isomorphism problem, the SSPR optimizes search efficiency while maintaining flexibility for complex matching scenarios. The rooted spanning tree generation is an effective strategy for clarifying computational complexity, which is highlighted by the handwritten pattern matching solution. Additionally, the SSPR is compatible with adding additional search strategies to improve performance, which is particularly beneficial for spatially embedded graphs. The SSPR discussion includes useful theoretical concepts and practical strategies for implementation, resulting in the effective graph pattern recognition and matching capabilities found within DGGML.

5.4 The Expanded Cell Complex

Before expanding a cell complex (defined in Chapter 2, section 2.6), it must first be generated. An outline of the cell complex generation process is shown in figure 5.5. Beginning with simulation space (in the context of DGGML rectangular) the initial step involves applying any amount of subdivision. Subsequently, a regular lattice grid graph is constructed with twice the original subdivision, which is then labeled. The doubling ensures that there will be enough nodes for each dimension and that the nodes will be in the geometric center of their respective dimensional cell. The labeling of nodes within this new graph depends on the chosen subdivision, with labels assigned according to respective dimensions i.e. 2D for interior, 1D for edges, and 0D for vertices. This labeled regular lattice graph, serving as the cell complex, contains all the necessary information for expansion. In terms of implementation, the cell complex is represented using a labeled graph in YAGL with a future possibility of making the cell complex dynamic under a meta-DGG.

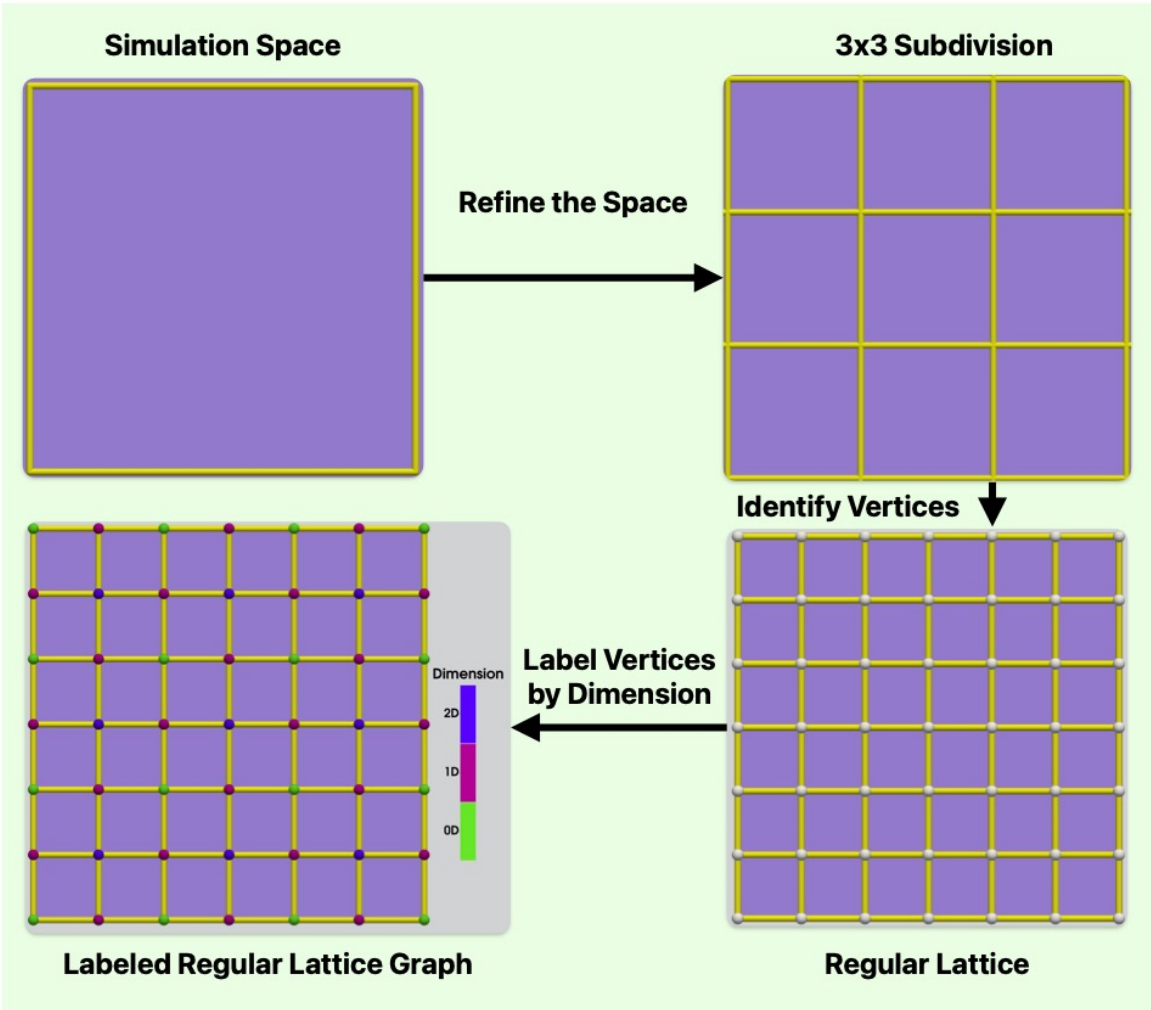
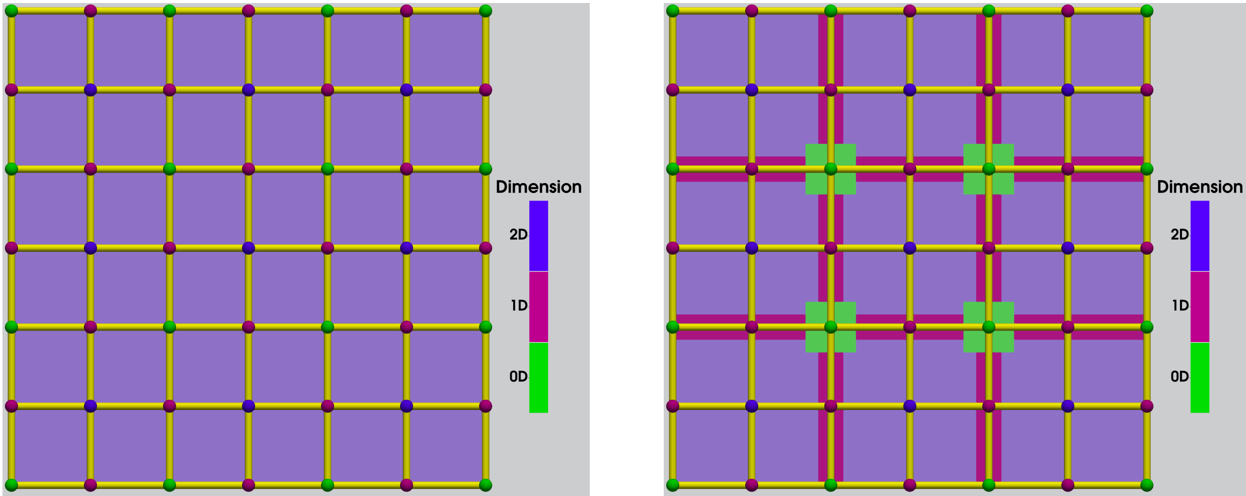


Figure 5.5: Cell complex generation and labeling. We start with a square domain, and then subdivide it into a 3×3 grid. From the grid, we generate a regular lattice graph representing the topology of the subdivided space. The regular lattice graph is then labeled by the dimension it belongs to resulting in the cell complex being labeled by dimension.

The mathematical idea behind the "*expanding*" of a cell complex is a process of consistently mapping each lower dimensional cell in a cell complex to a cell of the highest dimension in a new "expanded" cell complex. By expanding as in [89, 40] all of the abstract cell complex cells with dimension numbers less than the maximum, we get an *expanded cell complex* (ECC). We only apply expansion to the interior of lower dimensional cells. In our case,



(a) The pre-expansion cell complex.

(b) The expanded cell complex.

Figure 5.6: The pre-expansion of the cell complex generated in figure 5.5 and its expansion into well-separated lower dimensional cells.

we apply expansion to a two-level graded abstract cell complex with a coarse-scale 2D grid that is refined to a finer-scale 2D grid. Figure 5.6 exhibits a side-by-side visualization of a pre-expansion (5.6a) cell complex and its post-expansion (5.6b) cell complex.

In figure (5.6b), lower dimension interior cells are expanded such that they always have a "collar" width more than that of the cell of the dimension above, so that cells of the same dimension are always separated by at least a dimension-specific minimum distance. This key criterion of the ECC is *well-separatedness*, which means expanded cells of the same dimension do not even come close to overlapping. The "collar" we refer to is related to the idea of a tubular neighborhood in differential topology [60]. The ECC is implemented as a graph data structure in DGGML. Fundamentally it inherits from the cell complex and uses YAGL as the underlying graph representation.

5.5 Graph Transformation

Graph transformations are a fundamental part of rewriting systems, and within a category-theory approach there are two algebraic category-theoretic constructions: single-pushout (SPO) [72] and double-pushout (DPO) universal diagrams [38]. SPO focuses on local transformations, directly replacing a subgraph in the original graph with another subgraph. In contrast, DPO has more flexibility, allowing for the replacement subgraph to be embedded in a larger context. To accomplish this, DPO requires two additional constraints to ensure uniqueness and keep transformations well-defined. The dangling condition, the first constraint in DPO, requires all edges incident to a deleted node to be deleted as well. The second constraint, the identification condition, ensures that each element set for deletion has only one copy in the left-hand side subgraph. The combination of the two conditions is the gluing condition, which allows for a consistent and unambiguous transformation.

Alternatively, graph transformations can be described and given a formal semantics using the operator algebra in the mathematical framework for DGGs. The following equation from [76] demonstrates this fact:

$$
\begin{aligned}
\hat{W}_r \propto \rho_r(\lambda, \lambda') \sum_{\langle i_1 \ldots i_k \rangle_{\neq}} & \Bigg[\Big(\prod_{p \in B_r} \prod_{i \neq i_q | \forall q \in \overline{B}_r p} E_{i_p\, i} \Big) \Big(\prod_{p \in C_r} \prod_{i \neq i_q | \forall q \in \overline{C}_r p} E_{i\, i_p} \Big) \Bigg] \\
\times & \Bigg[\Big(\prod_{p', q' \in \mathrm{rhs}(r)} \Big(\hat{a}_{i_{p'} i_{q'}} \Big)^{g'_{p'q'}} \Bigg] \Bigg[\Big(\prod_{p' \in \mathrm{rhs}(r)} \Big(\hat{a}_{i_{p'} \lambda'_{p'}} \Big)^{h_{p'}} \Bigg] \\
\times & \Bigg[\Big(\prod_{p, q \in \mathrm{lhs}(r)} \Big(a_{i_p i_q} \Big)^{g_{pq}} \Bigg] \Bigg[\Big(\prod_{p \in \mathrm{lhs}(r)} \Big(a_{i_p \lambda_p} \Big)^{h_p} \Bigg].
\end{aligned}
\tag{5.2}
$$

Equation 5.2 is a generalized form of graph rule semantics that allows for the elimination of hanging edges as part of the grammar's operator algebra. When reading from right to left, all vertices and vertex labels of the LHS graph are annihilated (destroyed) in an arbitrary order and then all edges are annihilated in any order, then all vertices and vertex labels of the RHS graph are created in any order then all edges of the RHS are created in any order. The final operators on the first line are the dangling edge erasure operators that enforce that edges must connect active vertices. In this way, the erasure operators presented in this context ensure that hanging edges are removed. For a more detailed explanation please see [76].

In DGGML, the methodology used to perform graph rewrites follows the semantics presented in the operator algebra and works similarly to the DPO approach. The logic takes advantage of sets for the structural aspect of rewrites in the library. To clarify this, first let $r : G_{LHS} \longrightarrow G_{RHS}$ be the production of a grammar rule r. For shorthand $G_{LHS} = L$ and $G_{RHS} = R$. The vertices of the L are then $V(L)$ and the vertices of R are then $V(R)$. The edges of L are $E(L)$ and the edges of R are $E(R)$.

To determine how to structurally transform the L into R we need four sets, arising from the creation of new vertices and edges and the annihilation with restoration of no longer valid vertices and edges. Let K be the creation graph, and D be the destruction graph. The set of vertices to create consists of the vertices found in the right-hand side vertex set, but not in the left. Hence, $V(K) = V(R) \setminus V(L)$, i.e. the set difference. The edges to be created are then, $E(K) = E(R) \setminus V(R)$. For the creation graph, we simply have $K = R \setminus L$. The set of vertices and edges to destroy is the opposite of the vertex and edge sets of the creation graph. So, vertices and edges in the left-hand side that are not found in the right-hand side must be removed. Hence, the destruction graph is $D = L \setminus R$. Once we have the creation and destruction sets for a particular rule type r, we can take the node/edge numberings of any injective homomorphism of L denoted as L_i found in the system graph, G_{SYS}, and build corresponding sets K_i and D_i. Here the new vertices and edges in K_i are created using the key generator mentioned in figure 5.1. To get the new state of vertices in the system graph, G'_{SYS} we can use the following equation:

$$V(G'_{SYS}) = (V(G_{SYS}) \setminus V(D_i)) \cup V(K_i) = (V(G_{SYS}) \setminus (V(L_i) \setminus V(R_i))) \cup (V(R_i) \setminus V(L_i)) \tag{5.3}$$

When removing nodes, YAGL in DGGML cleans up hanging edges automatically. Finally, it should be noted that L and R are technically labeled graphs i.e. $G = (V, E, \alpha)$ where α is the labeling function. So, the labels of L are $\alpha(V(L))$ and the labels of R are $\alpha(V(R))$. However, in the context of the library, YAGL and in turn DGGML store the label information as data within the graph node itself, meaning the label is a type in the context of C++. So, the final step after building the creation and destruction sets for the vertex and edges sets and structurally transforming the graph is to set the new label data for any newly created nodes by copying the types from K to K_i, effectively equating to a relabelling. In DGGML, this is realized by switching all the newly created nodes to the correct type in the specialized spatial variadic templated variant node described in section 5.2 when discussing YAGL. In line with the DGG formalism, after the structural rewrite and type change of the rewrite occurs, the values the label data take on will be populated by the user-defined update function (described in Chapter 6, section 6.4) for the rule.

5.6 Incremental Update

A critical design decision in an efficient algorithmic implementation of the DGG formalism in DGGML concerns the state of the system following a graph rewrite event. Once a rule is selected to fire and fires, the system's state must be updated. The most straightforward approach would involve recomputing all matches in the system, which is clearly resource intensive. The following discussion presents an alternative solution to incrementally update the matches pertaining to connected components of the LHS graphs, instead.

Before proceeding, it is useful to clarify some terminology. In the context of DGGML, matches of the LHS graph for a rule r are stored hierarchically. We store them in this way because the LHS graph of a grammar rule may be a graph with more than one connected component. This is what we mean by a *multicomponent* rule/graph. From the grammar, there may be several rules that have connected components that when taken as subgraphs are isomorphic to one another. This common set of connected component patterns is what we mean by *motifs* for a grammar. For more details on *motifs* and grammar analysis in please see Chapter 7 section 7.2.

The *component match set* (represented as a map data structure) of all matches (injective homomorphisms) of these motifs. Since LHS grammar rules can be multicomponent, combinations of the matches in the component set are what form the *rule match set* (also represented as a map data structure). Together, the *component match set* and the *rule match set* form what we mean by the *match data structure* (introduced in Chapter 4, section 4.3). To accelerate the search for new rule matches, components matches are also sorted into a *cell list* i.e. a data structure allowing for fast queries of components matches nearby in space. For more details on the cell list in action see Chapter 7 section 7.3.

The goal of the incremental update is to keep the match data structure up to date following a graph rewrite. By using the incremental update, the connected component match set can be kept fully online, since connected component matches are invariant to motion. The rule match set, on the other hand, is incrementally updated within a geocell, but is periodically recomputed (Chapter 4, section 4.3) after all dimensional geocells have run. The recomputation occurs because the connected components of the rule matches move with respect to one another and may eventually go out of or come into the "fall-off" distance for the propensity function. So, we can say that the match data structure is *semi-online*.

Next, it is important to clarify the impact of certain rewrite operations on the system graph. Adding a node or an edge invariably triggers the SSPR, as it opens up new search paths and the possibility for more patterns to be found and added to the component and rule match sets. Conversely, removing a node or an edge solely invalidates, without initiating a search, owing to the graph rewrite semantics. In the context of DGGML, altering the node type is identical to adding a new node in terms of match invalidation, whereas modifying parameters such as position results in a passive rewrite. These passive rewrites affect multi-component spatial pattern matches and are addressed periodically by recomputation of all multicomponent matches globally (Chapter 7). As rewrites always involve a combination of these operations, there exists a systematic approach to processing them.

Following a rewrite event, the incremental update process involves the following steps: *component invalidation*, *rule match invalidation*, *component matching*, *component validation*, *rule matching*, and *rule match validation*. The ordering is similar to reading equation 5.2 from right to left, where nodes/edges are destroyed and then created. However, the incremental update differs from the operator equation being an implementation rather than semantics, because it is cast in the context of the data structures and internal implementation of DGGML. These details should be thought of as the mapping laid forth in Claim 1.

Initially, *component invalidation* requires searching the component set for all connected component match instances of any components containing the nodes and edges marked for removal from the graph rewrite discussed in the previous section, and removing those components. Due to the well-separated nature of geocells (Chapter 4 and Appendix C) and the localized execution of rewrites on the matches homomorphic to a subgraph of the spatially embedded system graph, concurrent processing of geocells of the same dimensions is possible. By using a suitable data structure for the set of component instances and optimizing the function φ of section 4.5, component matches can be safely removed in parallel or in a serialized manner by locating them in the component match data structure.

Rule match invalidation presents a more complicated challenge. Because rules are directly mapped to the geocell, only those rules mapped to the current geocell via φ can be invalidated. Given that the current geocell lacks the state of rules mapped to other geocells, it must temporarily store the list of invalidated components for future potential invalidation of rules owned by nearby geocells, which are of different dimension and therefor not being simulated at the same time. An optimization would be to only track the invalidated components near the boundary of the geocell. The future invalidation of rules occurs during the *synchronization* point in Algorithm 4. Also, for both component and rule match invalidation, spatial locality can be leveraged to improve the search efficiency for components containing an invalid node/edge or rules containing an invalid component. Using a grid of cells and a list of components found in that cell, neighboring cells can be quickly queried. So, instead of searching everywhere for a potentially invalidated component, the search is only performed within a local region. This method is leveraged within DGGML and is discussed more in Chapter 7.

In *component matching*, once all the invalidations of components are processed, new component matches must be found. A simple and slightly greedy way to accomplish this is to precompute the height of the tallest rooted spanning tree generated from all left-hand side motifs in the common set of components for the grammar. For each of the newly created nodes, we perform a depth-first search to the depth of the precomputed height and add newly visited nodes to a visited set. This does not need to be done for any removed node, since removal will only generate invalidations. Using the visited set, a subgraph $S_{search} \in G_{sys}$ is induced. The SSPR is then run on subgraph S_{search} to find all new component matches. The search is specifically designed to only accept matches and add them into the component match data structure if they contain a new node or newly added edge from the creation set generated by graph rewrite - this is what is meant by *component validation*. Once the components are validated, a spatial hash function is applied to map them to their respective cells in the cell list.

For *rule matching* the set of cells in the cell list new components were inserted into, and their collective neighbors, constitute a new search space S_{nbr}. This time the search space is not an induced subgraph to find component matches, but instead comprises all component matches in nearby cells to be combined to find new rule matches. To accomplish this, we search for all combinations of component matches that match the component(s) in the LHS of a rule. To ensure that only new rule matches are identified, matches involving newly validated components are exclusively considered and the others are rejected - this is what is meant by *rule match validation*. Additionally, φ is applied to determine the geocell to which the new rule matches are mapped. If the geocell is the current one, the rule matches are inserted into the rule list. Otherwise, they are set aside and inserted into the correct geocell during the *synchronization* phase of Algorithm 4, provided they still exist.

While alternative approaches exist for this incremental update procedure, the algorithmic idea proposed squarely addresses the problem as a graph algorithm. Moreover, given the assumption of a system significantly larger than the local search space, new matches can be identified and updated with considerably less overhead when compared to having to search the space globally. An alternative approach involves storing an entire copy of the full search space in the form of completed and partial matches, resulting in a complex hierarchy of search trees. While potentially faster than the approach employed in DGGML, this method would necessitate more storage.

5.7 Differential Equation Solving

To solve the ODEs for deterministic grammar rules, a state-of-the-art numerical solver, ARKODE [92, 91] is used. It offers adaptive-step time integration modules for stiff, nonstiff and mixed stiff/nonstiff systems of ordinary differential equations (ODEs). ARKODE is part of the suite of nonlinear and differential/algebraic equation solvers (SUNDIALS) [58, 46].

The solvers in the library offer well-tested methods for handling complex systems of equations efficiently and have the functionality to be run on GPUs [10]. In DGGML, ARKODE is wrapped into an interface class that allows **solving** functions to be implemented as lambda functions. In a scenario where DGGs are implemented as a language rather than a library, a transpiler could directly generate these functions from the DGG language.

Before the use of SUNDIALS, numerical finite difference schemes, including forward Euler and Runge-Kutta methods [52], were used directly. Direct implementation of these methods provides alternative approaches to using heavyweight ODE solvers and offers the advantage of simplicity. When choosing which ODE solving method to use, there are trade-offs. Forward Euler, for instance, is a simple and intuitive method that is quick to code in the absence of SUNDIALS and works as a placeholder until more advanced methods can be added. However, it lacks accuracy. On the other hand, Runge-Kutta methods, such as the classic fourth-order Runge-Kutta (RK4), are more accurate and versatile. RK4, in particular, uses a weighted average of different slope estimates to improve accuracy, making it suitable for a wide range of ODE problems. But, it can be more computationally intensive compared to simpler methods like forward Euler.

An advantage of using SUNDIALS is its built-in root-solving capabilities, which are surprisingly lacking in many solvers, and are essential for solving the time-warping equation in all of the algorithms presented in Chapter 4. For example, ODEINT [2], another alternative solver available as part of the Boost libraries does not have built-in root solving. In cases like ours where root finding is necessary in order to find the time at which ODE solving should yield to discrete event firing, having this capability readily available can significantly reduce the amount of code required to get a solution. For an example of what ODE libraries should strive for, the Julia differential equation library [88] is an excellent source.

For the case when root solving is required and SUNDIALS is not used, several methods are available to implement root-solving algorithms. Newton's method and the bisection method are two commonly used techniques for finding roots of equations. These methods involve iteratively refining estimates of the root until a desired level of accuracy is reached.

While not included in DGGML, DGGs can include reaction-diffusion equations [129]. In this case the Crank-Nicolson method [112] is a standard choice, offering stability properties. Another approach that could be used is Mimetic methods [23]. In the context of reaction-diffusion equations, mimetic methods offer several advantages. More importantly, they maintain local and global conservation properties, ensuring that mass, momentum, or other quantities are conserved accurately over time. This is particularly important in applications where conservation laws play a critical role.

Chapter 6

Defining a Grammar in DGGML

6.1 Introduction

This chapter provides general guidelines for creating a grammar and demonstrates how these guidelines are applied in DGGML through interactions with library interfaces and how they inform modeling choices. The primary user-facing interface components discussed include the model interface, grammar rule interface, configuration file reader interface, and metric collection interface (figure 5.1). Further, in the context of DGGML, a model represents the user's code-defined implementation of a grammar. These guidelines are also broadly applicable to any instantiation of a DGG simulator.

6.2 Building a Model

Understanding how to define grammars and build models in DGGML is essential for creating the connection (Claim 1) between implementation and the DGG formalism. There are six recommended steps for defining a grammar and building a model: (1) identify the initial conditions, (2) define a set of structure changing rules, (3) determine rate functions and differential equations, (4) define the simulation domain's geometry and topology, (5) set boundary conditions, and (6) determine time scale and other parameter settings. Steps (2) and (3) are arguably the central ones of the DGG formalism, because they encapsulate the **solving** and **with** clauses. All of the information from these steps can be defined in a derived subclass from the modeling interface in figure 6.1.

```
template<typename GraphGrammarType>
class Model {
public:
    NameType name;
    GraphGrammarType gamma;

    using GraphType = typename GraphGrammarType::graph_type;
    using KeyType = typename GraphGrammarType::key_type;

    KeyGenerator<KeyType> generator;
    GraphType system_graph;
    ExpandedComplex2D<> geoplex2D;
    SettingsType settings;

    virtual void initialize() = 0;
    virtual void checkpoint(unsigned int);
    virtual void collect(unsigned int) {};
    virtual void print_metrics(unsigned int) {};
};
```

Figure 6.1: A simplified example of the abstract base class for the modeling interface, templated based on the type of graph grammar provided. The GraphGrammarType defines the grammar (gamma) and provides type information for essential data structures: the key generator and system graph. The expanded cell complex is necessary for simulation space generation. The main functions include an initialize function, which must be defined, and optional check-pointing, metric collection, and metric printing functions that take in the current step as input.

Figure 6.1 presents a simplified overview outlining the required components and functions of a DGGML model. A notable component is the key generator, supplied for initializing the system graph. Its role is to guarantee consistent key generation throughout the simulation. This ensures that user-created keys do not conflict with those generated during simulation runs. Users must also subclass from the model class and define an initialization function but otherwise have freedom for customization. Ultimately, the model is passed into the simulator interface object provided by DGGML, with user-specific details. Example code illustrating the main program structure is available in figure 6.2. This design choice offers flexibility in model creation and execution while ensuring DGGML remains open for future extensions. For example, because the library does require how the settings in figure 6.1 must be set, there does not have to be a parser or user-defined parameter setting function in figure 6.2. The settings could be set internally in the model upon its construction.

```
int main()
{
    ParserType parser;
    DocType settings = parser.iterate(load("settings.json"));
    UserModel model;
    model.set_parameters(settings);
    SimulatorInterface<UserModel> simulator;
    simulator.setModel(model);
    simulator.simulate();
}
```

Figure 6.2: An example program for simulating a grammar. Initially, a parser is constructed, and a configuration file is parsed. Subsequently, a user-defined model is created and its parameter settings are configured. A simulation is then instantiated based on the type of model provided by the user. Once the model is set, the simulator executes the simulation.

A configuration file reader interface is provided by default, utilizing ‘simdjson” [65] for parsing JavaScript Object Notation (JSON) format files. These files contain custom input parameters and experimental settings that may change between program runs without requiring recompilation. The library facilitates parsing and enables users to initialize settings for their models. The JSON file’s contents are entirely customizable by the user, as the parser solely handles parsing and delivers parsed results. These settings can then be stored in any suitable data structure. An example of a configuration file for a cortical microtubule array grammar is shown in figure 6.3.

```
{
    "META": {
    "EXPERIMENT": "experiment name"
    },
    "SETTINGS": {
      "TOTAL_TIME": "simulation time units",
      "CELL_NX": "number of cells in x direction",
      "CELL_NY": "number of cells in y direction",
      "CELL_DX": "cell size in x direction",
      "CELL_DY": "cell size in y direction",
      "GHOSTED": "does the simulation have a halo boundary",
      "NUM_MT": "number of microtubules in the system",
      "MT_MIN_SEGMENT_INIT": "minimum initial size of microtubules",
      "MT_MAX_SEGMENT_INIT": "maximum initial size of microtubules",
      "LENGTH_DIV_FACTOR": "dividing length factor",
      "DIV_LENGTH": "dividing length",
      "DIV_LENGTH_RETRACT": "retraction end dividing length",
      "V_PLUS": "growing end velocity",
      "V_MINUS": "retraction end velocity",
      "SIGMOID_K": "sigmoid function coefficient"
    }
  }
}
```

Figure 6.3: Example of a JSON-formatted configuration file for a user-defined cortical microtubule array grammar.

Within the initialization function, the initial state of the system graph can be generated using a user-defined function. The sole requirement is that this function utilizes the key generator from figure 6.1 to ensure unique keys for nodes in the system graph. For instance, in the CMA model, microtubule positions are sampled from uniform probability distributions and randomly rotated. Generally, users could adopt similar approaches or take other creative paths, such as incorporating image analysis to determine the initial state of the system.

Furthermore, structure changing rules should likewise be defined within the initialization function and added to the grammar after creation, as elaborated in subsequent sections. The specifics of their definition should come from theoretical models, observations, and experiments conducted within the relevant domain. Similarly, rate functions, affecting the frequency of stochastic graph rewrites, and differential equations, governing parameter evolution deterministically, should be defined comparably. Hence, both rate functions and differential equations are determined by theoretical or observed dynamics specific to the domain.

The simulation domain's geometry represents the physical space to be simulated, with current support limited to 2D simulations via the expanded cell complex (ECC). User-defined parameters should be set for the ECC within the initialization function. Currently, since the ECC encodes information about the simulation space's connectedness and dimensionality of each spatial domain, it aids in configuring the initial state of the graph, so it's up to the user to initialize it. However, since the ECC serves as a crucial component in the simulation algorithm, it could also be initialized internally based on user-provided parameters.

Regarding boundary conditions, DGGML imposes no restrictions. Graph rewrite rules and domain constraints on solving ODEs can be used to bound dynamics to a spatial region and can allow for custom-shaped boundaries inside the ECC. Reflective boundary conditions can be explicitly incorporated into stochastic rules. However, for runtime safety, a feature to enable "ghost cells" for the Expanded Cell Complex (ECC) has been included. In this work, "ghost cells" are inspired by the concept in numerical computing, where additional grid cells are introduced around the boundaries of the computational domain to enable the application of boundary conditions in finite difference methods. In the context of the ECC in DGGML, "ghost cells" function differently. When activated, ODE solving occurs solely within non-ghosted geocells, and any rule instance assigned to a "ghosted" geocell has a zero propensity. For a visualization of "ghosted cells" in context, refer to Figure 7.5.

Currently, the output file writer is not customizable through the user modeling interface. It saves graphs in the Visualization Toolkit file format (VTU), which is a variation of Extensible Markup Language (XML) files. VTU files can be rendered using a combination of the Visualization Toolkit (VTK) [100] and ParaView [9]. The Expanded Cell Complex (ECC) is conveniently saved at the initial time step, and the simulator periodically saves system graphs during simulations. As all graphs are spatially embedded, the system evolution can be visualized and observed.

Optional metrics from the simulation, such as the number of connected components (computed by YAGL), total node count, counts of individual node types, etc., can be defined in the metric collection interface which takes the form of the collect and print functions in figure 6.1. During the checkpointing phase, the simulator executes the collect function, which can contain any functionality defined by the user to gather information about the system state provided through the modeling interface. Additionally, metrics can be printed via a metric printing function, either to the console or to a file in a format specified by the user. As an example, a metric collection function can work with a metric printing function to transform and output data collected during the run in a format compatible with the numerical Python library (NumPy) [54], enabling visualization with Matplotlib [61].

Overall, the process of building models within the DGG formalism using DGGML is meant to be flexible to enable users to customize simulations to their specific requirements. The structured six-step methodology presented here can be used to loosely guide and allow users to translate theoretical concepts into practical simulations effectively. Furthermore, while this general approach serves as a foundation for model construction, the subsequent sections of the chapter delve deeper into defining the API for grammar rules.

6.3 Defining Stochastic Rules

Defining a rule in DGGML is simple and flexible. The grammar rule interface allows the construction of the two types of rules as objects: stochastic rules denoted as **with** rules and deterministic rules denoted as **solving** rules. The process of defining a grammar rule in DGGML first starts with understanding how the grammar of a rule can be parsed at a higher level.

The following equation is an example of a stochastic DGG rule, which uses a Heaviside propensity function:

$$
\begin{array}{ll}
\textbf{(1)} & \textbf{Stochastic Growth:} \\
\textbf{(2)} & (\,\bigcirc_1 \;\text{——}\; \bullet_2)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2)\rangle\!\rangle \\
\textbf{(3)} & \longrightarrow (\,\bigcirc_1 \;\text{——}\; \bigcirc_3 \;\text{——}\; \bullet_2)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3)\rangle\!\rangle \\
\textbf{(4)} & \qquad \textbf{with} \;\; H(\|\boldsymbol{x}_2-\boldsymbol{x}_1\|;L_{div}) \\
\textbf{(5)} & \qquad \textbf{where} \;\; \begin{cases} \boldsymbol{x}_3 = \boldsymbol{x}_2 - (\boldsymbol{x}_2-\boldsymbol{x}_1)/100.0 \\ \boldsymbol{u}_3 = \boldsymbol{u}_1 \end{cases}
\end{array}
\tag{6.1}
$$

In Equation 6.1, (1) represents the rule's name with the complete listing of the name and rule definition in figure 6.9. (2) depicts the left-hand side (LHS) graph, while (3) shows the right-hand side (RHS) graph with code for each found combined in figure 6.6. (4) is the propensity function, which is a Heaviside function of LHS parameters and is found in figure 6.7. Lastly, (5) is the **where** clause, defining how parameters are updated upon rule selection and rewriting and found in figure 6.8. In this case, parameters are sampled exactly to match computed values, but choices like sampling from a normal distribution are also valid. Thus, essential components of a stochastic rule include the name, LHS graph, RHS graph, propensity function, and sampling function for parameter updates in the **where** clause. DGGML constructs stochastic rules using these components through an object constructor, as depicted in figure 6.4.

In C++ and other object oriented programming languages, objects are created using a constructor, which is a function that initializes the state of the object upon declaration and subsequent creation. However, a FluentAPI [45] is also another choice, and functions by

```
// Signature of a stochastic rule constructor
WithRule(NameType, GraphType&, GraphType&, PropensityType&&, UpdateType&&);
```

Figure 6.4: A constructor for building a stochastic rule in DGGML, where the type of rule is named by its **with** clause keyword. The arguments in order from left to right: NameType is the rule name, GraphType is the left-hand side graph type, GraphType is the right-hand side graph type, PropensityType is the type of the propensity function, and UpdateType is the type of the updating function.

chaining methods together to initialize the object after the declaration. Figure 6.5 demonstrates how this could work. A fluent interface can work well in certain contexts and is included for completeness, but to keep it simple and familiar DGMML uses object construction as in figure 6.4.

```
// Building a with rule using method chaining
WithRule rule;
rule.name(n).fromLHS(g1).toRHS(g2).with(p).where(u);
```

Figure 6.5: An alternate design choice for defining a rule, not used in DGGML, is the method chaining approach. It starts by constructing the rule and then allows the user to continue building using a fluent-like interface. The sequence from left to right is as follows: the method name initializes a name with NameType n and calls fromLHS, which sets the left-hand side with GraphType g1. This process repeats for setting the right-hand side with GraphType g2, then setting the propensity with PropensityType p using with, and finally, calling where to set the update with UpdateType u.

At a finer level, each of the five individual parts in equation 6.1 is treated as its own object. Part (1), the name, is simply a string of characters and can be defined either ahead of time or when the object is constructed, as depicted in figure 6.9. The LHS and RHS graphs, parts (2) and (3) of equation 6.1 are created and initialized first and can be utilized in multiple rules. Figure 6.6 provides a direct insight into the code construction of the LHS and RHS graphs in Equation 6.1.

The propensity function, part (4) of equation 6.1 defined in figure 6.7, is created using an anonymous lambda function, meaning it lacks a name. Despite being assigned to a

```
GraphType lhs_graph;
lhs_graph.addNode({1, {Intermediate{}}});
lhs_graph.addNode({2, {Positive{}}});
lhs_graph.addEdge(1, 2);

GraphType rhs_graph;
rhs_graph.addNode({1, {Intermediate{}}});
rhs_graph.addNode({3, {Intermediate{}}});
rhs_graph.addNode({2, {Positive{}}});
rhs_graph.addEdge(1, 3);
rhs_graph.addEdge(3, 2);
```

Figure 6.6: The left-hand side graph is constructed by creating nodes with corresponding numbering and node types. Node 1, of type Intermediate, includes a position and unit vector. Node 2 follows suit with a type of Positive. An edge is then added between nodes 1 and 2. The process is repeated for constructing the right-hand side graph.

variable, it remains unnamed in the traditional sense. The lambda's type and signature are intentionally chosen for automatic deduction of argument types and consistency, regardless of internal content. This ensures alignment with the constructor's type requirements since propensity functions inherently depend on the LHS and any unique mapping from the rule to the matching. In certain cases, the lambda function can be detected by a C++ compiler and be optimally placed inline, avoiding the need for a function pointer call. This design choice also allows for the defining of propensity functions as function objects (functors) and provides a way for users to inject code into the generic structure of a simulation algorithm, which only requires understanding the propensity's inputs and outputs.

The update function, part (5) in Equation 6.1, is depicted in figure 6.8. Similar to the propensity function, the update function is also a lambda. However, in this case, updates are functions of the LHS and RHS, as noted in the DGG formalism. The lambda's type and signature are still intentionally chosen for automatic deduction of argument types and consistency, regardless of internal content. This ensures alignment with the constructor's type requirements, as update functions inherently depend on the LHS, RHS, and two distinct

```
auto propensity = [&](auto& lhs, auto& m)
{
    auto& node1 = lhs.findNode(m[1])->second.getData();
    auto& node2 = lhs.findNode(m[2])->second.getData();
    auto len = calculate_distance(node1.position, node2.position);
    double propensity = heaviside(len, settings.DIV_LENGTH);
    return propensity;
};
```

Figure 6.7: The propensity function is defined as an anonymous lambda function. Its signature includes a bracketed ampersand to indicate the compiler can capture external variables, such as the settings variable. The function arguments are automatically deduced types, enabling the user-defined function to access the left-hand side graph's matching and its associated mapping to the true labeling in the system without being concerned with resolving types. Within the function, data from the current matching is accessed, distances are calculated, and a threshold check determines the returned propensity.

unique mappings from the rule to the matching. Choosing a lambda may also compatibly facilitate defining the update functions as functors and allow a user to inject code into the generic structure of a simulation's rewriting algorithm, requiring only an understanding of the update's input.

Combining all five parts and considering the constructor from figure 6.4, a rule can be defined as shown in figure 6.9. The resulting constructed object precisely represents and possesses the same semantics as the rule defined in equation 6.1 using the DGG language. Once defined, the rule can be added directly into a data structure that stores all these grammar rules as data.

```
auto update = [](auto& lhs, auto& rhs, auto& m1, auto& m2) {
    for(int i = 0; i < 3; i++) // set position
        rhs[m2[3]].position[i] = lhs[m1[2]].position[i]
            - (lhs[m1[2]].position[i] - lhs[m1[1]].position[i])/100.0;
    for(int i = 0; i < 3; i++) // next set the unit vector
        std::get<Intermediate>(rhs[m2[3]].data).unit_vec[i]
         = std::get<Intermediate>(lhs[m1[1]].data).unit_vec[i];
};
```

Figure 6.8: The update function in the **where** clause is defined as an anonymous lambda function. The function arguments are automatically deduced types, allowing the user-defined function to access the left-hand side graph's matching and its associated mapping, m1, to the true labeling of the matching in the system, which is passed into the function. The same applies to the right-hand side (RHS) graph. Within the function, mappings are utilized to access and update the positions and unit vector of the resulting RHS created after a rewrite. The use of get functions is essential for retrieving the unique data associated with a given node, as it gets obfuscated due to the graph node being defined using a type that can assume several different types. Consequently, this information is only accessible to users who possess knowledge of the numbering and type of a corresponding node.

```
using RT = WithRule<GraphType>; // rule type
RT stochatic_growth("with_growth", lhs_graph, rhs_graph, propensity, update);
gamma.addRule(stochatic_growth);
```

Figure 6.9: The code translation of equation 6.1 involves defining the rule's name during construction. The left-hand side and right-hand side graphs are defined before construction, as illustrated in figure 6.6. Similarly, the propensity function is defined before construction, as shown in figure 6.7, and the updating function is defined likewise, as seen in figure 6.8. Finally, the rule is added to a grammar named gamma.

6.4 Defining Deterministic Rules

Along with stochastic rules, there are also deterministic rules. The following rule is a dynamic graph grammar ordinary differential equation (ODE) rule:

$$
\begin{array}{ll}
\textbf{(1)} & \textbf{Solving Growth:} \\
\textbf{(2)} & (\ \bigcirc_1 \text{ —— } \bullet_2) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1)(\boldsymbol{x}_2, \boldsymbol{u}_2) \rangle\!\rangle \\
\textbf{(3)} & \longrightarrow (\bigcirc_1 \text{ —— } \bullet_2) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2 + d\boldsymbol{x}_2, \boldsymbol{u}_2) \rangle\!\rangle \\
\textbf{(4,5)} & \qquad \textbf{solving} \ \ d\boldsymbol{x}_2/dt = v_{\text{grow}} \boldsymbol{u}_2
\end{array}
\tag{6.2}
$$

In equation 6.1, part (1) represents the rule's name with the complete listing of the name and rule definition in figure 6.14. Part (2) is the left-hand side (LHS) graph of the rule, and part (3) is the right-hand side (RHS) graph with code for each found combined in figure 6.11. Solving rules deviate from stochastic rules in parts (4) and (5). Part (4), found in figure 6.12, is used to provide the memory addresses of the variables with associated ODEs to the solver. Lastly, part (5) constitutes the right-hand side of the ODE solving function and is found in figure 6.13. Hence, key components of any solving rule include the rule name, LHS graph, RHS graph, variable binding function, and ODEs to solve in the associated **solving** clause. In DGGML, solving rules are constructed using these five pieces with an object constructor. Figure 6.10 provides the signature of the respective constructor.

```
// Signature of a solving rule constructor
SolvingRule(NameType, GraphType&, GraphType&, NEqType, VarType&&, ODEType&&);
```

Figure 6.10: A constructor for building a deterministic rule in DGGML, where the type of rule is named by its **with** clause keyword. The arguments in order from left to right: NameType is the rule name, GraphType is the left-hand side graph type, GraphType is the right-hand side graph type, NEqType is the number of variables to be bound, VarType is the variable binding function, and ODEType is the type of the ordinary differential equation solving function.

Zooming in, each of the five individual parts in equation 6.2 is treated as its own object. Part (1), the name, is simply a string of characters and can be defined either ahead of time or when the object is constructed, as depicted in figure 6.14. On the other hand, parts (2) and (3) are created and initialized first. Figure 6.11 provides a direct insight into the code construction of the LHS and RHS graphs in equation 6.2.

```
GraphType lhs_graph;
lhs_graph.addNode({1, {Intermediate{}}});
lhs_graph.addNode({2, {Positive{}}});
lhs_graph.addEdge(1, 2);

GraphType rhs_graph = lhs_graph;
```

Figure 6.11: The left-hand side (LHS) graph is constructed by creating nodes with corresponding numbering and node types. Node 1, of type Intermediate, includes a position and unit vector. Node 2 follows suit with a type of Positive. An edge is then added between nodes 1 and 2. Since the graph structure of ODEs is unchanged after a rewrite, the right-hand side graph can just be a copy of the LHS.

The variable binding function (part (4) in equation 6.2), defined in figure 6.12, is created using an anonymous lambda function as well. The lambda's type and signature are again intentionally chosen for automatic deduction of argument types and consistency, and this ensures alignment with the constructor's type requirements. All the previous discussion of functors and inlining also applies here. Just like the propensity function in figure 6.7, there is an input for the LHS and a respective mapping from the rule to the true labeling in the system graph. However, there is also an automatically deduced type named "varset", facilitating the binding of variables to ensure correct system solving by the ODE solver.

The **solving** function itself, part (5) in equation 6.2, is translated to code in figure 6.13. Similar to the propensity function, the **solving** function is also a lambda. However, utilizing it can be somewhat tricky due to a couple of factors. First, C++ lacks the introspection. Essentially, the names of the variables themselves cannot be used internally by the solver i.e. position, etc. As a workaround, their memory addressees are instead given to the solver.

```
vars = [](auto& lhs, auto& m1, auto& varset)
{
    for(int i = 0; i < 3; i++)
        varset.insert(&lhs[m1[2]].position[i]);
},
```

Figure 6.12: The variable binding function is defined as an anonymous lambda function. Its function arguments are automatically deduced types, allowing the user-defined function to access the left-hand side graph's matching and its associated mapping to the true labeling in the system without resolving types. Additionally, there is "varset", which is an automatically deducible type defined by DGGML. It provides the ability to insert memory addresses of variables, effectively giving them a name that DGGML can understand. Within the function, the position variables for the node in the matching passed in as the LHS corresponding to node 2 in Equation 6.2, are bound and inserted into the "varset".

Additionally, DGGML permits users to inject code into the generic structure of a simulation's ODE solver, placing responsibility on the user for correct variable access.

Combining all five parts of the grammar rule decomposed in figure 6.1 and considering the constructor from figure 6.10, a rule can be defined as shown in figure 6.14. The resulting constructed object precisely represents and possesses the same semantics as the rule defined in equation 6.2 using the DGG language. Once defined, the rule can be added directly into a data structure that stores all these grammar rules as data.

```
auto solve = [&](auto& lhs, auto& m1, auto y, auto ydot, auto& varmap) {
    //growth rule params
    auto v_plus = settings.V_PLUS;
    auto& node1 = std::get<Positive>(lhs[m1[2]].data);
    auto& node2 = std::get<Intermediate>(lhs[m1[1]].data);
    for(auto i = 0; i < 3; i++) {
        auto search = varmap.find(&data1.unit_vec[i]);
        if (search != varmap.end()) {
            NV_Ith_S(ydot, varmap[&lhs[m1[2]].position[i]])
                += v_grow * data1.unit_vec[i]*NV_Ith_S(y, search->second);
        } else {
            NV_Ith_S(ydot, varmap[&lhs[m1[2]].position[i]])
                += v_grow * data1.unit_vec[i];
        }
    }
}
```

Figure 6.13: The **solving** function is defined as an anonymous lambda function. Its function arguments are automatically deduced types, enabling the user-defined function to access the left-hand side graph's matching and its associated mapping, m1, to the true labeling of the matching in the system, which is passed into the function. The y and ydot types are utilized for integration with the ODE solver, and via the varmap (built by DGGML from the varset), they can be employed to update a local portion of the ODE system's state vector. Within the function, the **solving** equation from (4,5) in equation 6.2 is defined and a condition is included to check if a variable is coupled to another equation, using the correct value if necessary.

```
using ST = DGGML::SolvingRule<GraphType>; // solving type
ST solving_growth("solving_growth", lhs_graph, rhs_graph, 3, vars, solve);
gamma.addRule(solving_growth);
```

Figure 6.14: The code translation of equation 6.2 involves defining the rule's name during construction. The left-hand side and right-hand side graphs are defined before construction, as illustrated in figure 6.11. The number of vars is also included. Similarly, the vars function is defined prior to construction, as shown in figure 6.12, and the solving function is defined likewise, as seen in figure 6.13. Finally, the rule is added to a grammar named gamma.

6.5 Conclusion

This chapter serves as a guide to understanding the flexibility inherent in defining models within DGGML, highlighting the general approach to model definition. Furthermore, it aims to demonstrate what the integration of the DGG language and formalism into another programming language looks like. Thus, the chapter serves a dual purpose: providing general instructions on utilizing DGGML while also laying the groundwork for nonnegotiable constructs required for a language implementation. The subsequent chapter will delve deeper into analysis and simulation, offering a more comprehensive view of DGGML's capabilities.

Chapter 7

Grammar Analysis and Simulation in DGGML

7.1 Introduction

Following the grammar definition, model creation, and initialization phases, a two-stage process begins: first, the grammar analysis phase occurs, and then the simulation phase occurs. Grammar analysis takes the user-defined grammar and transforms the data into a meaningful format for the core simulation algorithm. The analysis of the grammar optimizes the representation, as well. The simulation phase takes the transformed grammar and effectively simulates a single realization of the grammar according to Algorithm 4, and in the case of no subdivision Algorithm 4 is equivalent to Algorithm 1. In the analysis section (section 7.2), a novel method is proposed to build a set of motifs, a set of fundamental pattern types that is comprised of the unique connected components of all left-hand side graphs in the grammar. In the simulation section (section 7.3), the cell list and its use for multi-object (multi-component) matching of graph objects (connected components) represent a unique contribution to this thesis. The following sections further detail both the analysis and simulation phases and build on the groundwork laid out in Chapters 4, 5, and 6.

7.2 Analyzing a Grammar

In DGGML, the grammar analysis phase starts after the initialization function of the user-defined model is called within the simulate function shown in the sample code of figure 6.2 in Chapter 6. The grammar, which is defined inside the initialization function of the user-defined model, is defined using the API in Chapter 6. During runtime, this data, along with relevant simulation information, populates core data structures. In a fully featured language rather than a library like DGGML, this phase would be integrated into a DGG transpiler, occurring during semantic analysis and code generation. However, DGGML is a library with some similarities to an embedded language in C++, so analysis happens during the simulation execution rather than inside a compiler.

Figure 7.1 gives a general overview of what an abstract syntax tree would look like for the grammar rules and motivates the analysis discussion. At the topmost layer is the grammar with its name, and that is composed of all of the stochastic rules and all of the deterministic rules. Each of the terminal nodes in the "WithRule" and "SolvingRule" subtrees in figure 7.1 directly map to what is seen in the constructors in Chapter 6, figures 6.4 and 6.10. So, when a rule is constructed and then added to the grammar it is transforming the model into a data format that can be semantically analyzed.

Figure 7.2 emphasizes the relationship between the AST and the data by zooming in on the stochastic rules subtree and transforming it into a data representation. The distinction importantly allows the "WithRule" to be thought of as a data structure, composed of the syntactic pieces of data in the leaf nodes. The same holds for the "SolvingRule". DGGML

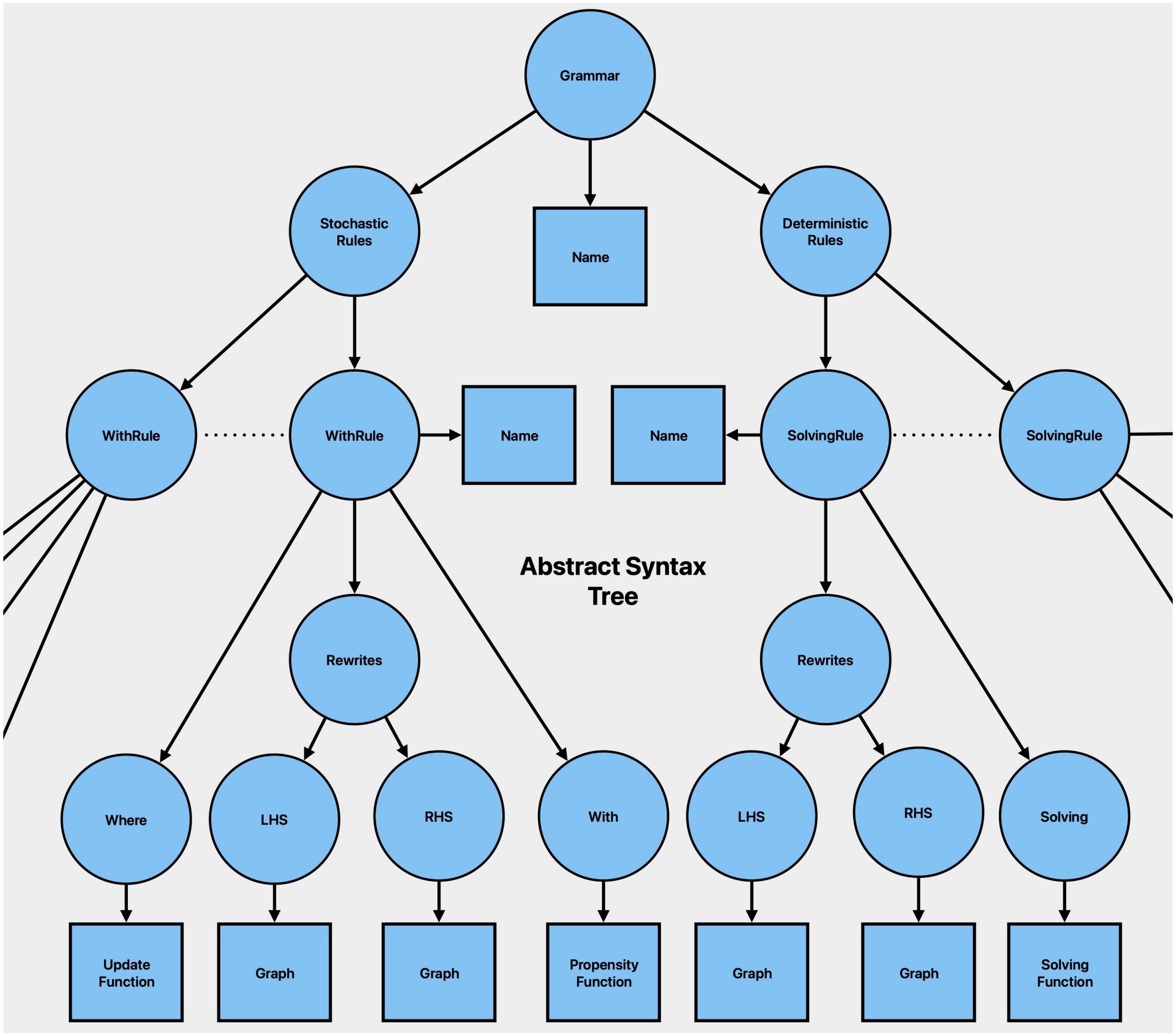


Figure 7.1: A section of an abstract syntax tree (AST) for a DGG, where circle nodes depict syntactic constructs composed of other nodes, while square nodes represent "leaf" or "terminal nodes", denoting the smallest syntactic units. Arrows pointing out from a node indicate its composition of the pointed to nodes. Dotted lines signify the repetition of a subtree with similar properties.

effectively stores all instances of these rule-type data structures in a map, where the keys are the name of the rule and the value is the rule instance itself. A fundamental requirement of this type of mapping requires rule names to be unique, which is enforced by the library. Another benefit of this naming convention lies in its ability to directly map user-defined functions to matches of rule instances, serving as a method to bind and resolve functions at runtime.

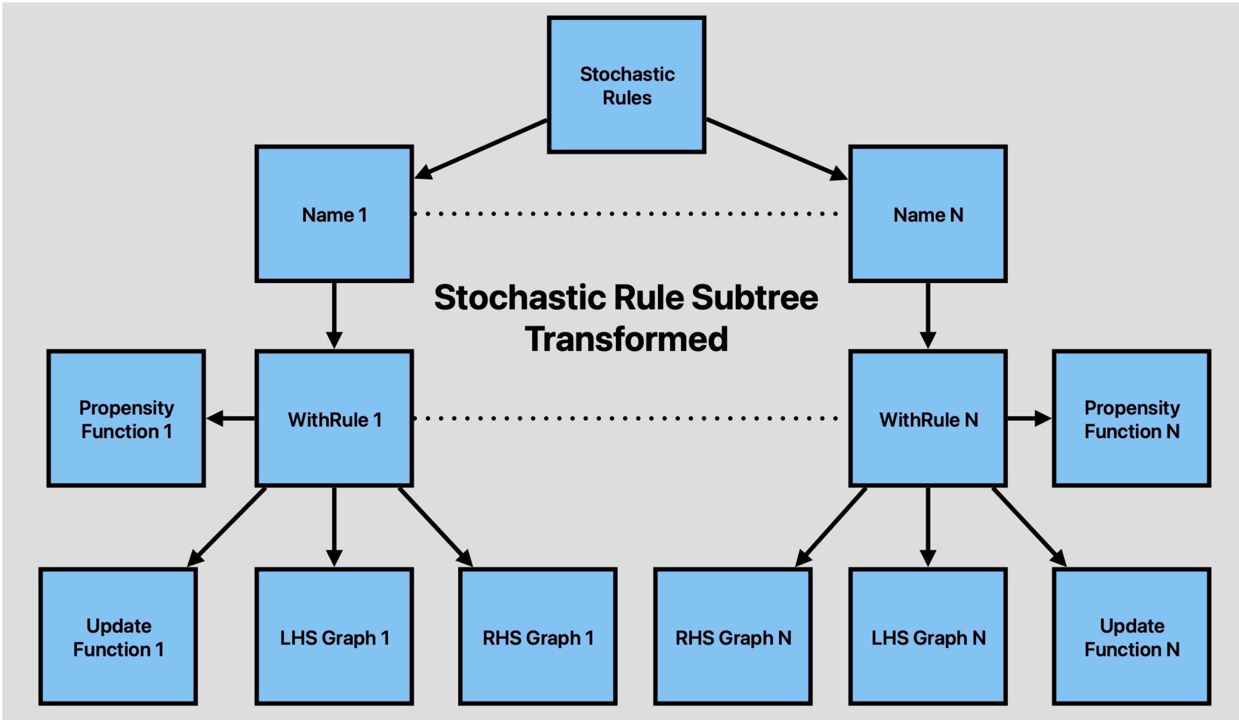


Figure 7.2: A transformed stochastic rule subtree, originating from an abstract syntax tree representing a grammar. In this hierarchy, squares represent data with arrows leaving a square indicating its composition with the pointed elements. Dotted lines denote the possible omission of additional instances of the names and “WithRules.”

Once the grammar rules are structured into an efficient data format, grammar analysis can begin. The initial step, which may occur in any order during this phase, involves precomputing the transformation from the LHS to the RHS. This ensures that when a rewrite occurs, a set of predefined steps will be applied. The “Rewrite” node in figure 7.1 represents this functionality, interpreted through its semantics. The rewrite is represented by four sets: nodes to be created, nodes to be destroyed, edges to be created, and edges to be destroyed. For a more comprehensive understanding of this procedure, the logic and motivation further detailed in the graph transformation building block are discussed in Chapter 5.

Next, the LHS graphs undergo analysis, as shown in figure 7.3. Each rule is processed, irrespective of being a solving or with rule. Every LHS graph is searched, and its connected components are identified. The components are stored and a mapping between them and the LHS is created. The ordering of how these connected components are stored represents the search pattern for the rule match. The components themselves serve as the fundamental objects in the pattern. The unique connected components that build the rules in the grammar are known as “motifs”. Subsequently, the components of all the LHS graphs are consolidated into a set containing unique motifs. The numbering of motifs in this set is a renumbering and not specific to each rule's component. However, as multiple LHS graphs may share the same motif structure (i.e. they are isomorphic), a mapping between the LHS of a rule and the motif is stored to ensure proper usage when invoking user-defined functions within the simulation. The uniqueness is determined using a custom graph isomorphism algorithm implemented in YAGL and largely inspired from reference [119].

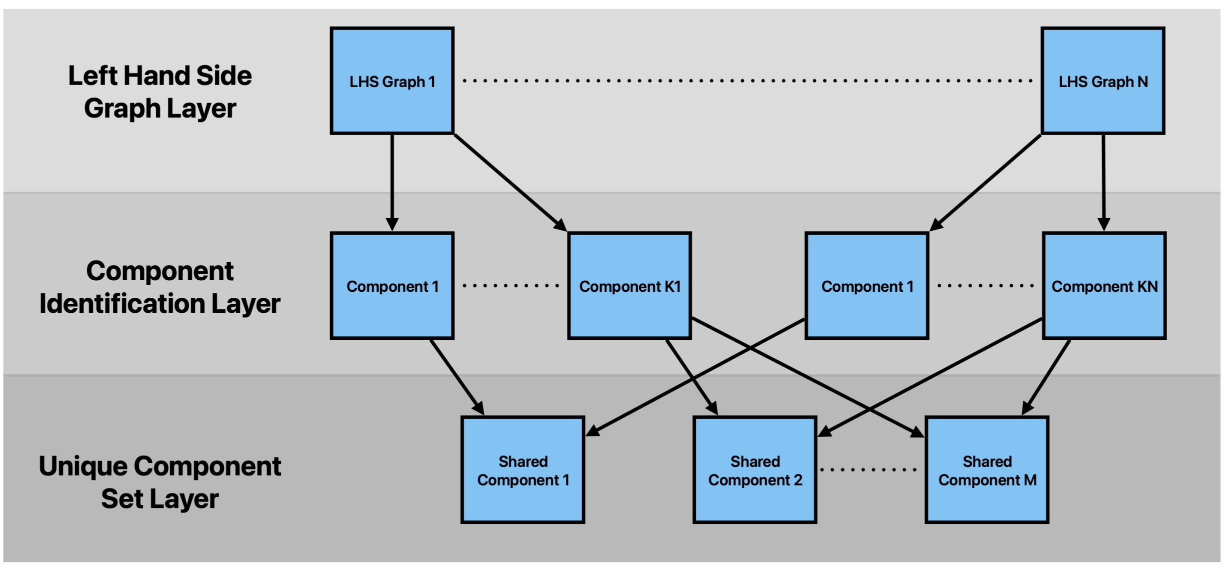


Figure 7.3: An overview of the parsing process, elucidating how graphs from the left-hand side of all grammar rules are parsed into their fundamental components (motifs) for each graph. Subsequently, these motifs are consolidated into a set of unique motifs, the shared components. In this hierarchy, squares represent data, with arrows leaving a square indicating its composition with the pointed to elements. Dotted lines denote the possible omission of additional instances at that particular layer.

The set of constructed motifs is then used to identify all unique instances in the system graph for each motif, using the subgraph-specific pattern recognizer (SSPR) building block described in Chapter 5. Figure 7.4 illustrates the relationship between a shared component and an instance of the motif pattern found. In the unique component set layer, each motif can be associated with multiple matching instances. This relationship is characterized as one to many, with instances in the figure 7.4 intentionally arranged in an unordered manner. This configuration emphasizes the potential for non-uniform access patterns for components, highlighting the fact that grouping components of the same type may not always be logical. This is especially apparent when considering rule matches, which are combinations of components themselves.

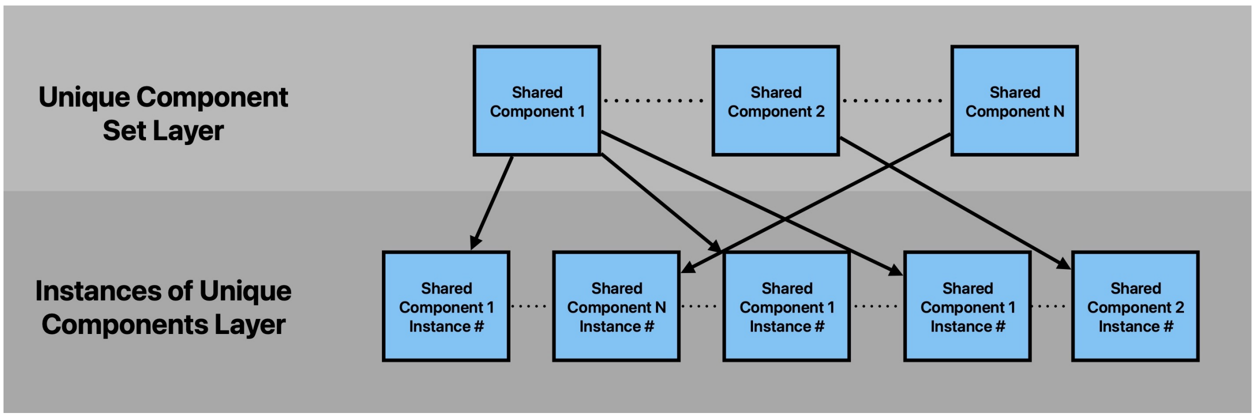


Figure 7.4: A pattern matching hierarchy, elucidating how fundamental components graphs (motifs) from a consolidated set of unique motifs are utilized to identify instances of the matching motif pattern. In this hierarchy, squares represent data, with arrows leaving a square indicating its composition with the pointed elements. Dotted lines denote the possible omission of additional instances at that particular layer.

In DGGML, this association is realized through a "component match map" data structure. Each component match includes keys that correspond to the component's order as defined by the rooted spanning tree (RST) generated by the SSPR. Additionally, each component match is assigned a unique key from a random key generator, a distinct identifier indicating the type of motif it matches, and the key of a node designated as the match's anchor. Considered a compressed representation of the original motif graph pattern, a component match does not explicitly retain edge information, although this information can be easily reconstructed using the original pattern. Other storage alternatives for these matches exist as well. Matching components are stored in a map and hashed by their unique IDs, facilitating various operations involving components such as building rule matches, which merely require querying this set.

Rule matches follow a structure similar to the map of component matches. Each rule match consists of a list of unique keys representing the associated components, accompanied by an anchor chosen from the first component in the list. Rule matches are then collected into a data structure just like the component matches. Each rule is hashed by a uniquely generated key and stored in the map.

7.3 Simulating a Grammar

Following the initialization of the model and grammar analysis, additional steps are necessary to fully initialize the simulation before execution. First, empty lists must be created to store all rules associated with each geocell. The number of lists required for each geocell dimension can be conveniently determined from the expanded cell complex (ECC) building block described in Chapter 5, which is constructed during model initialization. For example, figure 7.5 has nine 2D geocells, twelve 1D geocells, and four 0D geocells. We would then have twenty-five lists in total.

Each geocell is responsible for its own exact simulation and monitors components that become invalid i.e. no longer matching, during the process. This monitoring is used in the *synchronization* phase of the approximation of the exact algorithm (Algorithm 4) where any inconsistencies between rule matches owned by other geocells are corrected. This is the stage where errors from the commutator can manifest in the simulation. Since each geocell operates locally as an exact simulation, it requires its own propensity and warped waiting time, denoted by τ. Initially, the geocell's propensity is set to zero, and its waiting time is sampled from an exponential distribution as in Algorithm 4, Chapter 4.

At this point, DGMML creates a checkpoint by default, and if the user defines it - the ECC and the initial state of the system graph can be saved. By default, DGGML includes a file writer for the visualization toolkit (VTK) [100] file format to visualize the simulation in Paraview [9]. Initial metrics, if defined from the metric collection interface, are collected at this step and also optionally saved or printed as defined by the user. The generic nature of how and what metrics to collect during the simulation runtime reflects an inherent additional layer of analysis that could be interspersed between simulation steps.

Next, a specialized cell list is constructed. A "cell list" is a spatial data structure commonly utilized in Molecular Dynamics simulations [109], and is integral to efficiently simulating particles making use of Verlet integration [4]. This data structure acts as a spatial partitioning grid, dividing the space into cells or buckets, with each cell representing a distinct spatial region. Objects or data points, such as components in the case of DGGML, are assigned to these cells based on their spatial positions. The cell size is determined by a user-defined parameter known as the reaction radius. The reaction radius is the distance at which nearby objects in a system can interact with each other with non-negligible probability. Typically the cell size will be set to a multiple of the reaction radius so that the cell list can stay valid for a short period of time. Eventually, motion in the system will cause it to become out of date. While it is still up to date, the cell list is advantageous because it enables rapid spatial queries by confining searches to neighboring cells' lists. Cell lists address geometric search problems which are important for collision detection [43], with other well-known methods including K-d trees [14] and bounding volume hierarchies [66].

The cell list can be thought of as a spatial reaction grid. Its existence is justified by the propensity function "fall off" of a rule instance, i.e. when the components of a match are too far apart, they are skipped because the propensity is assumed to be negligible. The ECC and a reaction grid are visualized in figure 7.5. For this work, a cell complex of a regular Cartesian grid is expanded. A cell complex graph is used for the expansion and it is labeled with additional data for the well-separatedness criteria. The reaction grid is a finer-scale Cartesian grid and it is aligned with the coarser scale of the ECC. Regions of the same pre-expansion dimension are separated from each other by a minimum distance, and so are the reaction cells contained in them. The grid spacing of the reaction cells in DGGML is set using the reaction radius, and this in turn represents the "fall off" distance. Therefore, reaction grid cells keep the search space for reactions spatially local and are smaller than the minimum separation distance of geocells.

The cell list functions as a spatial reaction grid, and is necessitated by the propensity function's "fall-off" behavior in rule instances; when match components are too distant, they are disregarded due to their assumed negligible propensity. Both the ECC and the reaction grid are depicted in figure 7.5. In DGGML, the ECC and its underlying reaction grid meet the well-separatedness criteria. Regions of identical pre-expansion dimensions are separated by a minimum distance, as are the reaction cells within them. The reaction grid, aligned with the ECC's coarser scale geocells, constitutes a finer-scale Cartesian grid essential for the function of the cell list. Furthermore, reaction grid cells maintain spatially local search spaces for rule firings and are much smaller than the geocells' minimum separation distance.

Therefore, the reaction grid essentially implements the concept of a cell list. Given a cell list aligned with the ECC, the components must have a way to be mapped to cells in the cell list, and DGGML uses a 2D spatial hashing method [93]. The hashing function is designed for single points; however, since component matches in DGGML refer to subgraphs of the spatially embedded system graph, they typically span more than a single node. To simplify, the anchor node of a component match serves as the input for the spatial hashing function. The spatial hashing function used in the cell list bears a striking resemblance to the single point φ function, and this similarity is not coincidental. Using a single point, the spatial hashing function uses modular arithmetic to determine in $\mathcal{O}(1)$ time the (i, j) index pair of the grid cell to which a point belongs. Accordingly, this pair is converted into a 1D cardinal

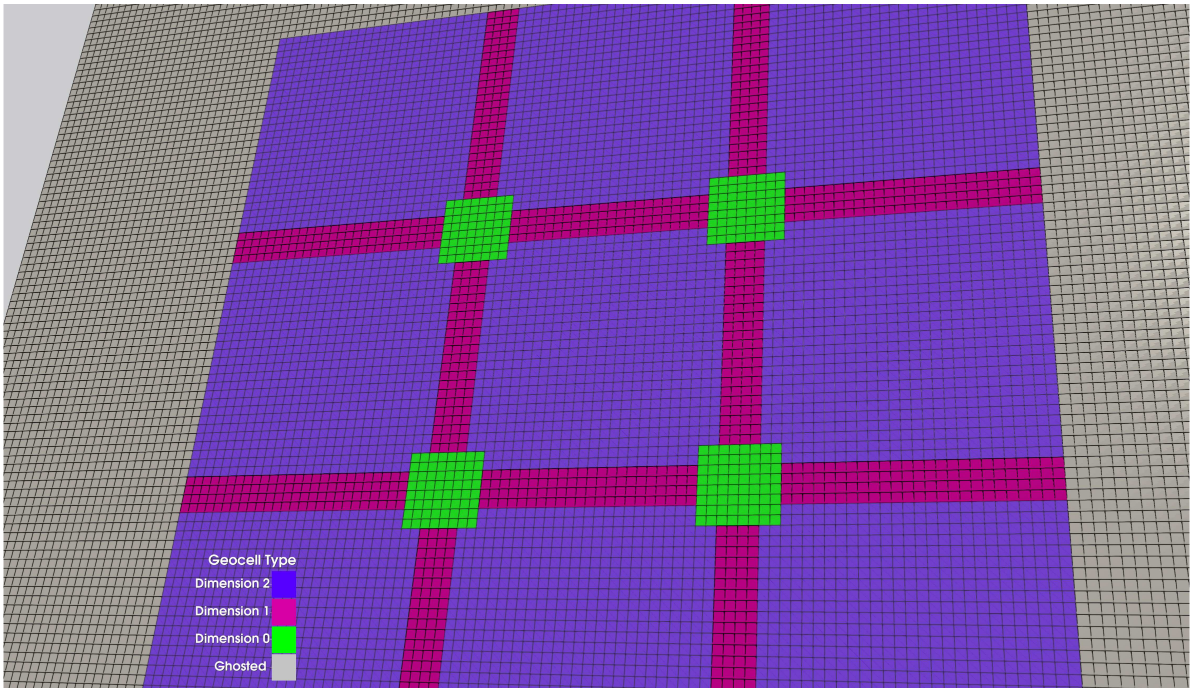


Figure 7.5: Expanded cell complex (ECC) with well-separated lower dimensions. Regions of the same pre-expansion dimension are separated from each other. Note how only interior lower dimensional cells are expanded. A reaction grid is aligned with the geocells, and reaction cells are smaller than the geocells. The outer boundary of the ECC is padded with optional ghosted geocells. Ghosted geocells are just geocells that are not processed by Algorithm 4. These optional ghost cells operate as a buffer for any computational errors or as a capture condition in the case of no grammar rules addressing boundary conditions.

index, serving as the hash code for the corresponding cell covering a region of space. This approach enables a quick mapping of spatially embedded objects associated with a cell.

Using the newly constructed cell list of component matches, all rule matches can be computed by performing spatial queries and searching for a matching pattern in the results. Essentially, for each component in each cell in the cell list, all component matches within a nearby radius are queried, providing a set of candidates. Given the ordering of the pattern of components of an LHS determined during the grammar analysis phase, only component matches that were previously matched by the SSPR will complete the pattern for the rule match. A consequence of finding all combinations rather than just a single permutation is that automorphisms of the matching graph rules LHS are also found.

After identifying all rule matches, the matches are mapped to the appropriate geocell using the minimum-dimension function φ from Chapter 4, section 4.5. This choice ensures that all automorphic matches, regardless of component ordering, are assigned to the same dimension and therefore the same geocell. These matches are then stored in the list of rules associated with that geocell. While other storage options such as a tree structure could be considered, this decision is a design choice that does not affect the algorithm's function.

Next, the core simulation loop begins. For each dimension, the simulation runs for a brief period, as described in Algorithm 4. While there are no restrictions on the order in which dimensions can be simulated, DGGML defaults to simulating in the order of $2D$, $1D$, and then $0D$, in a cycle. The geocells within each dimension can be executed in parallel or serially, although the current version utilizes a serialized mode of Algorithm 4. Parallel execution on the CPU requires thread-safe data structures for accessing and updating the system's state. Similarly, running in parallel on accelerators like GPUs would require a design that maximizes throughput while keeping in mind similar considerations.

For each geocell in Algorithm 1, the simulation requires the grammar, rule matches, component matches, ECC, cell list of components, and geocell related properties. In the core of the simulation algorithm, the initial step involves computing the propensities for all rule instances in the system individually, followed by summing them globally to obtain the total propensity for each of the user-defined stochastic **with** rules. For each rule match in the map of rule matches, the associated propensity function is queried based on the rule's unique name (e.g. such as the name "with_growth" in figure 6.9 in Chapter 6). The rule match and its associated component matches are then passed into the user-defined propensity function, along with a map facilitating manipulation of a matching graph consistent with its original numbering in the grammar. Summing the propensities of each rule yields the total propensity of the system. As mentioned in the parallel version of the exact algorithm (Algorithm 2, this calculation is a great opportunity for a parallel reduce, and could be accomplished using performance portable parallel libraries like Raja [12] and Kokkos [116].

Next, the ODE solver is initialized. When processing a match, the function outlined in Chapter 6 to bind variables dynamically constructs a local set of equations. Since different grammar rules may define equations involving the same variables, a set of memory addresses is used. The value of each variable in the set is copied to an array compatible with the solver, representing the state vector for the ODE system. Additionally, a map (hash table) is created to map the variable's memory address to its location in the state vector. This enables locally named access to variables within the user-defined **solving** function. An extra equation for the τ time warping is included as in [129, 76], along with a function to check if τ has exceeded the sampled waiting time for the geocell. The initial condition of the warping equation is set as the total propensity.

After initializing, the ODE solver solves for a specified period, which is always shorter than the longer time period for which a geocell for a given dimension can be run or until the waiting time is reached, signaling the occurrence of rule firing. Between **solving** steps the total propensity calculation is carried out, as mentioned earlier. Upon surpassing the sampled threshold according to the time warping equation, a rule is chosen to fire. The selection method constructs a discrete distribution from the computed propensities for rule types and samples from it to make the selection.

Once the rule is selected, the set of rewrite operations produced during the grammar analysis phase is applied. Consequently, the incremental update, as discussed in Chapter 5, is performed using the resulting rewrite operations applied to the system graph. This process begins with the invalidation of any component and rule matches. Next, new matches are incrementally updated using the rule matches. Any invalidated components that may participate in a rule of a differing dimension are then set aside (recorded) for future processing. Additionally, rules that are valid but should be mapped to another geocell are also set aside. Finally, the waiting time, τ is resampled, and the ODE solver is reinitialized. This involves removing any old equations invalidated by the rewrite and adding any new equations created by completing a new match for a **solving** rule.

The entire process continues until the specified time for that dimensional geocell simulation is reached. Upon completion, the current propensity and waiting time of the geocell are returned. Additionally, any instances of invalidated component matches that may participate in rules assigned to neighboring geocells of a lower dimension, or potentially still valid rules belonging to geocells of a lower dimension, are returned. The remaining geocells in the same dimension are processed. Rules belonging to geocells of lower dimensions are checked, and any appropriate rules are invalidated. The returned rule matches are then checked to see if they still contain valid components. If so, they are mapped to the correct geocell using φ.

After completing this full loop, i.e., simulating for a sufficient Δt, the set of rule matches and the cell list can optionally be fully rebuilt or repaired. The duration of simulation time a collection of matches stays valid is crucial, as matching patterns of spatially embedded graph grammar that move may eventually have the propensity to "fall off" and should no longer be considered. At least a couple of options are available. When the overall motion of component matches in the system is considered to change infrequently, only incremental local updates to the data structure are necessary. However, in a high resolution chemical reaction DGG where many molecular graphs are moving and interacting with each other, the cost of incremental updates to recompute new rule matches after a firing may become more expensive than simply recomputing all rule matches.

By default, the simulator periodically recomputes all rule matches but does not recompute component matches. These component matches represent fundamental patterns that only change when a structural rewrite occurs and will never become invalid due to rewrites of the position parameter - a sort of invariant. In a more fully featured simulator, monitoring the position and velocity of component matches in the system could potentially automatically detect when recomputing all rule matches is necessary. Finally, optional metrics specified by the user are collected and optionally saved. The simulation repeats the core simulation loop until it reaches its stopping criteria.

7.4 Conclusion

In DGGML, grammar analysis is integrated into simulation initialization, enabling semantic analysis of user-defined grammar rules. The efficiency of grammar simulation is achieved through incremental updates and the use of a cell list. DGGML offers customizable simulation of DGGs, leveraging features like dynamic ODE solving, while also providing metric collection during the simulation loop. The model building interface of DGGML abstracts away much of the complexity of the underlying simulation algorithm, opening up the path for higher-level usage and modeling applications.

Chapter 8

Modeling the Cortical Microtubule Array

8.1 Introduction

The primary work in the chapter is taken from [73]. Dynamical graph grammars (DGGs) are capable of modeling and simulating the dynamics of the cortical microtubule array (CMA) in plant cells by using an exact simulation algorithm derived from a master equation; however, the exact method is slow for large systems. In Chapter 4 we presented work on an approximate simulation algorithm that is compatible with the DGG formalism. The approximate simulation algorithm uses a spatial decomposition of the domain at the level of the system's time-evolution operator, to gain efficiency at the cost of some reactions firing out of order, which may introduce errors. The decomposition is more coarsely partitioned by effective dimension (d=0 to 2 or 0 to 3), to promote exact parallelism between different subdomains within a dimension, where most computing will happen, and to confine errors to the interactions between adjacent subdomains of different effective dimensions. In this chapter, to demonstrate these principles we implement a prototype simulator (CajeteCMA) that is the precursor to DGGML, and run three simple experiments using a DGG for testing the viability of simulating the CMA. We find evidence that the initial formulation of the approximate algorithm is substantially faster than the exact algorithm, and one experiment leads to network formation in the long-time behavior, whereas another leads to a long-time behavior of local alignment.

8.2 Biological Motivation

Eukaryotic organisms comprise complex cells with many subsystems that are well-suited to be modeled with dynamic graphs. Over time cells can divide, allowing for a plant to grow, among other processes. Understanding the exact biomechanical mechanisms taking place during cell division in plants is a long-standing question [123], but it is known that there is a connection between division plane orientation in plants and a change in the orientation of cortical microtubules (MTs) associated with the plasma membrane [24], as they form the pre-prophase band (PPB). For example, one hypothesis for the PPB orientation process is "survival of the aligned" [115], and another is alignment through selective katanin mediated severing [35]. The ensemble of MTs associated with the plasma membrane of the cell is the cortical microtubule array. The question of how MTs contribute to cell shape and other processes during cell division motivated us to develop a simplified model for the dynamics found in the CMA, with the potential to extend this work to larger systems with more complicated dynamics and interacting networks at different spatial scales.

8.2.1 Mapping biology and relevant physics to dynamical graph grammars

In plant cells, the cortical microtubule array (CMA) plays an important role in cell division and determining shape [98]. A microtubule is a polymer composed of alternating α and β tubulin proteins arranged in a cylinder of usually 13 longitudinal protofilaments [19]. MTs can be thought of as relatively stiff tubes around 25 nanometers in diameter. They can be coarsely represented in a graph as chains of stiff rod segments. Cortical microtubules (CMTs) in the CMA undergo structural dynamics such as treadmilling, zippering, induced catastrophe, and crossover [24] - all of which can be represented as DGG rules.

The graph representation of CMTs is compatible with elastic dynamics and beam theory [69]. For example, in [84], MTs are represented as a string of points, using the standard formula for bending elasticity to allow MTs to bend under external forces and resist these forces elastically. The persistence length is one way to measure a microtubule's resistance to bending, and it characterizes the length scale over which the microtubule maintains its direction while indicating its stiffness or flexibility. External forces include random thermal fluctuations, which can be described using the Boltzmann distribution. Thermal fluctuations can cause the MT to lose directionality over short length scales, resulting in a shorter persistence length [86]. The Boltzmann distribution plays a crucial role in determining the probability of the MT moving to a different energy state with a particular conformation and persistence length.

Currently, the model does not include the exact physics of these internal and external forces directly; however, these dynamics could be added as ODEs attached to nodes in the graph. The ODEs supported within the DGG formalism can also be extended to stochastic differential equations (SDEs) that include random fluctuations. In the current work, we have simply approximated fluctuations in the direction of the growing end by adding in small perturbations in the direction of growth when an instance of the stochastic rewrite rule in equation 8.1 below is selected to occur.

While microtubules are stiff (but still bendable) and can function to provide structure to a cell, they also have dynamic properties. In particular, it has been observed that MTs have the ability to undergo rapid growth and shrinkage, known as dynamic instability [107]. Dynamic instability and dynamic MTs provide a means for the cell to reorganize the cytoskeleton rapidly during cell division [19] or because of changes in the environment [15]. It has also been hypothesized that microtubules can act as tension sensors [53], providing biomechanical feedback.

During dynamic instability, the microtubule can grow by rapidly polymerizing tubulin protein sub-units bound to guanosine triphosphate (GTP) [19]. The cell must keep the concentration of GTP-tubulin high to allow for polymerization [19]. As long as the end remains stable, the microtubule will continue to grow, but as soon as instability is reached the microtubule begins to splay apart and shrink [53]. In the grammar, we encode these dynamics into our stochastic/deterministic growth/retraction rules. These processes are called rescues and catastrophes, respectively. Dynamic instability is regulated by microtubule associated proteins (MAPs) and incorporating them is a possible path for future work.

CMTs in the CMA also are subject to additional structural graph-changing dynamics. Three primary processes have been observed when one MT collides with another [24] and the mechanisms that control them are still a subject of debate. They are: zippering, crossover (junction formation), and induced catastrophe. If we let θ be the angle of collision and θ_{crit} be the critical angle of collision, zippering occurs at a higher probability with $\theta < \theta_{crit}$ and catastrophe and crossover occur at $\theta \geq \theta_{crit}$ [37]. Grammar rules for the mentioned dynamics are provided in the supplementary material.

For this chapter, we restrict our simulation to be an idealized version of a region of the plasma membrane and leave an exact physical interpretation for future work. We focus on replicating the simplified dynamics mentioned and using an implicit capture condition (growing MTs reaching the boundary are no longer simulated) for MT segments that reach the simulation boundary [121]. In the future, we could impose a more realistic boundary condition on the domain (such as capture and release) and add additional grammar rules to model transport dynamics within cells [98].

The following is an example of a stochastic dynamic graph grammar rule for growth:

$$
\begin{aligned}
&\textbf{Positive MT Overgrowth:}\\
&(\ \bigcirc_1 \text{———} \bullet_2) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2) \rangle\!\rangle\\
&\longrightarrow (\ \bigcirc_1 \text{———} \bigcirc_3 \text{———} \bullet_2) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3) \rangle\!\rangle\\
&\qquad \textbf{with} \ \ \sigma\Big(\frac{\|\boldsymbol{x}_2 - \boldsymbol{x}_1\|}{L_{\text{div}}}; k = 10\Big)\\
&\qquad \textbf{where} \ \ \begin{cases} \boldsymbol{x}_3 = \boldsymbol{x}_2 - (\boldsymbol{x}_2 - \boldsymbol{x}_1)\gamma \\ \boldsymbol{u}_3 = \frac{\boldsymbol{x}_3 - \boldsymbol{x}_2}{\|\boldsymbol{x}_3 - \boldsymbol{x}_2\|} \end{cases}
\end{aligned}
\tag{8.1}
$$

In equation 8.1, $\sigma(\ell/L_{div}; k) = 1/(1 + e^{-k(\ell/L_{div})})$ is a sigmoid activation rate function. Here $\ell = \|\boldsymbol{x}_2 - \boldsymbol{x}_1\|$ is the length of the edge, L_{div} is the maximal dividing length and $k = 10$ is a "gain" parameter that determines how quickly the function turns on as the edge length gets close to the dividing length. Other k values could work, but we choose 10 for a rapid activation. The rate function increases rapidly when an MT segment grows too long which increases the propensity that a growth rewrite rule-firing event will occur. The quicker the rate function activates, the sooner a new segment is added when the threshold is reached.

In the results section we make use of the term "starting MT". What we mean by "starting MT" is a graph of the form:

$$
(\blacksquare_1 \text{———} \bigcirc_2 \text{———} \bullet_3) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3) \rangle\!\rangle \tag{8.2}
$$

A "starting MT" has a retraction node (closed square), an intermediate node (open circle), and a growth node (closed circle). Edges simply represent relationships between nodes and the distance between the nodes in space can be computed by using the l_2 norm and node position vectors $\vec{X}_i$. As rewrite operations are applied to the MT using e.g. the rule in equation (8.1), growth is simulated.

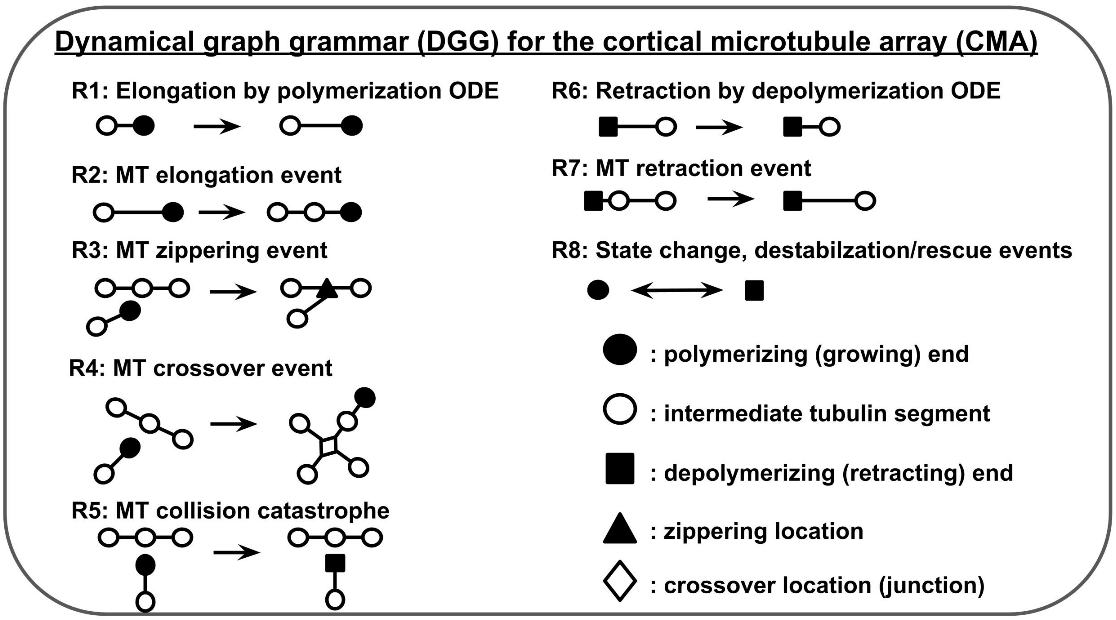


Figure 8.1: Summary of all the rules used in the CMA grammar.

In figure 8.1 we include a high level overview of all the graph rewrite rules used in the CMA grammar. Rule 1 is a deterministic rule that models the elongation of a polymerizing MT (growth) with an ODE. Rule 2 is a stochastic rule also used to model growth. When an MT segment becomes too long under the action of Rule 1, Rule 2 can insert another node and split the segment into two segments as seen in equation 8.1. Rules 3 through 5 are stochastic rules that determine what outcomes may occur when a growing end of an MT comes close to two intermediate segments. In Rule 3, the outcome is zippering (bundling) if the MT comes in at a shallow angle. In Rule 4, the outcome is the MT crossing over the other and forming a junction. In Rule 5, the outcome is a catastrophe event and the colliding MT destabilizes and begins to retract. Rule 6 is another deterministic ODE-solving rule like Rule 1, but in this case, it models retraction. Rule 7 is the stochastic version of retraction, like Rule 2, and determines what happens when an MT segment gets too short. Whereas Rule 2 adds a node, Rule 7 removes a node. Rule 8 is a reversible stochastic rule that allows the growing end and retracting ends to change states, to effectively model dynamic instability. Further details on ODEs, propensity functions, and rules may be found in the Appendix A.

8.2.2 Overview

We have developed and implemented preliminary work on an approximate algorithm (Chapter 4, Algorithm 4) implemented in C++ for accelerating the simulation of spatially embedded DGGs. Our simulator is also capable of running the exact algorithm (Chapter 4, Algorithm 1), which is used as a baseline for the performance comparison. The current code is serial and single-threaded, leaving substantial room for parallel speedup due to the "parfor" in Algorithm 4. We have tested and evaluated our prototype simulator by running three experiments using the example CMA DGG found in Appendix A. The CMA DGG uses artificial parameters to demonstrate proof of concept, and in the future more biologically inspired ones should be used.

In the first experiment, we simulated the CMA DGG several times for 1600 MTs and we evaluated the long-time behavior of the realizations. Our realizations include the change in the quantity of the five types of nodes of the microtubule graphs over time: retraction (negative growth), intermediate (interior nodes), elongation (positive growth), zipper (bundling), and junction (crossover). A plot of each of these node types and the several realizations of the simulations for the first experiment can be seen in figure 8.5.

In the second experiment, we simulate the CMA DGG for 1600 MTs again, but with a low crossover rate and all other conditions remaining the same. We evaluate the long-time behavior of the simulation and compare it to the long-time behavior of the first experiment.

A side-by-side comparison of the ending states can be seen in figure 8.11.

In the third experiment, we analyze the run-time performance of 3200 MTs with different domain decompositions. In the performance section, we include a comparison plot (figure 8.13) of performance using the exact algorithm (1x1 case) vs. the approximate algorithm (remaining cases) for the CMA DGG. The approximate algorithm allows for speedup by breaking the system into well-separated reaction sub-systems, which obviates the need to evaluate most possible matches, and by firing some rules out of order as defined by operator splitting, at the cost of accuracy.

8.2.3 Experiment 1: Long-time Network Formation

We initialize each simulation of the system with 1,600 MTs. An example of the starting state of a realization is seen in figure (8.2a). The initialization follows a uniform distribution of MTs over the domain space, where MTs are randomly rotated. The domain is $\Omega = \{0 \leq x, y \leq 100 \mid x, y \in \mathbb{R}\}$, a 100×100 square area in $\mathbb{R}^2$. For each simulation, we subdivide the domain into a uniform 8×8 grid resulting in 64 grid cells. The subdivided domain is then transformed into a cell complex and expanded.

The average number of initial MTs per highest dimensional cell is 25, since we have $1,600$ uniformly distributed MTs over 64 subdomains. The average node density is then 75 i.e. 25×3 when using the starting MT as specified in equation 8.2. The initial average MT density is chosen to be 25 per subdomain, rather than a larger quantity to keep the starting MTs well-separated and to allow for polymerization to occur before junction/zipper formation begins (room for growth). We take the boundary conditions to be the capturing condition [121].

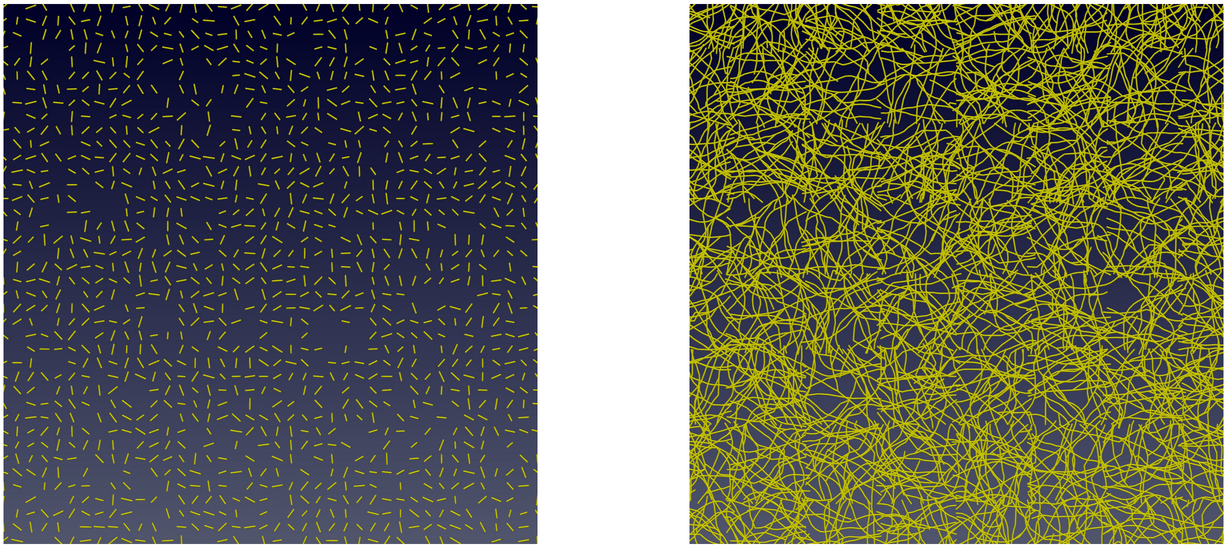
System Dynamics and Long Term Behavior

For the experiment, we let the simulations run for 1600 units of simulated time. We define one unit of coarse-scale simulated time $\tau_{ref} = l/v$ to be the time it takes one MT positive node to move a segment distance l, given a velocity v. Conceptually, this is similar the the Courant–Friedrichs–Lewy (CFL) condition in numerical PDEs [112]. Under our constraint, we ensure that not too much happens in the system in one time unit as required by our approximate simulation algorithm.

Propensity function model parameters are chosen to evaluate the simulation algorithm and code, by equally exercising all the DGG rules derived from recent literature, rather than to represent biophysical knowledge.

Figure (8.2b) is the final state of the realization of the third simulation after time = 1600 τ_{ref}. It shows network formation and the onset of a steady state in the long-term behavior of the system. A side-by-side comparison of the starting state and ending state can be seen in figure 8.2. The starting state in figure (8.2a) shows 1600 disconnected starting MTs uniformly distributed. In figure (8.2b) the ending state consists of a highly connected network, and the same behavior occurs in the other realizations. We verify this with a plot of the connected components for all simulations in figure 8.3.

In figure 8.3, we start with 1600 connected components for each simulation. One connected component for each MT is exemplified in figure (8.2a). Over time, we see the connected components decrease and trend toward the long-time behavior of a highly connected network. A fully connected network is expected to emerge if we ran the simulations for longer. To make the difference in connected components of each realization clear, figure 8.4 is included. In figure (8.4a) all realizations are plotted from the beginning to iteration 400. Figure (8.4b) plots the realizations from iteration 400 to the end and distinctly indicates the slight difference between realizations in the number of connected components over time.



(a) Starting state of the simulation (t = $0\,\tau_{ref}$). A uniform distribution of 1600 MTs.

(b) Ending state of the simulation (t = $1600\,\tau_{ref}$). A highly connected network has formed.

Figure 8.2: Side by side comparison of beginning and end state of the CMA DGG simulation of 1600 MTs for realization 3.

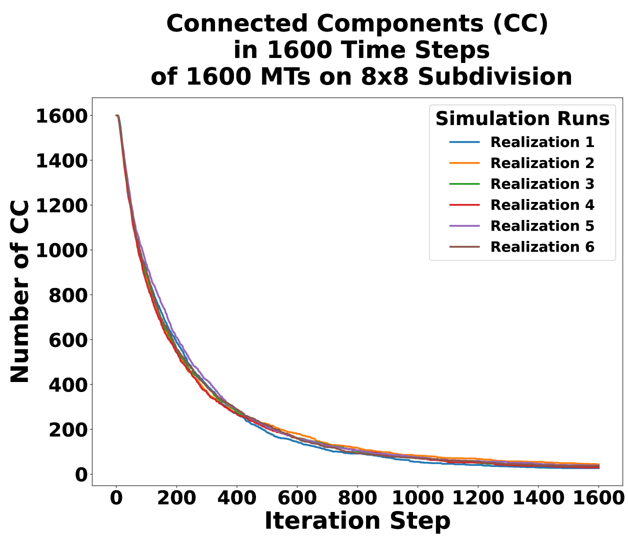
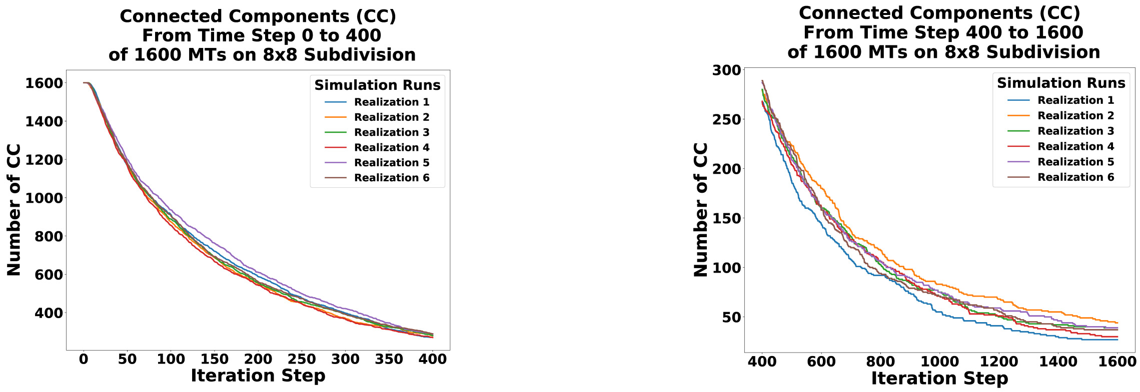


Figure 8.3: Six realizations of the change in connected components over time.

In figure 8.5 we see the long-term behavior of each node type in the system for all of the realizations. The plot shows how many of each node type we have in the system after every iteration. The top line in the plot is the total number of nodes. In each simulation, we start with 4,800 nodes (three for each starting MT - equation 8.2). In all of the end states of the



(a) Six realizations of the number of connected components from iteration 0 to iteration 400.

(b) Six realizations of the number of connected components from iteration 400 to iteration 1600.

Figure 8.4: Zoomed in plots of the beginning and end of six realizations of the number of connected components changing over simulation iterations, where one realization becomes a fully connected network

realizations we have over 17,000 nodes, indicating an average increase by at least a factor of three.

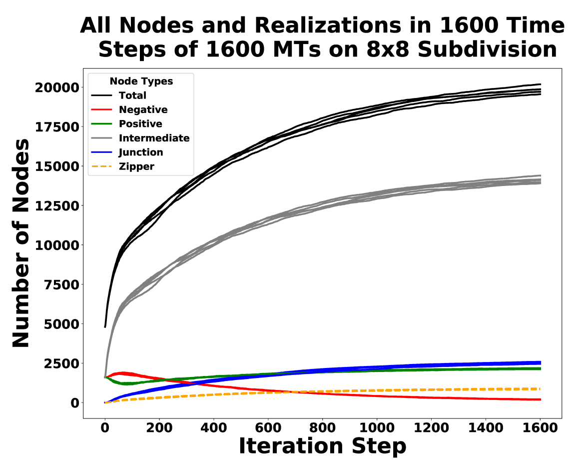


Figure 8.5: Plot of the long-term behavior of all node types, including all six realizations of the CMA DGG simulations.

The number of junction nodes in the system is different than the number of zippering nodes (on average three times as many junction nodes as zippering nodes), but they still follow a similar long-time trajectory as seen in figure 8.5. Since the zipper node dynamics are similar to the junctions, we only provide analysis for one. In figure 8.6, we can see how the zipper nodes change for each realization. Figure 8.6 indicates the number of zippering nodes increase rapidly at first and then begins to slow as we reach the long-term asymptotic behavior.

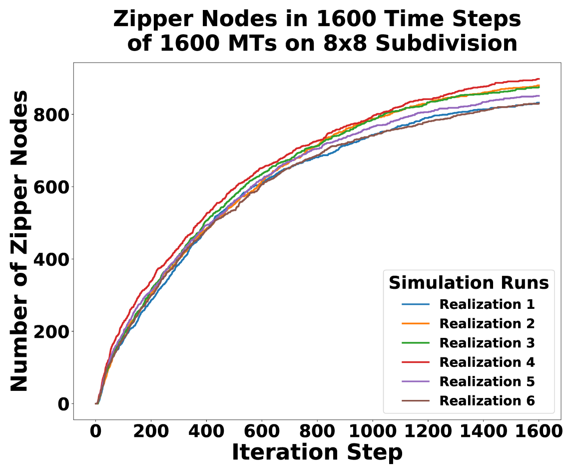


Figure 8.6: Plot of the change of zippering nodes over time for all six realizations of the CMA DGG simulations.

Figure 8.7 shows how the number of positive growth nodes in the system changes. The positive nodes are primarily responsible for the creation of new MT segments because of their participation in the growth rule, with the only other creation of new segments occurring during a junction/zippering rule firing. The rate of MT polymerization was set to be four times as fast as the rate of depolymerization. If the capturing boundary condition (BC) had not been imposed, the number of positive nodes in the system may have grown without bounds, since the rate of polymerization exceeds that of depolymerization. There is also a state change rule, which occasionally switches a negative end to a positive end or a positive end to a negative end.

Initially, we see a drop in the number of positive nodes at the beginning of the simulation. The cause is likely a combination of the state change rule, along with the capturing BC. Eventually, the growth recovers, and over time the positive nodes begin to again be captured by the BC or restricted in their directional dynamics due to the onset of zippering and junction formation. Any time a junction or zipper is formed, it creates a permanent and on average irreversible directional barrier for the growing end. The barrier is on average irreversible since there is no CMA DGG rule yet included to reverse the formation of a junction or zippering node. The growing rule does include a stochastic unit vector wobble, which means an MT could eventually circle around to form a new junction or zippering node, but that behavior is not likely. Thus, in general, the number of new positive nodes added into the system is expected to decrease over time and the total number of positive nodes is expected to reach a steady state depending on the particular realization, as seen in figure 8.7.

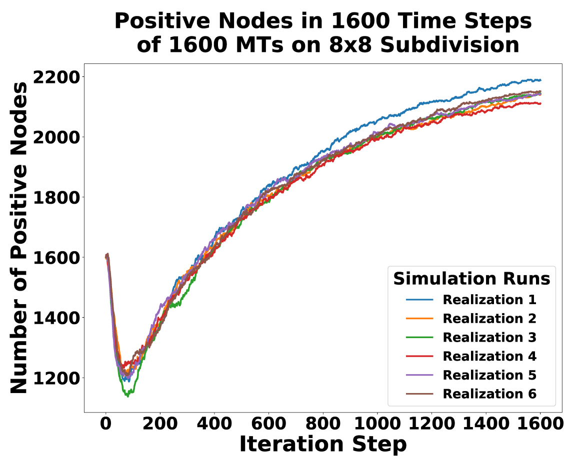


Figure 8.7: Plot of the change of positive nodes over time for all six realizations of the CMA DGG simulations.

The negative nodes in the simulations follow inverted dynamics when compared to positive nodes (figure 8.8). Each simulation starts with a fixed number of negative nodes that should decrease over time due to the BC and junction/zipper formation. We see this exact behavior but with a slight initial increase in negative nodes before long-term decay into a steady state. If the simulations ran longer, it is expected that no negative nodes would exist, because they state-changed to positive and got captured.

Finally, figure 8.9 is a plot of how the number of intermediate nodes change over time in each realization. In the CMA DGG simulations presented, the number of intermediate nodes in the system directly corresponds to the number of MT segments that exist. The growth rule

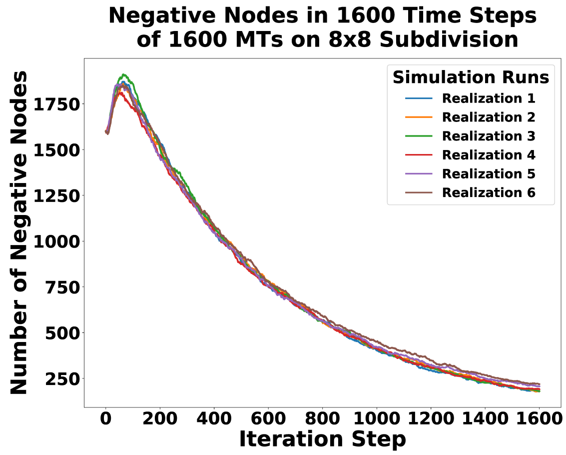


Figure 8.8: Plot of the change of negative nodes over time for all six realizations of the CMA DGG simulations.

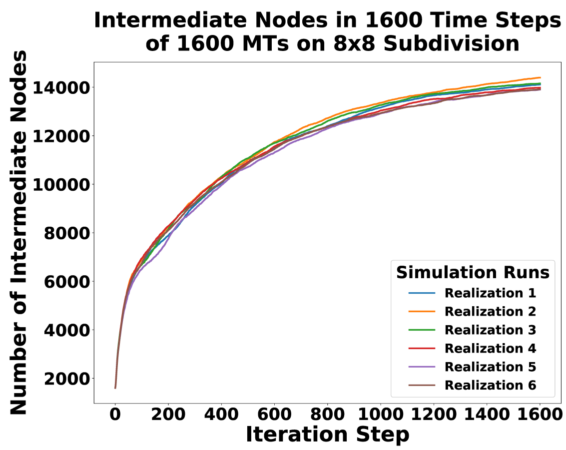


Figure 8.9: Plot of the change of intermediate nodes over time for all six realizations of the CMA DGG simulations.

functions to add more intermediate nodes; however, the zippering/junction rules and the capturing BC lock the system into place and slow growth. Consequently, network formation is encouraged, but longer-term growth is discouraged. So the more a network begins to form, the more intermediate nodes we get. As it forms, the addition of intermediate nodes decreases. Eventually, a steady state is reached and the number of intermediate nodes existing stabilizes.

Reactivity and Iteration Analysis

The MT dynamics of even a relatively simple system can be complicated. More complex dynamics require more computation and make performance a concern. We measure performance over the duration of one simulation step of τ_{ref} time, an *iteration*. We use reactivity per iteration as a quantitative measure of performance, where *reactivity* is the wall clock time of an iteration. Wall clock time is an appropriate measure because iteration time is correlated to the number of reactions occurring. For example, preliminary experiments with a grammar including a katanin-mediated severing rule had reactivity increase rapidly.

For the previous experiment of 1600 MT, the initial density was chosen to be low enough for each simulation to keep MTs in the starting state far apart from interacting with each other and to leave room for growth. Figure 8.10 shows how the system run-time dynamics change over time for different realizations. The plot is the actual real-world run-time per iteration. The reactivity plotted is the sum of the run-time for all of the geocells in an iteration. In general, the exact algorithm run within a geocell of a given operator-split dimension for any subdivision may not run in the exact amount of time as other geocells in that same dimension; however, in this experiment, they should on average because the MTs are initially uniformly distributed (figure 8.2a).

In figure 8.10, the reactivity of the system increases rapidly over most of the first 50 iterations. The reactivity observed is reflective of the dynamics as detailed in figure 8.5 and a consequence of MTs growing at a rate faster than they shrink. The peak reactivity occurs just before iteration 50. After the peak, the network begins to form as irreversible junction/zippering nodes are created and the reactivity of the system decreases. By around iteration 100 and onwards, the reactivity trends downward towards a steady state, which corresponds to the realized system dynamics in figure 8.5 and the network in figure (8.2b).

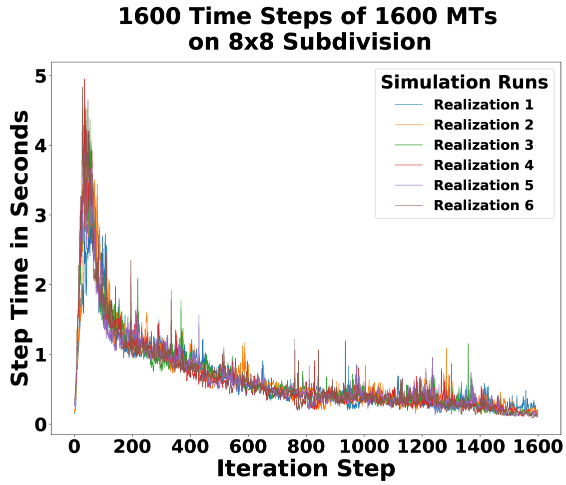


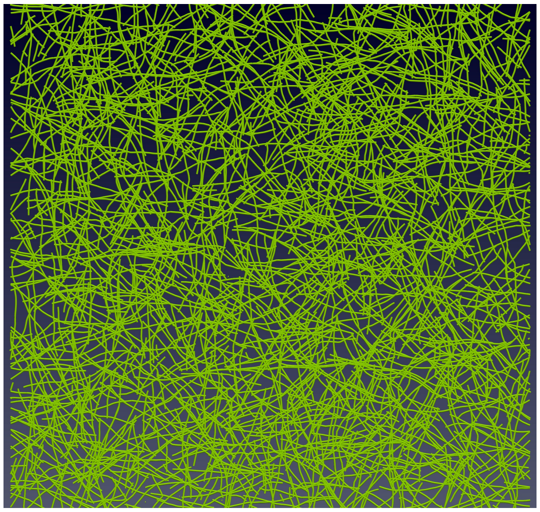
Figure 8.10: Plot of run-time per simulation iteration of six realizations of the CMA DGG simulations.

8.2.4 Experiment 2: Long-time Local Alignment

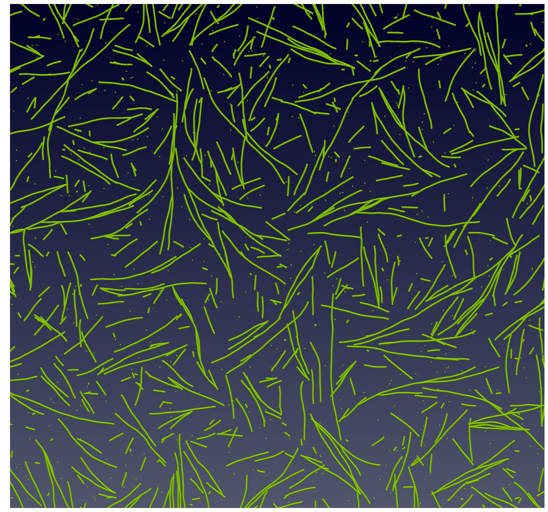
We use the same parameters and CMA DGG (figure 8.1) as experiment 1, but with the rate of crossover events lowered to near zero. Effectively, zippering and catastrophe events are favored. The starting state is the same as in figure (8.2a).

Figure 8.11 compares the ending state of the system with a low crossover rate to the system with the original crossover rate. In figure (8.11a) a highly connected network has formed, whereas in figure (8.11b) we can see that lowering the crossover rate leads to less connectivity, inhibits network formation, and reduces the number of surviving MTs. Significantly, figure (8.11b) exhibits localized alignment where the first experiment did not.

We compare how aligned the two ending states are by by computing an MT orientation correlation function defined as the squared cosine between the orientation of the MT segments, and average within bins of roughly constant distance. (This measure can be derived [73] as the trace of the product of the two rank-one projection matrices defined by the two unit



(a) Ending state of the simulation with high crossover ($t = 1600\ \tau_{ref}$). A highly connected network has formed.



(b) Ending state of the simulation with low crossover ($t = 1600\ \tau_{ref}$). Local alignment has occurred, instead of network formation.

Figure 8.11: Side-by-side comparison of the end states of the CMA DGG simulation of 1600 MTs on a 100x100 unit grid showing the effect crossover has on the system.

vectors; it is invariant to sign reversals of these unit vectors. We then have:

$$\mathrm{Tr}((u_i \otimes u_i) \cdot (u_j \otimes u_j)) = (u_i \cdot u_j)^2 = \cos^2(\theta_{ij}). \tag{8.3}$$

The function measures on average how “aligned” MT segments a distance away are from one another. The square is needed to remove anti-symmetry, since nearby MTs may be aligned but in anti-parallel directions, and anti-parallel alignment is not visibly distinguishable from parallel alignment in typical MT imagery. Values close to 0 using this measure indicates orthogonality and therefore no alignment, whereas values close to 1 indicate complete parallel or anti-parallel alignment. Typical intermediate values for lines at 45 degrees (equivalently 135 degrees) to one another are 1/2. Consequently, we subtract 1/2, the “uncorrelated” value, before averaging within distance bins of width defined by the reaction radius and fitting an exponential decay as a function of distance.

We can see the fitted correlation vs. distance functions in figure 8.12. When we fit an exponential decay $Ae^{-d/\xi}$ as seen in figure (8.11b), we get the fit $c(d) = 0.34e^{(-d/3.14)}$ with a mean absolute squared error (MASE) of 0.797. We initially have a high correlation and then a quick drop off with a correlation length $\xi_2 \approx 3.14$ with a standard error (SE) of 0.06 for the plot in figure 8.12. When we fit to an exponential decay on figure (8.11a), we get the fit $c(d) = 0.09e^{(-d/1.34)}$ and a MASE of 0.709. There is a much lower initial correlation and then a rapid drop off with a correlation length $\xi_1 \approx 1.34$ with a SE of 0.039 for the plot in figure 8.12. Even if we were to very conservatively zoom in on figure (8.11a) by a factor of 1.6 to equalize the number of MT segments in each window, the resulting correlation length of $\hat{\xi}_1 \approx 2.14$ is (as detailed in Appendix A) many standard deviations short of $\xi_2 \approx 3.14$; even more so for ξ_1 vs. ξ_2. Similar statistics, among many others (e.g. [102] for graph structure), could in the future be used to compare model-generated with biological-experiment imagery.

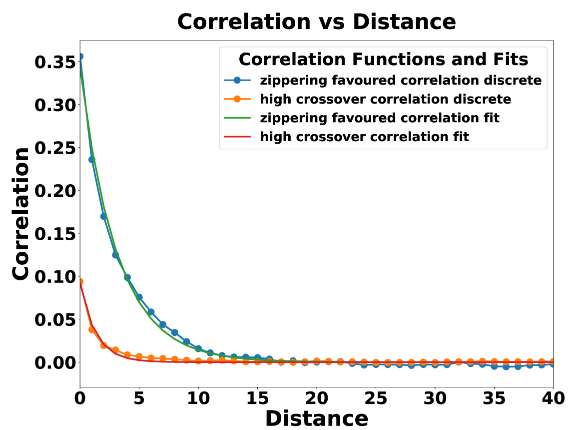


Figure 8.12: Sampled correlations of alignment over distance and their exponential fits of ending system states seen in 8.11

Our comparison indicates that zippering and catastrophe may lead to local alignment, partially supporting the "survival of the aligned hypothesis" [115]. The results seen in figure (8.11b) also look closer to what a real system of CMTs might look like. Alternatively, experiment 1 indicates that zippering, catastrophe, and high crossover lead to network formation. The addition of a selective katanin-mediated severing rule using an alternative hypothesis [35] also has the potential for global alignment of MTs in the system after the network has formed, but is a topic for future work.

8.2.5 Experiment 3: Approximate vs. Exact Performance

As a computational performance experiment, we started the simulation with an initial condition of 3200 microtubules randomly uniformly distributed across a 100x100 unit grid. We use the same grammar rules and parameters as the first experiment, along with identical node-capturing boundary conditions.

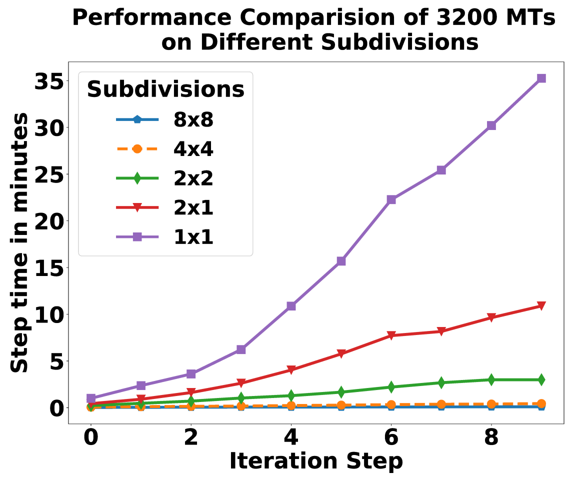


Figure 8.13: Plot of performance analysis of five separate simulation runs for 10 iterations.

We ran the simulation five times, once with no subdivisions and four times with different subdivisions as seen in figures 8.13 and 8.14. The first run, with no subdivisions, is the 1x1 domain. The 1x1 case does not use an operator splitting by geocell dimension and is equivalent to the Exact Hybrid Parameterized ODE SSA in Algorithm 1 [75].

For each step of τ_{ref}, the maximum time step that can be achieved is the adaptive step, $\mathbf{Max}\{\frac{1}{reactions}, \frac{l_{max}}{v_{max}}\}$. If we move beyond this step size, the ODE solver may miss reaction dynamics. As can be seen in figure 8.13, the exact algorithm is prohibitively slow because it must take more steps to solve the system. The step time of iteration 10 for the 1x1 subdivision in figure 8.13 reflects the slowdown and takes over 2,000 seconds or approximately 33 minutes on a single core of an Intel Core i7-7700HQ CPU @ 2.80GHz. This is not practical for long-term simulations with serial computation and server-grade CPUs would not fare significantly better. In parallel computation, similar bottlenecks for individual geocells would show up if the experiment was scaled up so that each of the 8x8 subdivisions was the same size as a 1x1 subdivision; however, there would still be the benefit of running parallel computations.

The subdivisions 2x1, 2x2, 4x4, and 8x8 each show a significant speedup over the original 1x1 Exact SSA. Each of these runs uses Algorithm 4. In the 2x1 case, we see a speedup (caused by subdividing the domain) of around a factor of four instead of a factor of two. The difference may be because larger steps can be taken and the search space is smaller. In the 2x2 case, it becomes a factor of twenty. In figure 8.13 the 8x8 and 4x4 cases look similar due to the time scale, however, there is also a significant speedup. There may be diminishing returns to scale beyond 8x8, for our initial condition of 3200 MTs. We include the semi-log plot (figure 8.14) of the same data in figure 8.13 to make the step time differences more clear.

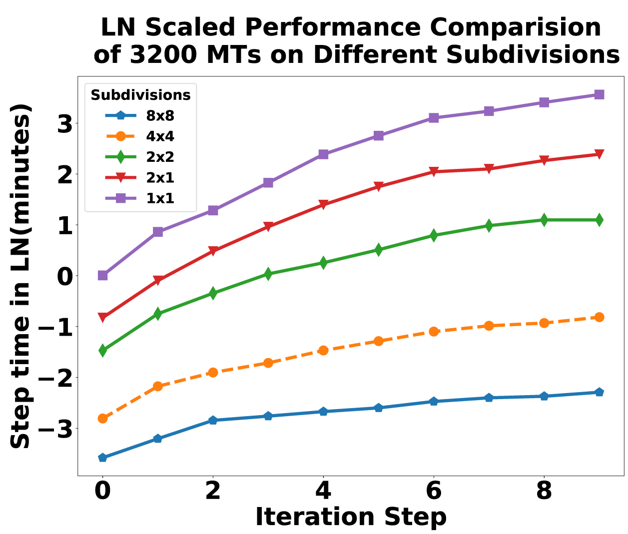


Figure 8.14: Natural log scaled plot of performance analysis of five separate simulation runs for 10 iterations.

From these results, we find that the approximate algorithm achieves a significant speedup over the exact algorithm. The resulting speedup comes with a few trade-offs. First, we get the speedup at the potential cost of accuracy due to some reactions firing out of order. Second, there is a saturation speedup point for every system. Simulation cells can only be minimized to a factor of the "fall off" distance and still need to maintain well-separatedness. Finally, the practical lower limit of the ODE solver step size depends on the dynamics being simulated. Whereas simulation speed is limited by cell size and ODE solving step size, there is no such limit on scalability - making this algorithm appropriate for modeling very large systems, or smaller systems in greater detail.

8.3 Conclusion

DGGs can be used to simulate complex biological systems using a simulation algorithm derived from their corresponding master equation. We have introduced an initial implementation of the approximate algorithm, for spatially embedded and local DGG dynamics, which achieves improvements to performance over an exact algorithm at some potential cost of accuracy. We demonstrated the speedup in simulated dynamics of a Dynamical Graph Grammar (DGG) model of a plant cortical microtubule array, which forms a cytoskeletal network and can exhibit localized alignment. The work in this chapter represents in initial implementation of Algorithm 1, but comes with the cost of being rigid and inflexible in its design. The model in the next chapter makes use of the flexible, customizable, and more efficient DGGML. Nonetheless, the work in the chapter lays down the foundation for building an even more complex CMA model.

Chapter 9

Revisiting the Cortical Microtubule Array Model with DGGML

9.1 Introduction

In the previous chapter (Chapter 8), we introduced a simplified model for a CMA DGG using the precursor [73] to DGGML. Building upon this model, we have further refined the model to be more biologically realistic. This refinement includes the addition of new rules, which effectively demonstrate the capabilities of DGGML. On the modeling side, we use these new rules to investigate the mechanisms underlying the formation of specific array patterns of cortical microtubules in the absence of other cell information by incorporating well-known dynamics observed in the outer periclinal face of plant cells.

In this chapter, we conduct a novel simulation study using the new DGG model to investigate the plausibility of various dynamics in driving self-organization. We vary parameters such as periclinal face geometry and other boundary conditions. Additionally, we briefly explore the impact of the crossover rule on global alignment. Our findings also underscore the role longer-lived aligned MTs [115] play in suppressing the survival of shorter, unaligned MTs.

The study is also designed to explore the long-time behavior emerging from the addition of the new rules and determine the influence geometry and underlying mechanisms have on the selection of alignment axes. Instances where alignment fails to emerge indicate a reevaluation of parameters is required or signal there may be the unaccounted-for biological mechanisms driving array alignment that does not exist in the the current DGG model.

Finally, the core of the work in this chapter assumes the periclinal face of a plant cell is either a square or rectangle with sharp edges (high curvature) during the transition from the periclinal to anticlinal face. We also add in an approximation of the proteins associated with attaching/linking microtubules to the cell cortex (cytoplasmic linker associated proteins (CLASP) [113]) on the edges to generate different boundary conditions to broaden the types of array alignment dynamics we can observe.

9.2 Background

The mechanics of plant morphogenesis are complex and play out on many different scales of space [28], and these mechanics are well suited to be used to develop DGG models [76]. The reorganization of the cortical microtubule array (CMA) [106] is one part of the scale hierarchy. The CMA is significantly complex and there are still many outstanding questions [39], such as what general principles govern the organization of cortical microtubules into functional patterns?

The Periclinal CMA (PCMA) model presented in this chapter approximates the periclinal face of a cell, which is the face parallel to the organ's surface. The anticlinal face is perpendicular to it [1]. A common approximation of the shape of a plant cell for CMA simulations is a cube [3] or polygonal prism, or a curved surface such as a cylinder [114, 42]. In our case, the cell is approximated to be a rectangular prism, which includes the cube and could be extended to other polyhedral prisms. Therefore, the approximated cell has six faces, 12 edges, and 8 vertices. Using this notion of prisms, we choose the base faces of the prism to be the top and bottom periclinal faces of the cell. During *in vivo* experiments, the top periclinal face is easier to observe, so we chose this location as a reasonable representation of the simulation space for our model.

CLASP (Cytoplasmic linker associated protein) is a well studied microtubule associated protein (MAP) that is found in different types of plant, fungal, and animal cells. In the plant cells of Arabidopsis it has been observed to facilitate transitions between CMA patterns [113]. The localization of the Arabidopsis thaliana CLASP (AtCLASP) protein to specific cell edges was shown experimentally and computationally to mediate MT polymerization when encountering cell corners of high curvature and the selective localization biased the alignment of the CMA [5]. In our model, we use this observed behavior to make broad idealized assumptions about the effects it has on the boundaries.

In terms of CMA orientation, a common hypothesis for MT array pattern formation is "survival of the aligned" [115], where lower rates of crossover and higher rates of zippering lead to an organized orientated array and MTs that are created but do not align do not survive. A common pattern in the CMA is observed band formation [99] where several bands form along the cell face's horizontal or vertical axis, indicating the network has fully wrapped around the surface in that alignment configuration. There is also a "picket fence" phenomenon, whereby MTs forming and aligned to existing MT orientation in the anticlinal faces are oriented to enter the periclinal faces at angles perpendicular to the edge with some variance [120]. The entering MTs, to an extent, are aligned in parallel arrays in the anticlinal plane [106]. A visualization of the approximation we make can be seen in figure 9.1. There is also evidence showing that light can have an effect on the reordering of the array and can result in an influx of MTs from the lateral anticlinal walls in light-sensing mutants and hormone stimulated plants [97].

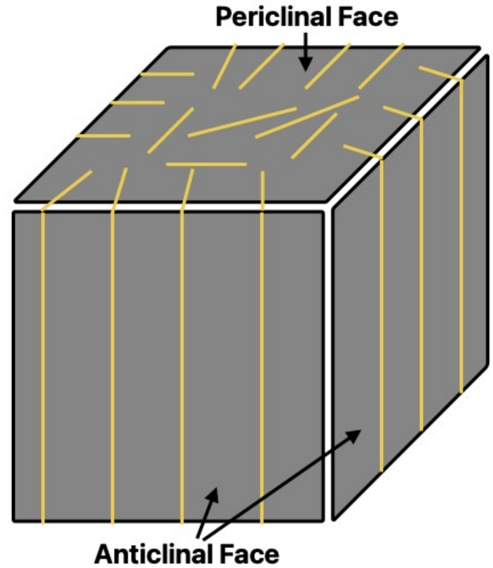


Figure 9.1: Visualization of our approximation of the cell as a polyhedral prism (top: periclinal, sides: anticlinal) with our approximation of the "picket fence" phenomena, where microtubules (MTs) in the anticlinal faces are aligned in a perpendicular array pattern [106]. Newly created MTs forming in the anticlinal face or those aligned to existing MT orientation in the anticlinal faces are likely oriented to enter the periclinal faces at angles perpendicular to the edge with some variance [106].

9.3 Modified CMA Grammar

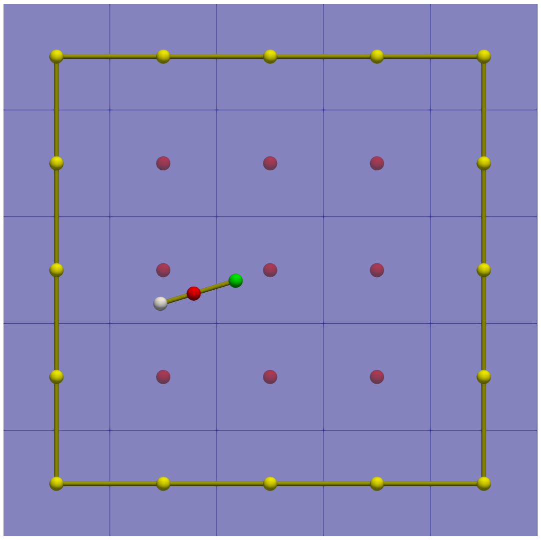
The Periclinal CMA (PCMA) DGG used in this chapter extends, modifies, and refines the grammar used in Chapter 8 with rules found in Appendix A. A full listing of all the grammar rules for the PCMA along with their corresponding parameters can be found in Appendix B. We also note that the CMA model used artificial parameters and this new PCMA model uses more bio-inspired parameters, so the parameter regimes are not clearly comparable.

In this section, we highlight several of the grammatical distinctions. For example, we have added an alternate zippering rule that does not zipper directly into the extant MT graph segment (i.e. have the snap on action) but rather functions as an entrainment rule with a stand-off separation distance [25]. By introducing this alternative zippering rule, MTs are now capable of "piling up" and forming high alignment on the local scale for the array. A zippering guard rule has been added to act as an approximation to bundling proteins [39], to prevent a funneling effect that can otherwise occur. Essentially, another MT can enter the gap between zippered MTs unless there is a rule to prevent it.

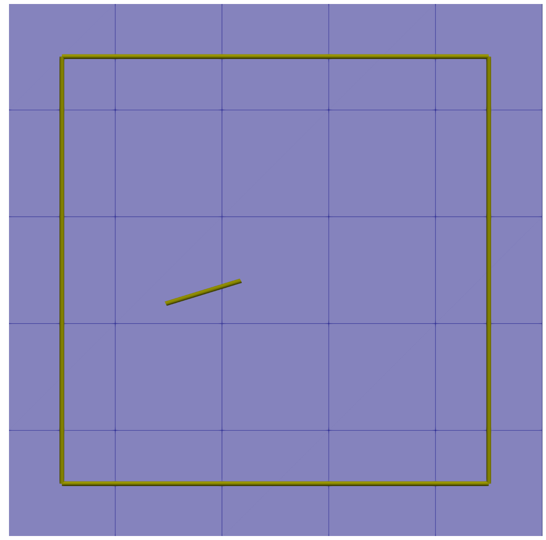
The previous grammar started with a fixed number of MTs that did not change over time. The PCMA now includes nucleation and destruction of MTs to simulate full depolymerization and removal from the system. For nucleation, we have included a grammar rule for uniformly random nucleation of MTs in the background using a nucleation rate [3] functioning independently from the existing array of MTs. We have not, however, included a rule for MT dependent nucleation, where new MTs are nucleated on existing MTs [82, 83] or MT creation by means of severing at crossover site [125]. In our previous CMA model [73] the closest comparison to nucleation was what occurred when a MT depolymerized. In that case, it would enter a dormant state until a rescue occurred to effectively change the state of one of the ends back to a growing node.

We have also modified how the boundary works. In the previous model, the boundary worked by implicitly exploiting the functionality of the simulator by ignoring any MTs that moved outside the simulation space, effectively functioning to capture the MTs. In the new model, the boundary is now explicitly defined as a graph. An example of this can be seen in figure (9.2a). To prevent any MTs from leaving the interior of the space bounded by the boundary graph, an induced catastrophe rule has been added. Alternatively, the ODE in the deterministic growth rule could have an additional constraint that decays the velocity to zero when a boundary is reached, which would work as another route for using the DGG to specify boundary conditions.

The Periclinal CMA (PCMA) also includes new special boundary rules, to approximate the



(a) System after a short time of simulation. Nucleation points are shown in the interior and the exterior graph is the boundary graph. A single microtubule exists within the simulation box.



(b) Same system as figure (9.2a) after a short time of simulation. The visualization is without nodes and makes the structural model clearer.

Figure 9.2: Side by side example of what simulated system looks like, and what we mean by boundary graph and nucleation points.

effects of CLASP in the case of high curvature on the boundary between face transitions. Biologically, there are different types of CLASP, but for the simulation study, we use CLASP as a means to explore one of its functions, which is the ability to mediate MT polymerization from the periclinal into the nearest anticlinal face in the context of an idealized plant cell. In reality, CLASP likely can localize to only parts of the edge [3], but we make the assumption that CLASP has localized uniformly on the edge. When an MT reaches the boundary, two outcomes may occur: the MT will cross over to the periclinal plane mediated by CLASP or it will collide and induce catastrophe.

MTs can also fully depolymerize from the periclinal plane into the anticlinal, and this is modeled as a rewrite rule that removes the MT attached to the boundary from the system. There is also the ability to allow the MT to statistically unstick from the boundary which models the collision of an MT entering the anticlinal face, and encapsulates the unknown dynamics that play out in the "hidden" plane, one of which is the "picket fence" phenomena mentioned in the background section and illustrated in figure 9.1.

Finally, a noticeable distinction between the CMA model in Chapter 8 and the PCMA is the lack of a wobble added to the stochastic growth rule. Due to the inclusion of the new zippering rule, wobble could only be added with the addition of more yet-to-fully-be explored modeling rules for bundling proteins. As a result, we have opted to keep the MTs straight for simplicity; however, every MT is still made up of many smaller segments (as in figure 9.2a).

9.4 Results: Boundary Conditions and Alignment

For the Periclinal CMA, we explore the effects boundary conditions have on the alignment axis. In the previous chapter, we ran experiments where the boundary had an implicit capture condition. Effectively, any MT that made it to the boundary was put into a pause state. In this chapter, we explore two categories of periclinal domain geometry: squares and rectangles. Within each of these categories, we run six different computer experiments.

In the first experiment, we have a collision-induced catastrophe (CIC) boundary, where any MT that collides with the boundary will begin to rapidly depolymerize. The CIC boundary is used as a baseline for the default behavior for predicting the alignment axis and the remaining five experiments investigate perturbations of the system through means of rule changes and parameter variations. In the second experiment, we enable the CLASP-mediated crossover boundary conditions to stabilize the array at the boundaries and explore the effects of high MT crossover rate on the respective array. By high we mean we change the initial rate found in Appendix B from 200 to $8,000$ (a fortyfold increase), which makes crossover nearly as likely as CIC (which has a default rate of $12,000$). In the third experiment, we introduce an influx of MTs from the boundary and increase the initial rate of 0.001 to 0.008, meaning new MTs enter the system 8 times as often. Finally, in experiments 4 through 6, we again have the CLASP-mediated boundary crossover enabled, but we accordingly vary the range of angle of entry and exit of MTs from the perpendicular to the boundary in increments of $15°$: $[-30°, 30°]$, $[-45°, 45°]$ and $[-60°, 60°]$. In experiments 4 through 6, all other parameters and rates are the default.

In total, there are twelve different experiments, with half being run for each respective domain geometry. Each of the individual experiments is run as ensembles of sixteen simulations, culminating in a total of 192 realizations run using DGGML. Unless otherwise specified, the experiment will use the default parameters. For a complete overview of the grammar and the default parameters, please see the appendix.

Each of the simulations is run for two hours of biological time, and checkpoints are taken every 24 seconds of that time. At each step, the angles of all intermediate, retracting, and growing MT nodes are collected and projected to the right half of the plane with $0°$ being the positive x-axis ($-90°$ (negative y-axis) to $90°$ (positive y-axis)). These projected angles are binned by increments of $15°$, creating a histogram [57]. The correlation length vs. distance histogram discussed in Chapter 8 is also computed at each checkpoint. From the correlation length vs. distance histogram, we take the average over the first third of the correlation vs distance function to get the local correlation average, and over its entirety to get the global average. While there are other measures of alignment [11], the local and global mean correlations work as a sufficient measure to quantify alignment when combined with the histogram.

9.4.1 The array orientation of square domains is multi-modal

The area of the simulated domain is a $5\mu\text{m} \times 5\mu\text{m} = 25\mu\text{m}^2$ and is initially empty. New microtubules can be created by background nucleation or enter from the boundary when

CLASP rules are enabled. For each of the experiments, we have $N = 16$ samples of the local and global correlations. In figure 9.3 we present the square CIC boundary experiment data used to calculate the mean local and global correlations shown in figure 9.4.

For the local correlation (left in figure 9.3), we have plotted all individual samples from the square domain with the collision-inducing catastrophe (CIC) boundary condition experiment. From these samples, the mean and standard deviation are computed and plotted. Using the mean, a best-fit logarithmic curve ($f(t) = a\log(bt) + c$) is estimated and works as a sufficient fit for the 120 minute range of interest. The same approximation is made for the global correlation. The plots for all remaining square experiments can be found in Appendix B.

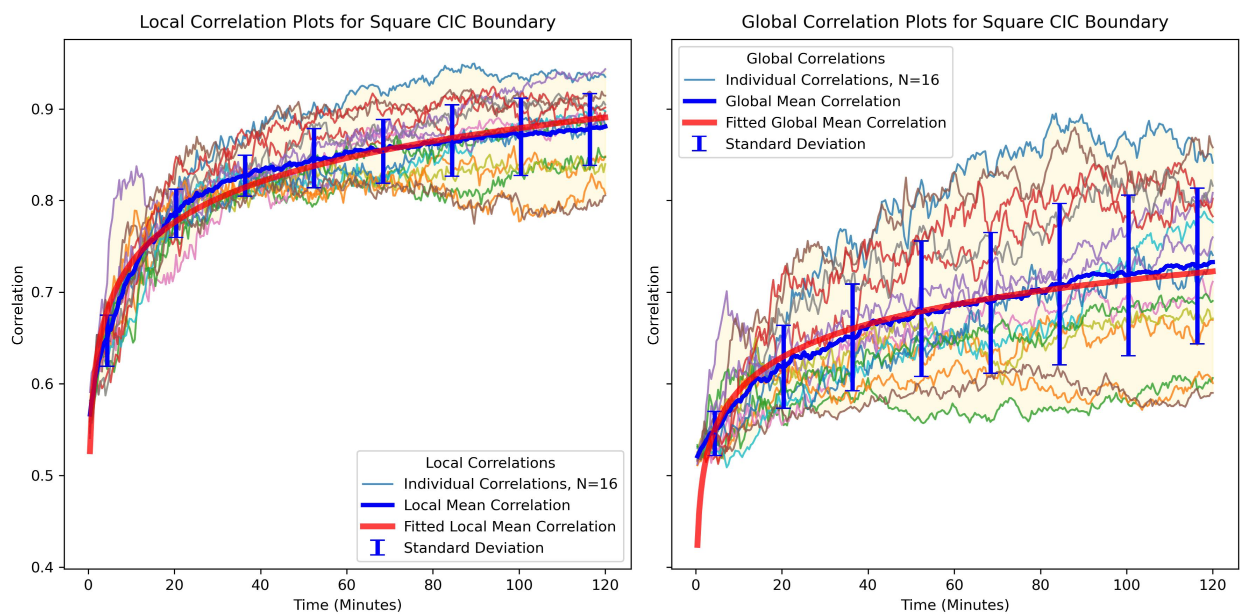


Figure 9.3: For the square experiment with a CIC boundary, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

In figure 9.4 the local and global correlation mean fits are plotted for each of the experiments. The initial step is omitted which has undefined local and global correlation, since the system is empty. For all simulations of the square domain, except CLASP mediated boundary crossover and the high MT crossover experiment (which remains near the fully uncorrelated value of 0.5), both correlation measures reach a steady state after about one hour. These simulations all become locally aligned, but according to the global correlation measure appear not to ever become fully aligned.

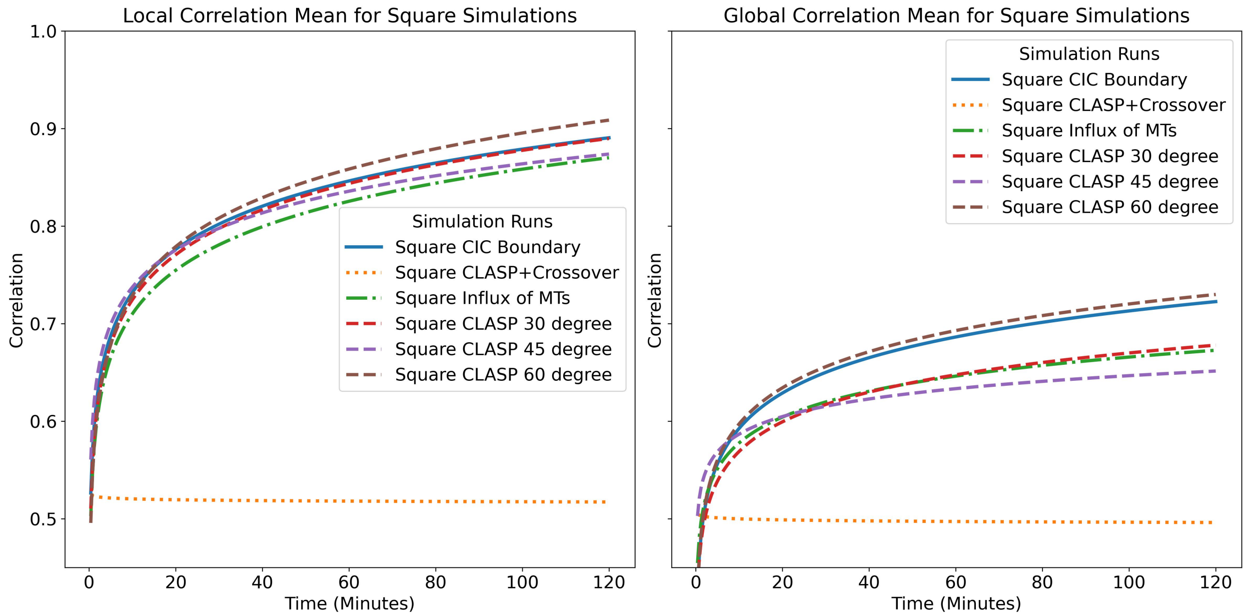
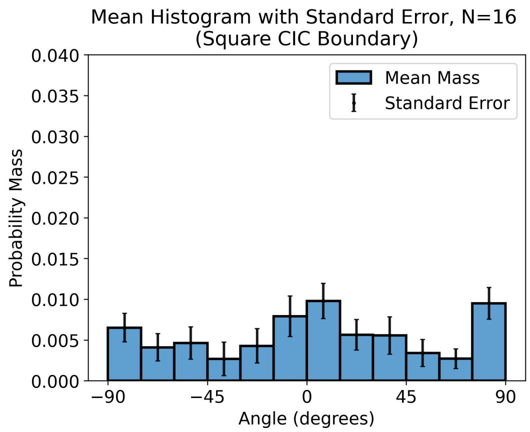


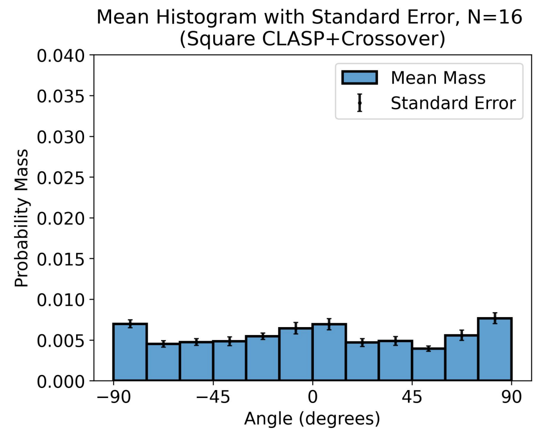
Figure 9.4: The plot contains the a best-fit logarithmic curves ($f(t) = a\log(bt) + c$) for the 120 minute range of interest fitted to the local and global mean correlation over time for each of the square simulation experiments. All simulations start empty and have zero correlation.

In figure 9.5, we have the histograms for the CIC boundary (figure 9.5a) and the CLASP with Crossover (figure 9.5b). In figure (9.5a), the most likely to occur angles are clustered around $-90°$, $0°$, and $90°$ with relatively small error bars. That is, in the absence of any boundary condition aside from collision, the network will tend to align in horizontal and vertical directions. It is also possible for it to align in the diagonal direction. When we introduce the CLASP rules for the boundary and allow for a high rate of crossover (change the default rate from the low value of 200 to the high value $8,000$) the orientation of the array quickly becomes uncorrelated and the behavior persists for the remainder of the simulation, as evident by the dotted line with near constant long time behavior in figure 9.4. The effect of the lower global and local correlation can be observed in figure (9.5b), where the computed histogram is more uniform.

In figure 9.6 we have the ending state mean histogram for the experiment with an influx of



(a) Baseline histogram estimating the probability mass function for array orientation for the square domain with CIC boundary, including standard error bars and averaged over 16 runs.

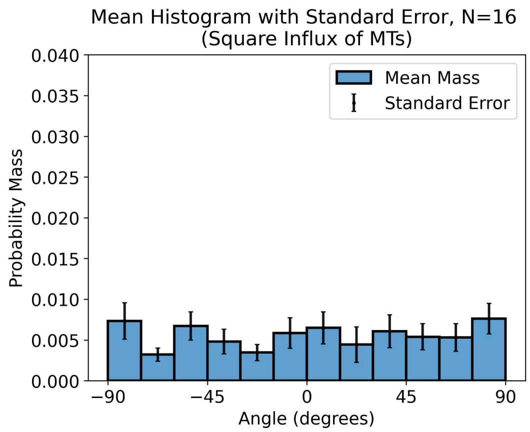


(b) Histogram estimating the probability mass function for array orientation of square domain with a CLASP boundary condition and high crossover, including standard error bars.

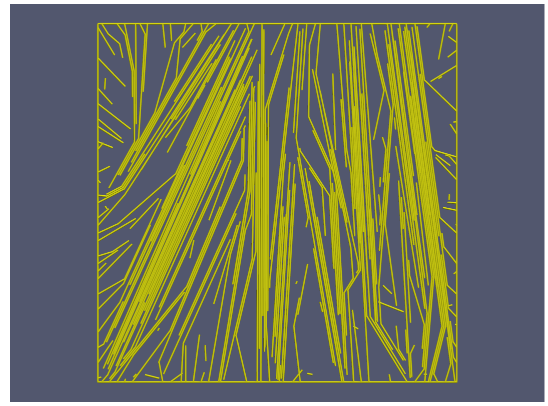
Figure 9.5: Histograms of the mean angular orientation for the arrays at $t = 120$ minutes for the square CIC boundary and CLASP+Crossover and averaged over 16 runs.

MTs at the boundary (figure 9.6a) and a sample image of an ending state used to compute the histogram (figure 9.6b). In figure (9.6a), where the influx of MTs occurs at the boundary, we see a seemingly more uniform histogram with larger error bars. The key distinction between figure (9.5b) and (9.6a), can be observed from the local and global correlation data in figure 9.4. While they both have fairly uniform angle distributions, they have very different local and global correlations. The simulations with the influx of MTs on average become highly correlated at a local length of approximately 1.67μm and somewhat correlated at the global scale. The CLASP+Crossover experiment stays uncorrelated the entire simulation. In figure (9.6b), we show a sample of the simulation of the high influx exhibiting vertical alignment. The local correlation = 0.93, and global = 0.8 in the sample and is quite different than what the angular orientation histogram initially appears to be indicating. The histograms for the other CLASP experiments look similar and the behavior follows a similar pattern.

Looking deeper, we analyze some of the samples of the CLASP (30°, 45°, and 60°) experiments. Kernel density estimation (KDE) is a method for estimating the probability density



(a) Histogram estimating the probability mass function for array orientation for the square domain with an influx of MTs, including standard error bars and averaged over 16 runs.



(b) Sample of a network used to compute the PMF in figure (9.6a), with local correlation = 0.93, and global = 0.8

Figure 9.6: Histograms of the mean angular orientation for the arrays at t = 120 minutes for the square influx of microtubules averaged over 16 runs, along with an ending state sample of the influx experiment.

function of a random variable [103]. The Gaussian kernel is a commonly used kernel function in KDE, which assigns weights to data points based on their distance from a given point. The Scott bandwidth is a parameter used to control the smoothness of the estimated density, with larger bandwidths resulting in smoother estimates. For this study, the KDE is used as a qualitative way to visualize and reason about the orientation of the array. A closer look at how KDE estimates a density function using the histogram from an individual CLASP 30° experiment can be found in figure 9.7. Samples using the KDE for the square CLASP 30°, 45°, and 60° experiments are presented in figure 9.8 and figure 9.9.

In figure (9.8a), we see the results for the CLASP 30° experiment. Sample 5 from the figure (9.8a) has the distinct characteristics of vertical alignment, which is represented by peaks at both ends of the plot. However, in sample 7 of the figure (9.8a), the PDF indicates an array with mixed orientations. This can be effectively visualized in figure (9.8b). Looking towards the center of the image we see a mix between horizontal alignment and the partial formation of a diagonal alignment from upper left to lower right. These two alignments

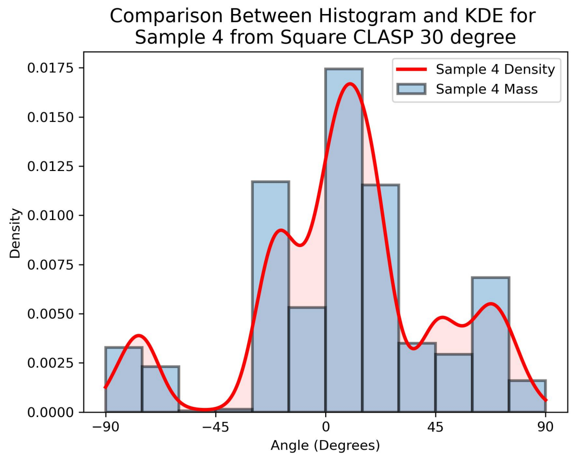
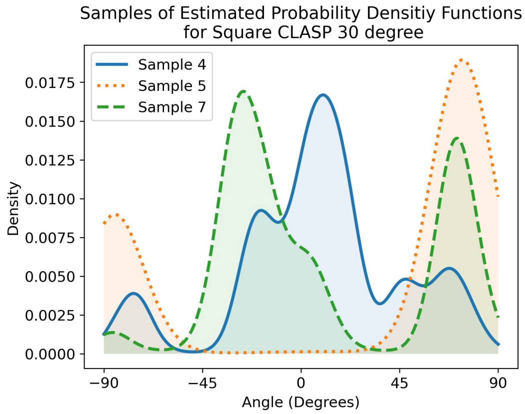


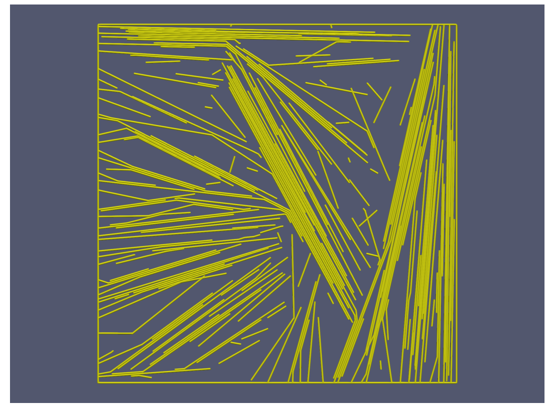
Figure 9.7: A closer look at the KDE with a Gaussian kernel and the Scott bandwidth [103] of the orientation PDF from sample 4 of the CLASP 30° experiment, along with the histogram it was estimated from. In this case, the resulting density is a close match for a qualitative representation of the shape of the histogram.

are characterized by the right-shifted central peak in the plot. The remaining orientations angled around the border are effectively represented by the peaks at the ends. Cases like this occur when no central aligning axis can win out. In line with the "survival of the aligned" hypothesis, only aligned MTs tend to survive, which strengthens the staying power of these multiple locally aligned arrays. Once a mixed state such as the one in figure (9.8b) is reached, the system may stay in this state indefinitely.

In the examples from the CLASP 45° experiment (figure 9.9a), we observe a situation where there are three identifiable clusters of orientations: vertical, diagonal left, and diagonal right. In the examples for the CLASP 60° experiment (figure 9.9b), we observe a situation where there are three identifiable cluster orientations: horizontal, diagonal left, and diagonal right. However, in all the plots of figure 9.9a and figure 9.9b there are multiple peaks with wide bases, which indicates that there may be mixed orientations as well. Returning to the seemingly uniform histogram in figure (9.6a), we can see that the histogram is correctly uniform; but, not because there is no alignment. The reason for the uniform behavior is there appear to be several underlying modes of orientation that are being averaged out. If enough samples were taken and then clustered, we might be able to recover the underlying



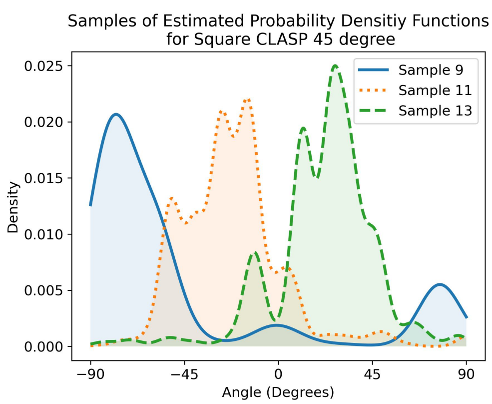
(a) Three PDFs fit using kernel density estimation with a Gaussian kernel and the Scott bandwidth [103] from the histograms computed for individual samples of the CLASP 30° experiments.



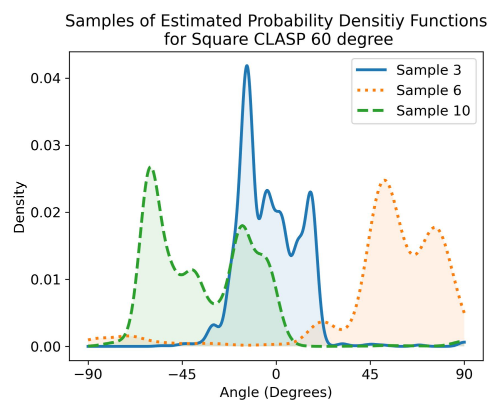
(b) Sample 7 from (9.8a) with local correlation = 0.84 and global correlation = 0.51

Figure 9.8: An example with three samples of an estimated orientation PDF from the CLASP 30° experiments, along with the ending state of sample 7 from the CLASP 30° experiment.

modes of the angle orientation for the square geometry.



(a) Three PDFs fit using kernel density estimation with a Gaussian kernel and the Scott bandwidth [103] from the histograms computed for individual samples of the CLASP 45° experiments.



(b) Three PDFs fit using kernel density estimation with a Gaussian kernel and the Scott bandwidth [103] from the histograms computed for individual samples of the CLASP 60° experiments.

Figure 9.9: Two examples with three samples of estimated orientation PDFs from the CLASP 45° and 60° experiments

Through each of these experiments, we were able to find that the high crossover rates lead to no defined axis of alignment, and the remaining simulations lead to multiple defined and sometimes mixtures of global and local alignment. Without additional mechanisms of control, the rules combined with a square domain do not lead to a preferred axis of alignment. However, if they do align it will be in the horizontal, vertical, or diagonal directions. Therefore, the array orientations of square domains are actually multi-modal.

9.4.2 Boundary conditions reorient arrays in rectangular domains

The area of the simulated rectangular domain is initially empty and has dimensions of $8.33\mu\text{m} \times 3\mu\text{m}$ and is set to be equal to the area of the square simulation domain. For each of the experiments, we have $N = 16$ samples of the local and global correlations. In figure 9.10 we present the rectangular CIC boundary experiment data used to calculate the mean local and global correlations shown in figure 9.11.

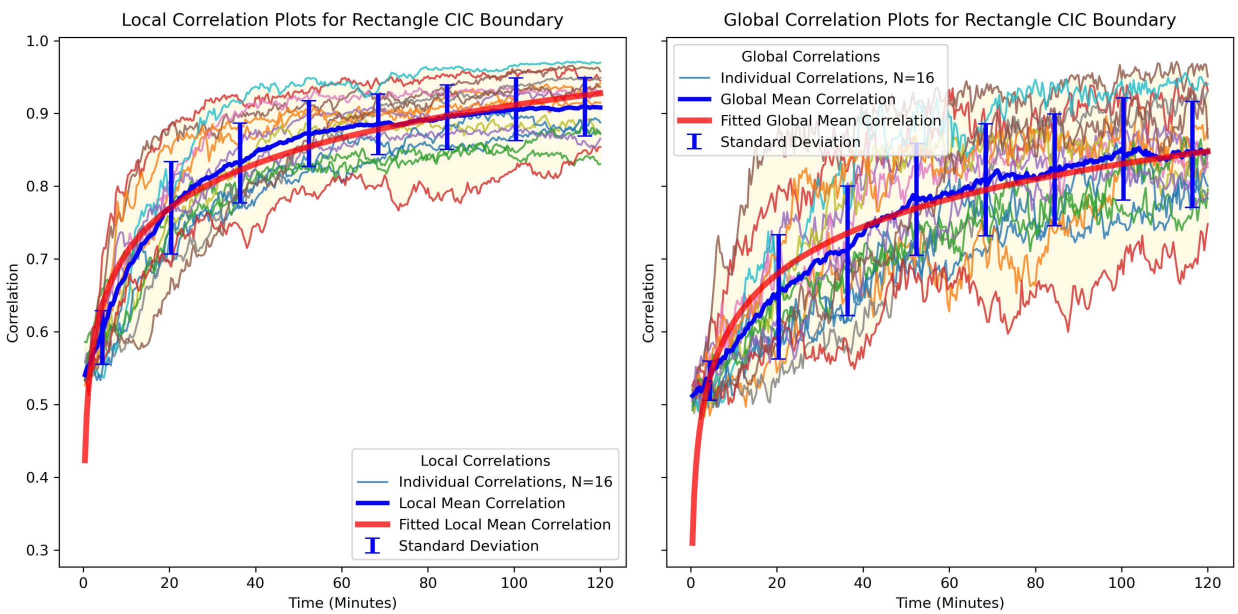


Figure 9.10: For the rectangular experiment with CIC Boundary, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

For the local correlation (left in figure 9.10), we have plotted all individual samples from the rectangular domain with the collision-inducing catastrophe (CIC) boundary condition experiment. From these samples, the mean and standard deviation are computed and plotted. Using the mean, a best-fit logarithmic curve is estimated ($f(t) = a\log(bt) + c$) and works as sufficient fit for the 120 minute range of interest. The same approximation is made for the global correlation. The plots for all remaining rectangular experiments can be found in Appendix B.

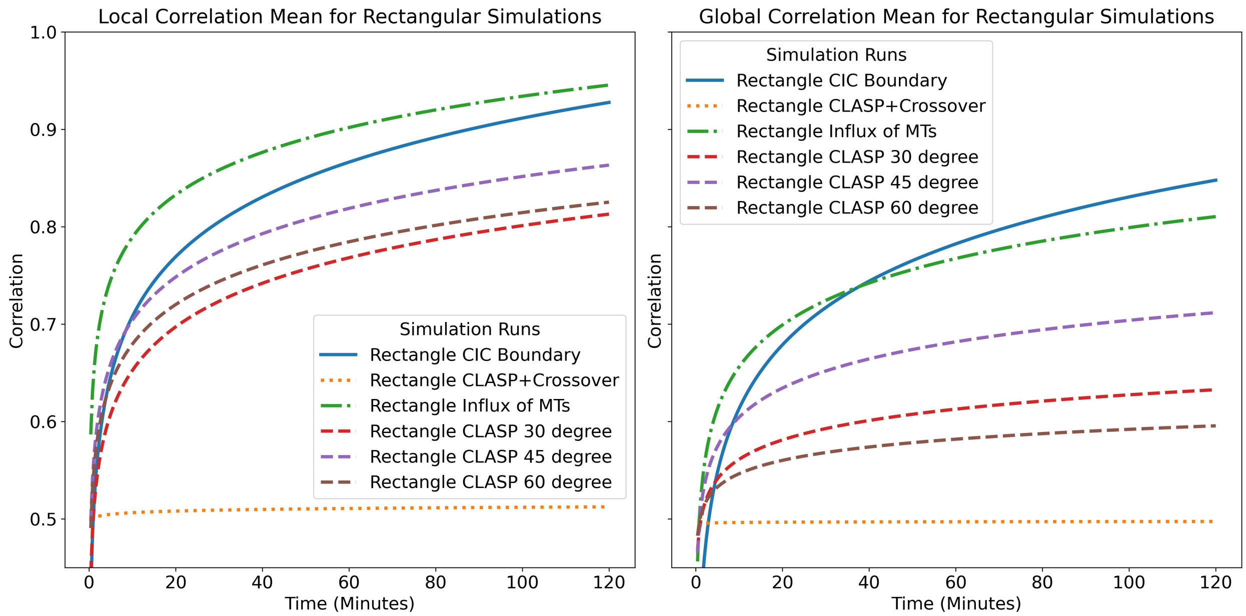
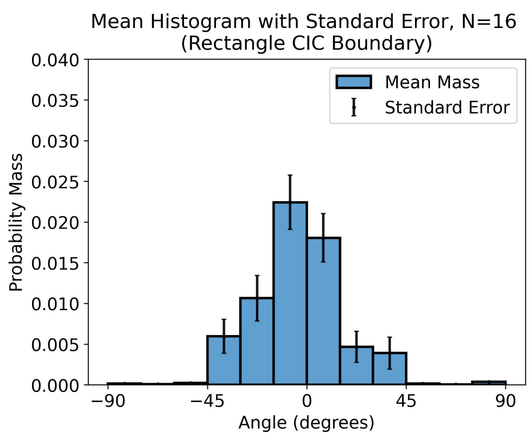


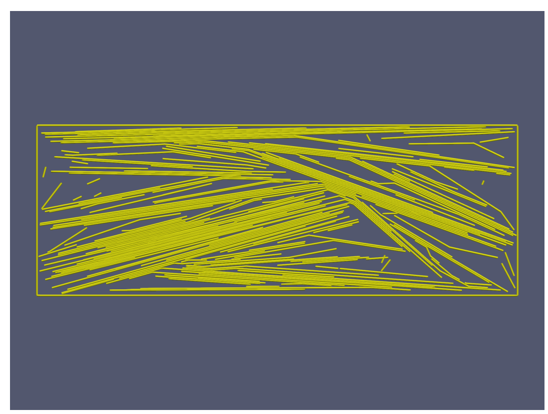
Figure 9.11: The plot contains the best-fit logarithmic curves ($f(t) = a\log(bt) + c$) for the 120 minute range of interest fitted to the local and global mean correlation over time for each of the rectangular simulation experiments. All simulations start empty and have zero correlation.

In figure 9.11 the local and global correlation mean fits are plotted for each of the experiments. The initial step is omitted. It has an undefined local and global correlation since there are no MTs present in the system. For all simulations of the rectangular domain, except CLASP-mediated boundary crossover and the MT crossover experiment with the higher rate of $8,000$ (which remains near the fully uncorrelated value of 0.5), both correlation measures reach a steady state after about one hour. The behavior is similar to the square case. On average these simulations all become locally aligned. According to the global correlation measure, only the collision-induced catastrophe (CIC) boundary experiments and the influx experiments become correlated enough to be considered globally aligned. Crossover is globally uncorrelated, and the remaining experiments show a mixture of alignments.

The defining alignment behavior for the CIC boundary experiments is illustrated in figure 9.12 by a mean histogram and a sample from the data used to compute it. The histogram in figure (9.12a) prominently centers around 0°, which indicates a high occurrence of angles oriented in the horizontal (long axis) direction. The sample array in figure (9.12b) is mostly aligned horizontally, with some deviation towards the diagonals, which can be seen in the histogram as well. The local alignment is 0.93 and the global is 0.83. What we have found is that given enough time, under these conditions, the array orientation is uni-modal, and oriented horizontally i.e. the long direction of the rectangle. Given a CIC at the boundary, the array likely orients horizontally because the longest axis allows the MTs to survive for longer.



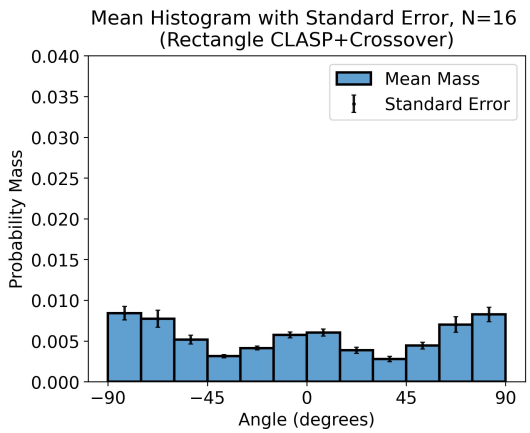
(a) Histogram estimating the probability mass function for array orientation of rectangular domain with CIC boundary, including standard error bars and averaged over 16 runs.



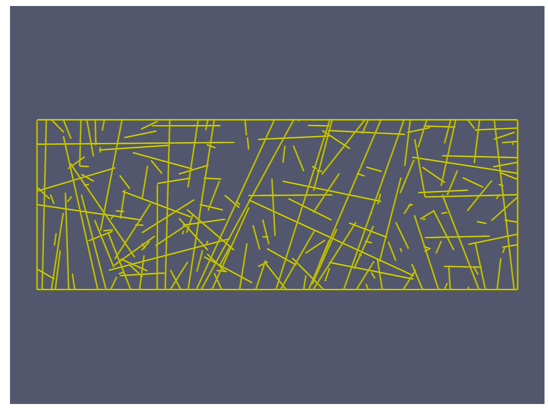
(b) Example of one of the samples used to calculate the array orientation. The array is oriented horizontally, with some variation. Local correlation = 0.93 and global correlation = 0.83.

Figure 9.12: Histogram of the mean angular orientation for the arrays at t = 120 minutes for the rectangular case with a CIC boundary averaged over 16 runs, along with an ending state sample of the CIC experiment.

Given the uni-modal behavior of this baseline case, we attempt to see if we can modulate (switch) it to another mode. In figure 9.13 we have increased the rate of crossover from 200 (default value in Appendix B) to $8,000$ and enabled the CLASP boundary conditions to stabilize the edges of the array. The histogram in figure (9.13a) is markedly flatter than the histogram for the CIC case. In figure (9.13b) we have a sample ending state of the simulation used to calculate the histogram. It has a local correlation = 0.5 and a global correlation = 0.49. The MTs in the array for the sample are completely uncorrelated and a network is formed, just like the array with high crossover in Chapter 8, figure (8.11a).



(a) Histogram estimating the probability mass function for array orientation of rectangular domain with a CLASP boundary and a crossover rate of $8,000$, including standard error bars and averaged over 16 runs.

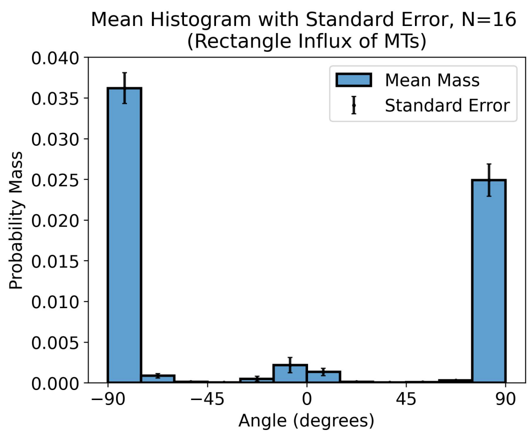


(b) Example of one of the samples used to calculate the array orientation. The array has no preferred orientation as has network-like behavior. Local correlation = 0.5 and global correlation = 0.49

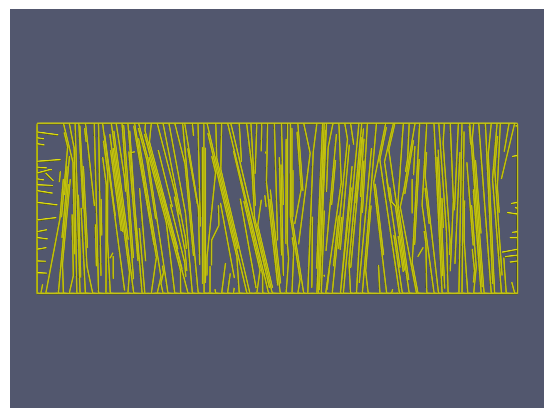
Figure 9.13: Histogram of the mean angular orientation for the arrays at $t = 120$ minutes for the rectangular case CLASP rule enabled and a crossover rate of $8,000$ averaged over 16 runs, along with an ending state sample of the CLASP+Crossover experiment. The ending state has network-like behavior similar to figure (8.11a) in Chapter 8.

In the next experiment, we activate an influx of MTs (by changing the influx rate from 0.001 to 0.008) on the boundaries and lower crossover back to the default rate of 200. MTs can still exit the simulation through the CLASP exit rule as well. In figure 9.14 we have another side-by-side of the mean histogram for the ending state of the realizations of the experiment, along with a sample. In the histogram in figure (9.14a), we observe the angle orientations have switched to be clustered around $-90°$ and $90°$. In this case, this indicates a vertical alignment i.e. aligned with the shortest side of the rectangle (projecting the angles to the right half of the plane causes vertical alignment to appear this way in the histogram). The sample shown in figure (9.14b) verifies the behavior. The sample has a local correlation

= 0.94 and a global correlation = 0.86. In the rectangular domain, the influx of MTs is enough to allow for the majority of MTs on the longer axis to reach the other side first and have a chance to cross over into the anticlinal face. Once they do, they will stay attached until periodically released from the boundary to approximate collision in the hidden plane and any other potential severing effects at the boundary not included in the model. Any newly nucleated MTs will then be forced to align with the dominant axis and entering MTs in the non-dominant axis will be unable to overcome the effects of alignment.



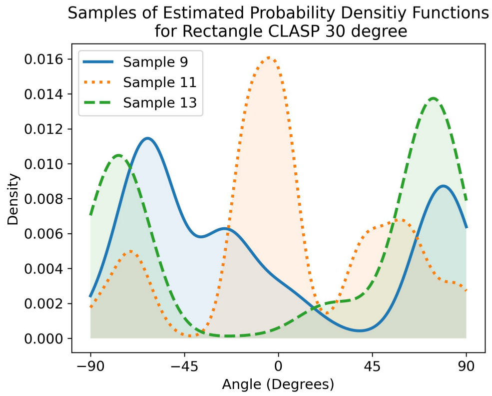
(a) Histogram estimating the probability mass function for array orientation of rectangular domain with a CLASP boundary and an influx of microtubules entering, including standard error bars and averaged over 16 runs.



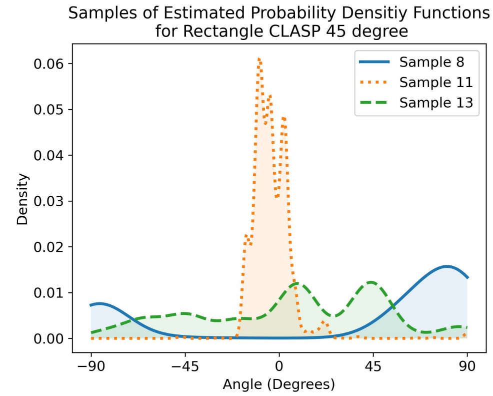
(b) Example of one of the samples used to calculate the array orientation. The array is oriented vertically. Local correlation = 0.94, global correlation = 0.86.

Figure 9.14: Side-by-side comparison of estimations of the array orientation and a sample for the rectangular case with CLASP and an influx of microtubules entering the boundary.

Finally, in the last set of experiments, we lower the rate that MTs can enter back to the default value of 0.001 and vary the entry and exit angles in increments of 15° from 30° to 60°. Instead of analyzing the histograms in this case, we investigate samples and estimations of the density functions of histograms. In figure 9.15 we have two sets of three samples, one set for CLASP 30° (figure 9.15a) and one set for CLASP 45° (figure 9.15b). Comparing the two, we observe different types of orientations. In one case, we get horizontal alignment and in another we have vertical. There are also mixtures.



(a) Three PDFs fit using kernel density estimation with a Gaussian kernel and the Scott bandwidth [103] from the histograms computed for individual samples of the CLASP 30° experiment in a rectangular domain.

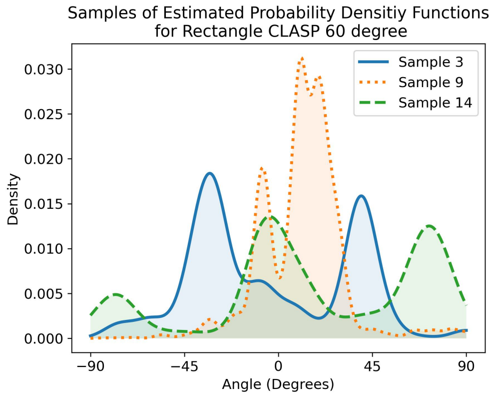


(b) Three PDFs fit using kernel density estimation with a Gaussian kernel and the Scott bandwidth [103] from the histograms computed for individual samples of the CLASP 45° experiment in a rectangular domain.

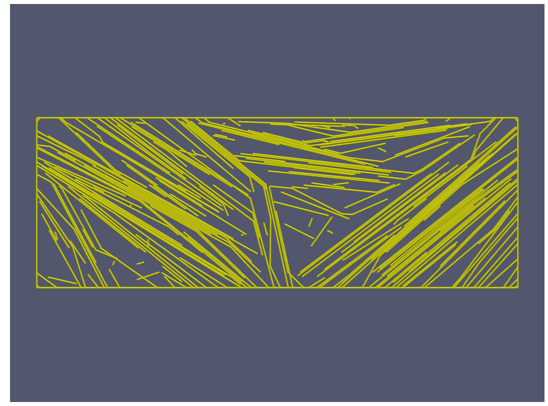
Figure 9.15: Two examples with three samples of estimated orientation PDFs from the CLASP 30° and 45° experiments in rectangular domains.

In figure 9.16, we have the CLASP 60° (figure 9.16a) and sample 3 (figure 9.16b), which is the ending state array. We can use these two plots to effectively observe mixtures of orientations. Sample 3 is a representative example, where there are two distinct diagonal orientations. The local correlation is = 0.81, indicating we have a fairly tight local alignment. The global correlation is = 0.42 and effectively shows how distinct clusters of local alignment can still exist even when the global state appears uncorrelated.

From the final sets of experiments, we have demonstrated that lower rates of MT entry on the boundary can put the orientation in a state where multiple orientations for alignment are possible. However, when one axis fails to win on the global scale, we end up with a mixture of alignment orientations on a local scale. From each of the experiments, we have demonstrated that is possible to shift to different alignment orientations by changing the boundary conditions and modifying the rate at which MTs can cross over one another.



(a) Three PDFs fit using kernel density estimation with a Gaussian kernel and the Scott bandwidth [103] from the histograms computed for individual samples of the CLASP 60° experiment in a rectangular domain.



(b) Sample 3 from 9.16a Local correlation = 0.81 and global correlation = 0.42, which is less than 0.5 due to the inverse orientations at the ends.

Figure 9.16: An example with three samples of an estimated orientation PDF from the CLASP 60° experiment, along with the ending state of sample 3 from the CLASP 60° experiment.

9.5 Conclusion

In this chapter, we have presented a new model based on cortical microtubule array dynamics observed in the periclinal face of a plant cell and run simulations, both of which demonstrate the capability of the Dynamical Graph Grammar Modeling Library (DGGML). In both the rectangular and square domains, we have shown that boundary conditions, as well as MT-MT crossover, can have an impact on the axis of alignment both on a local and a global level. In both domain geometries, we observe that the default set of rules with a collision-inducing catastrophe (CIC) boundary condition alone is insufficient to act as a mechanism for orientation control. We also find that increasing the crossover rate is enough to uncorrelate the array on a local and a global scale. Greater control of alignment is possible, however, in a way that depends on periclinal face geometry.

In the case of square domains, there are actually multiple modes of orientation that can occur: vertical, horizontal, diagonal, and a mixture - regardless of the boundary conditions when using the parameters given in Appendix B. In contrast, boundary conditions play an important role in the alignment results for the rectangular domain. In the CIC case, there is a high likelihood of horizontal (along the longest axis) alignment. In the experiment with an influx of MTs from the boundary, the orientation of the array in the rectangular domain results in a vertical (along the shortest axis) alignment. When the influx is lowered back to the default value and the entry/exit angles are varied, the resulting array is a mixture of orientations. From the rectangular experiments, we find that different boundary conditions in rectangular domains can reorient the axis of alignment for the array.

As a future path forward for the experimental setup outlined in this chapter, we could instead simulate the top periclinal face of the cell using different domain shapes. For example, we could use a hexagonal face for the simulation space. As a further variation, different stretching and skewing transformations could be applied the the cell face. More experiments can also be added with different variations of parameters and rules. For example, we could introduce the severing of MTs at crossover sites with a Katanin rule [35] or add in a stochastic wobble to growth to model thermal fluctuations [86].

The computational nature of the experiment outlined in this chapter also makes collecting more measurements of the system, which are often difficult to collect during lab experiments, more feasible that for real wet-lab biological experiments, and can effectively enable a more sophisticated analysis of the system that would not otherwise be available. For example, we were able to collect MT angular orientations by just defining a metric collection function. An added benefit of DGGML is that it can be used to computationally screen biological experiments that are too costly or cumbersome to perform *en masse*. DGG models defined using and simulated by DGGML can also be used to search for and find corresponding wet-lab experiments that would distinguish between and test alternative hypotheses.

Finally, DGGML offers several other key advantages over the simulation code of Chapter 8. A generic interface for defining rules is exposed, allowing for the creation of more complicated grammars with far less effort as discussed in Chapter 6. The library is also more efficient thanks to improvements such as the incremental update discussed in Chapter 5. Most importantly, DGGML is flexible and customizable.

Chapter 10

Conclusion

10.1 Summary of Motivation

Dynamical Graph Grammars (DGGs) [76] allow for an expressive and powerful way to declare a set of local rules to model a complex dynamic system with graphs. DGGs also have a well-defined meaning. They map graph dynamics into a master equation, a set of first-order linear differential equations governing the time evolution of joint probability distributions of state variables of a dynamic system. Using operator algebra [75], DGGs can be simulated using an exact algorithm (Chapter 4, Algorithm 1). The exact algorithm becomes slow for large systems and signals a need to develop a faster and more scalable algorithm. Using operator splitting, an approximate faster and scalable algorithm was derived for spatially embedded graphs (Chapter 4, Algorithm 4).

The original implementation of the DGG formalism is realized by the software Plenum [129] with powerful symbolic processing provided by Mathematica [128], but only uses the exact algorithm and does not use native graph data structures or graph algorithms. To allow for a more graph and performance-oriented implementation, we have developed the Dynamical

Graph Grammar Modeling Library (DGGML) in the C++ programming language. DGGML also serves as a way to directly demonstrate the utility of the new approximate algorithm.

Two different biological models for the plant cell cortical microtubule array (CMA) were developed. The first model (Chapter 8 and Appendix A) is implemented using the precursor [73] to DGGML, while the second model (Chapter 9, and Appendix B) is completely defined with the DGGML user-interface and simulated using the implementation of Algorithm 4 of Chapter 4 provided by DGGML. Both models are presented to showcase the evolution of DGGML and the flexible power of DGGs.

Work has already been done to simulate the dynamics of MTs in plants [84], [24]. However, to our knowledge, there is no other known formalism that does it by using dynamic graphs. The only previous work is found in Plenum [129] and theoretically discussed in [76] and [130]. Finally, the algorithms, library, and models with corresponding simulations presented in this thesis are intended to fully demonstrate the synergy between the technical computing side of computational science and the mathematical modeling side of systems biology.

10.2 Conclusions and Future Work

The original Dynamical Graph Grammar (DGG) simulation algorithm in [75] can be sped up by processing reactions out-of-order at the cost of accuracy. We have presented the approximate hybrid ODE SSA (Algorithm 4), which is an operator splitting algorithm that imposes a domain decomposition by means of an expanded cell complex that corresponds to summing operators, over pre-expansion dimensions d, and without loss of accuracy over cells c of each dimension. There were two key assumptions made in our approximation of the exact algorithm: spatial locality of the rules and, well-separatedness of the geometric cells (geocells) comprising the expanded cell complex. These two assumptions provide a strong connection between the algorithm and its use for accelerating models rooted in the natural sciences, where dynamics play out locally to sculpt emergent behavior observed at a larger scale in space.

While the approximate hybrid ODE SSA is specifically designed for spatially embedded graphs, it may also have uses in other non-spatial instances if equipped with a measure for locality in the parameter space. Using Algorithm 4 for non-spatial instances could be a topic for future exploration. Further, in Algorithm 2 we have proposed points of parallelization for the exact hybrid ODE SSA (Algorithm 1) in the form of a parallel propensity computation and parallel solving of ODEs. We also introduced an incrementally updated match data structure to greatly improve efficiency and proposed a clear point of parallelization for Algorithm 4, where geocells of the same dimension can be processed in parallel for a short time before the remaining dimensions can be processed and the global inconsistencies in local geocell states can be repaired during the *synchronization* phase. For all algorithms, the best-suited use cases were also discussed. Additionally, choices for the φ function were briefly discussed. In Appendix C, we provide more work on the algorithm and justification that can lead to a formal proof of an error bound in future work. The performance difference between using the exact and approximate algorithm is also demonstrated in Chapter 8.

Following the discussion of the algorithms, the Dynamical Graph Grammar Modeling Library (DGGML) was introduced. Chapter 5 introduced the overall design of the library, along with fundamental building blocks. Notable contributions from it include Yet Another Graph Library (YAGL), a dynamic graph library designed for DGGML, and several foundational elements such as the spatial variant node, subgraph specific pattern recognizer (SSPR), expanded cell complex (ECC), and the process for incrementally updating the set of rule matches after graph rewrites occur.

YAGL, which is responsible for handling the dynamic graphs and providing standard graph algorithms, is full of opportunities for extension and future work. More of the standard graph algorithms could be added, support for all static graph data structures could be added, and most importantly the graph data structure could be adapted to work for parallel graph rewrites both on the CPU and GPU. Moreover, YAGL could be extended to include different modes of parallelization that support different types of concurrent insertion and removal patterns. The SSPR could also benefit from parallelization. More importantly, a C++ implementation of subgraph matching may have the potential to be improved beyond state-of-the-art performance by first doing away with runtime recursion [22] and instead producing unrolled, optimized iterative subgraph pattern matching functions that are precompiled for a specific pattern as was done in the precursor to DGGML [73]; however, this approach is only appropriate for an inflexible, hard-coded DGG model or from one generated from a future DGG compiler.

The ECC, which offers a specialized graph data structure essential to capture both geometric and topological aspects of the simulation space and was a key component of Algorithm 4, also has a clear future path for extension. Currently, the ECC in DGGML is defined for rectangular domains; however, it can also be upgraded to 3D. A 3D ECC for a rectangular prism would also allow Algorithm 4 to be extended to the same dimension. In a higher dimension, we could use the DGG formalism to develop grammars working on the interior or surface of an actual plant cell. ECCs can also be generated for other shapes as well. In fact, DGGs are cabled of being defined on any higher dimensional manifold [76]. The primary issue, however, is the curse of dimensionality. In the case of a higher dimension or the 2D ECC discussed the cell complex can be extended to create a cell complex with different levels of cell complexes that are hierarchically connected at different scales of space.

Chapter 6 functioned as a brief tutorial on how to use DGGML and the user interface created for defining DGG models. Furthermore, it also demonstrated what an embedding of the DGG language and formalism into another programming language looks like. Chapter 7, on the other hand, fully detailed the DGGML framework and its realization of Algorithm 4 by using the building blocks of Chapter 5 and diving deeper into what happens behind the scenes in Chapter 6. We discussed how grammar analysis in DGGML works and how fundamental patterns (*motifs*) in the grammar are found and used in conjunction with the cell list to accelerate the search for matching patterns for the left-hand side of graph grammar rules. We also discussed the performance improvements gained through the use of incremental updates to the match data structure of possible matches to connected components of left-hand side (LHS) labeled graphs and rule matches that are comprised of component matches. Effectively, DGGML has been shown to offer customizable simulation of DGGs, leveraging features like dynamic ODE solving, while also providing observable metric collection during the simulation loop. The model-building interface of DGGML abstracts away much of the complexity of the underlying simulation algorithm, opening up the path for future higher-level usage and modeling applications such as the synaptic morphodynmaics of neurons [62].

We also demonstrated how DGGs, DGGML, and our new approximate hybrid ODE SSA (Algorithm 4) can be used to simulate and accelerate simulations of complex biological systems through our development of two models for the CMA. In the first model, (Chapter 8), we introduced an initial implementation of the approximate algorithm, for spatially embedded and local DGG dynamics, and achieved improvements to performance over an exact algorithm with the caveat that there may be some potential cost in accuracy. We ran two experiments for the CMA DGG (Appendix A). In the first we observed a long-time behavior of cytoskeletal network-like formation, and in the second we observed a long-time behavior of localized alignment for the simulated array.

In the follow-up chapter (Chapter 9), we presented a new model based on cortical microtubule array dynamics observed in the periclinal face of a plant cell, and ran simulations, both of which demonstrated the capability of the Dynamical Graph Grammar Modeling Library (DGGML). In the model, we restricted ourselves to the CMA in the periclinal face of a plant cell and explored the effects that different face shapes and boundary conditions had on local and global alignment. In the case of a square face shape, we found the array orientation to be multi-modal, and in the case of a rectangular face shape, we found that different boundary conditions reorient the array. As a future modeling path, we could instead simulate the top periclinal face of the cell using different domain shapes. For example, we could use a hexagonal face for the simulation space. As a further variation, different stretching and skewing transformations could be applied the the cell face. More experiments can also be added with different variations of parameters and rules. For example, we could introduce the severing of MTs at crossover sites with a Katanin rule [35] or add in a stochastic wobble to growth to model thermal fluctuations [86].

The computational nature of the experiment outlined in this thesis also makes collecting more measurements of the system, which are often difficult to collect during lab experiments, more feasible and can effectively enable more sophisticated analysis of the system that would not otherwise be available. For example, we were able to collect MT angular orientations by just defining a metric collection function. An added benefit of DGGML is that it can be used to computationally screen biological experiments that are too costly or cumbersome. DGG models defined using and simulated by DGGML can also be used to search for corresponding wet-lab experiments that would distinguish between and test alternative hypotheses for emergent phenomena. There is also potential to reduce the model or learn governing equations for emergent behavior by using machine learning, as in [44] or [101].

As made evident by now, the applications for Dynamical Graph Grammars are vast, and the field of DGG research is expansive. As the landscape for computing continuously changes and evolves, new implementations scaling with and accelerated by new hardware will be needed, and new, more complex models will finally be unlocked or discovered. The clearest future path for the Dynamical Graph Grammar Modeling Library is the evolution to a Dynamical Graph Grammar compiler for the Dynamical Graph Grammar Modeling Language. If Dynamical Graph Grammars were combined with a modeling language and machine learning, even more powerful and intelligent graph rewriting systems could emerge.

Bibliography

[1] E. B. Abrash and D. C. Bergmann. Asymmetric cell divisions: A view from plant development. *Developmental Cell*, 16(6):783–796, 2009.

[2] K. Ahnert and M. Mulansky. Odeint – solving ordinary differential equations in c++. *AIP Conference Proceedings*, 1389(1):1586–1589, 2011.

[3] J. F. Allard, G. O. Wasteneys, and E. N. Cytrynbaum. Mechanisms of self-organization of cortical microtubules in plants revealed by computational simulations. *Molecular Biology of the Cell*, 21(2):278–286, 2010. PMID: 19910489.

[4] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, 06 2017.

[5] C. Ambrose, J. F. Allard, E. N. Cytrynbaum, and G. O. Wasteneys. A clasp-modulated cell edge barrier mechanism drives cell-wide cortical microtubule organization in arabidopsis. *Nature Communications*, 2(1):430, Aug 2011.

[6] J. C. Ambrose and G. O. Wasteneys. Clasp modulates microtubule-cortex interaction during self-organization of acentrosomal microtubules. *Molecular biology of the cell*, 19(11):4730–4737, 2008.

[7] T. Atkinson, D. Plump, and S. Stepney. Evolving graphs by graph programming. In M. Castelli, L. Sekanina, M. Zhang, S. Cagnoni, and P. García-Sánchez, editors, *Genetic Programming*, pages 35–51, Cham, 2018. Springer International Publishing.

[8] A. Auger, P. Chatelain, and P. Koumoutsakos. R-leaping: Accelerating the stochastic simulation algorithm by reaction leaps. *The Journal of Chemical Physics*, 125(8):084103, 08 2006.

[9] U. Ayachit. *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc., 2015.

[10] C. J. Balos, D. J. Gardner, C. S. Woodward, and D. R. Reynolds. Enabling GPU accelerated computing in the SUNDIALS time integration library. *Parallel Computing*, 108:102836, 2021.

[11] V. A. Baulin, C. M. Marques, and F. Thalmann. Collision induced spatial organization of microtubules. *Biophysical Chemistry*, 128(2):231–244, 2007.

[12] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. W. Scogland. Raja: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE/ACM, 2019.

[13] M. Bender, M. Farach-Colton, and M. Mosteiro. Insertion sort is o(n log n). *Theory of Computing Systems*, 39, 06 2006.

[14] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975.

[15] S. Besson and J. Dumais. Universal rule for the symmetric division of plant cells. *Proceedings of the National Academy of Sciences*, 108(15):6294–6299, 08 2011.

[16] A. Bortz, M. Kalos, and J. Lebowitz. A new algorithm for monte carlo simulation of ising spin systems. *Journal of Computational Physics*, 17(1):10–18, 1975.

[17] A. Bretto. *Digital Topologies on Graphs*, pages 65–82. Springer, Berlin, 2007.

[18] E. Brisson. Representing geometric structures ind dimensions: Topology and order. *Discrete & Computational Geometry*, 9(4):387–426, 04 1993.

[19] K. S. Burbank and T. J. Mitchison. Microtubule dynamic instability. *Current Biology*, 16(14):R516–R517, 07 2006.

[20] F. Busato, O. Green, N. Bombieri, and D. A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, 2018.

[21] Y. Cao, D. T. Gillespie, and L. R. Petzold. Efficient step size selection for the tau-leaping simulation method. *The Journal of Chemical Physics*, 124(4):044109, 01 2006.

[22] V. Carletti, P. Foggia, A. Saggese, and M. Vento. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):804–818, 2018.

[23] J. Castillo and G. Miranda. *Mimetic discretization methods.* CRC PRESS, 2019.

[24] B. Chakrabortty, V. Willemsen, T. de Zeeuw, C.-Y. Liao, D. Weijers, B. Mulder, and B. Scheres. A plausible microtubule-based mechanism for cell division orientation in plant embryogenesis. *Current Biology*, 28(19):3031–3043.e2, 10 2018.

[25] J. Chan, C. G. Jensen, L. C. Jensen, M. Bush, and C. W. Lloyd. The 65-kda carrot microtubule-associated protein forms regularly arranged filamentous cross-bridges between microtubules. *Proceedings of the National Academy of Sciences*, 96(26):14931–14936, 1999.

[26] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[27] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[28] E. Coen and D. J. Cosgrove. The mechanics of plant morphogenesis. *Science*, 379(6631):eade8055, 2023.

[29] L. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the vf graph matching algorithm. In *Proceedings 10th International Conference on Image Analysis and Processing*, pages 1172–1177, 1999.

[30] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[31] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[32] R. Dechter. Bucket elimination: a unifying framework for processing hard and soft constraints. *Constraints*, 2(1):51–55, Apr 1997.

[33] R. Dechter. *Constraint Processing*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, San Francisco, 2003.

[34] P. Degano and U. Montanari. A model for distributed systems based on graph rewriting. *J. ACM*, 34(2):411–449, apr 1987.

[35] E. E. Deinum, S. H. Tindemans, J. J. Lindeboom, and B. M. Mulder. How selective severing by katanin promotes order in the plant cortical microtubule array. *Proceedings of the National Academy of Sciences*, 114(27):6942–6947, 2017.

[36] R. Diestel. *Graph Theory*. Springer, 5th edition, 2017.

[37] R. Dixit and R. Cyr. Encounters between Dynamic Cortical Microtubules Promote Ordering of the Cortical Array through Angle-Dependent Modifications of Microtubule Behavior. *The Plant Cell*, 16(12):3274–3284, 12 2004.

[38] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 167–180, 1973.

[39] A. Elliott and S. L. Shaw. Update: Plant Cortical Microtubule Arrays. *Plant Physiology*, 176(1):94–105, 11 2017.

[40] D. Engwirda. Conforming restricted delaunay mesh generation for piecewise smooth complexes. *Procedia Engineering*, 163:84–96, 2016. 25th International Meshing Roundtable.

[41] D. Eppstein. Subgraph isomorphism in planar graphs and related problems, 1999.

[42] E. C. Eren, R. Dixit, and N. Gautam. A three-dimensional computer simulation model reveals the mechanisms for self-organization of plant cortical microtubules into oblique arrays. *Molecular Biology of the Cell*, 21(15):2674–2684, 2010. PMID: 20519434.

[43] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann series in interactive 3D technology. CRC Press, 2004.

[44] O. K. Ernst, T. Bartol, T. Sejnowski, and E. Mjolsness. Learning dynamic boltzmann distributions as reduced models of spatial chemical kinetics. *The Journal of Chemical Physics*, 149(3):034107, 2018.

[45] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[46] D. J. Gardner, D. R. Reynolds, C. S. Woodward, and C. J. Balos. Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 48(3):1–24, 2022.

[47] J.-L. Giavitto and O. Michel. Mgs: A rule-based programming language for complex objects and collections. *Electronic Notes in Theoretical Computer Science*, 59(4):286–304, 2001. RULE 2001, Second International Workshop on Rule-Based Programming (Satellite Event of PLI 2001).

[48] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[49] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[50] D. T. Gillespie. A rigorous derivation of the chemical master equation. *Physica A: Statistical Mechanics and its Applications*, 188(1):404–425, 09 1992.

[51] D. T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *The Journal of Chemical Physics*, 115(4):1716–1733, 06 2001.

[52] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, 2nd edition, 1993.

[53] O. Hamant, D. Inoue, D. Bouchez, J. Dumais, and E. Mjolsness. Are microtubules tension sensors? *Nature Communications*, 10(1):2360, 05 2019.

[54] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 09 2020.

[55] A. Hatcher. *Algebraic topology*. Cambridge University Press, 2019.

[56] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, 2006. Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

[57] R. Himmelspach, R. E. Williamson, and G. O. Wasteneys. Cellulose microfibril alignment recovers from dcb-induced disruption despite microtubule disorganization. *The Plant Journal*, 36(4):565–575, 2003.

[58] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 09 2005.

[59] M. Hirsch. *Differential Topology*. Graduate Texts in Mathematics. Springer New York, 2012.

[60] M. W. Hirsch. *Differential Topology*, chapter 4, page 109. Springer, 1976.

[61] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 05 2007.

[62] M. Hur, M. Bonilla-Quinatana, T. M. Bartol, T. J. Sejnowski, P. Rangamani, and E. Mjolsness. Simulating actin networks in synaptic spine heads using dynamical graph grammars. *Biophysical Journal*, 122(3):283a, Feb 2023.

[63] A. Jüttner and P. Madarasi. Vf2++—an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81, 2018. Computational Advances in Combinatorial Optimization.

[64] B. Lane. *Cell Complexes: The Structure of Space and the Mathematics of Modularity*. PhD thesis, University of Calgary Computer Science Department, 09 2015.

[65] G. Langdale and D. Lemire. Parsing gigabytes of json per second. *The VLDB Journal*, 28(6):941–960, 10 2019.

[66] D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S. R. Slattery. Arborx: A performance portable geometric search library. *ACM Trans. Math. Softw.*, 47(1), dec 2020.

[67] P. Lecca, I. Laurenzi, and F. Jordan. 1 - deterministic chemical kinetics. In P. Lecca, I. Laurenzi, and F. Jordan, editors, *Deterministic Versus Stochastic Modelling in Biochemistry and Systems Biology*, Woodhead Publishing Series in Biomedicine, pages 1–34. Woodhead Publishing, 2013.

[68] P. Lecca, I. Laurenzi, and F. Jordan. 4 - modelling in systems biology. In P. Lecca, I. Laurenzi, and F. Jordan, editors, *Deterministic Versus Stochastic Modelling in Biochemistry and Systems Biology*, Woodhead Publishing Series in Biomedicine, pages 117–180. Woodhead Publishing, 2013.

[69] E. Lifshitz, A. Kosevich, and L. Pitaevskii. *Theory of Elasticity*. Butterworth-Heinemann, third edition edition, 1986.

[70] J. Lipková, G. Arampatzis, P. Chatelain, B. Menze, and P. Koumoutsakos. S-leaping: An adaptive, accelerated stochastic simulation algorithm, bridging τ-leaping and r-leaping. *Bulletin of Mathematical Biology*, 81(8):3074–3096, 08 2019.

[71] J. I. Lopez-Veyna, I. Castillo-Zuñiga, and M. Ortiz-Garcia. A review of graph databases. In J. Mejia, M. Muñoz, Á. Rocha, and V. Hernández-Nava, editors, *New Perspectives in Software Engineering*, pages 180–195, Cham, 2023. Springer International Publishing.

[72] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1):181–224, 1993.

[73] E. Medwedeff and E. Mjolsness. Approximate simulation of cortical microtubule models using dynamical graph grammars. *Physical Biology*, 20(5):055002, jul 2023.

[74] E. Mjolsness. Towards measurable types for dynamical process modeling languages. *Electronic notes in theoretical computer science*, 265:123–144, 09 2010.

[75] E. Mjolsness. Time-ordered product expansions for computational stochastic system biology. *Physical Biology*, 10(3):035009, 06 2013.

[76] E. Mjolsness. Prospects for declarative mathematical modeling of complex biological systems. *Bulletin of Mathematical Biology*, 81(8):3385–3420, 08 2019.

[77] E. Mjolsness. Explicit calculation of structural commutation relations for stochastic and dynamical graph grammar rule operators in biological morphodynamics. *Frontiers in Systems Biology*, 2, 09 2022.

[78] E. Mjolsness and E. Medwedeff. Parallel dgg simulation draft. 2019.

[79] E. Mjolsness, D. Orendorff, P. Chatelain, and P. Koumoutsakos. An exact accelerated stochastic simulation algorithm. *The Journal of Chemical Physics*, 130(14):144110, 04 2009.

[80] E. Mjolsness and G. Yosiphon. Stochastic process semantics for dynamical grammars. *Annals of Mathematics and Artificial Intelligence*, 47(3):329–395, 01 2007.

[81] J. R. Munkres. *Topology*. Prentice Hall, 2nd. edition, 2000.

[82] T. Murata, S. Sonobe, T. I. Baskin, S. Hyodo, S. Hasezawa, T. Nagata, T. Horio, and M. Hasebe. Microtubule-dependent microtubule nucleation based on recruitment of γ-tubulin in higher plants. *Nature Cell Biology*, 7(10):961–968, Oct 2005.

[83] M. Nakamura and T. Hashimoto. A mutation in the Arabidopsis γ-tubulin-containing complex causes helical growth and abnormal microtubule branching. *Journal of Cell Science*, 122(13):2208–2217, 07 2009.

[84] F. Nedelec and D. Foethke. Collective langevin dynamics of flexible cytoskeletal fibers. *New Journal of Physics*, 9(11):427–427, 11 2007.

[85] D. Orendorff and E. Mjolsness. A hierarchical exact accelerated stochastic simulation algorithm. *The Journal of Chemical Physics*, 137(21):214104, 12 2012.

[86] F. Pampaloni, G. Lattanzi, A. Jonáš, T. Surrey, E. Frey, and E.-L. Florin. Thermal fluctuations of grafted microtubules provide evidence of a length-dependent persistence length. *Proceedings of the National Academy of Sciences*, 103(27):10248–10253, 2006.

[87] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, Heidelberg, 1996.

[88] C. Rackauckas and Q. Nie. DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software*, 5(1), 2017.

[89] A. Rand and N. Walkington. Collars and intestines: Practical conforming Delaunay refinement. In B. W. Clark, editor, *Proceedings of the 18th International Meshing Roundtable*, pages 481–497, Berlin, 2009. Springer.

[90] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proc. VLDB Endow.*, 8(5):617–628, jan 2015.

[91] D. R. Reynolds, D. J. Gardner, R. C. Carol S. Woodward, and C. J. Balos. User documentation for arkode. urlhttps://sundials.readthedocs.io/en/latest/arkode, 2024. v6.0.0.

[92] D. R. Reynolds, D. J. Gardner, C. S. Woodward, and R. Chinomona. ARKODE: A flexible IVP solver infrastructure for one-step methods. *ACM Transactions on Mathematical Software*, 49(2):1–26, 2023.

[93] R. Robey and Y. Zamora. *Parallel and High Performance Computing*. Manning, 2021.

[94] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, page 1–10, New York, NY, USA, 2015. Association for Computing Machinery.

[95] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*. WORLD SCIENTIFIC, 1997.

[96] S. J. Russell, P. Norvig, and M.-W. Chang. *Artificial Intelligence: A modern approach*. Pearson, 2021.

[97] A. Sambade, A. Pratap, H. Buschmann, R. J. Morris, and C. Lloyd. The Influence of Light on Microtubule Dynamics and Alignment in the Arabidopsis Hypocotyl. *The Plant Cell*, 24(1):192–201, 01 2012.

[98] A. Sampathkumar, P. Krupinski, R. Wightman, P. Milani, A. Berquand, A. Boudaoud, O. Hamant, H. Jönsson, and E. M. Meyerowitz. Subcellular and supracellular mechanical stress prescribes cytoskeleton behavior in *Arabidopsis* cotyledon pavement cells. *eLife*, 3:e01967, 04 2014.

[99] R. Schneider, K. v. Klooster, K. L. Picard, J. van der Gucht, T. Demura, M. Janson, A. Sampathkumar, E. E. Deinum, T. Ketelaar, and S. Persson. Long-term single-cell imaging and simulations of microtubules reveal principles behind wall patterning during proto-xylem development. *Nature Communications*, 12(1):669, Jan 2021.

[100] W. Schroeder, K. M. Martin, and W. E. Lorensen. *The Visualization Toolkit (2nd Ed.): An Object-Oriented Approach to 3D Graphics*. Prentice-Hall, 1998.

[101] C. B. Scott and E. Mjolsness. Multilevel artificial neural network training for spatially correlated learning. *SIAM Journal on Scientific Computing*, 41(5):S297–S320, 2019.

[102] C. B. Scott, E. Mjolsness, D. Oyen, C. Kodera, M. Uyttewaal, and D. Bouchez. Graph metric learning quantifies morphological differences between two genotypes of shoot apical meristem cells in Arabidopsis. *in silico Plants*, 5(1), 01 2023. diad001.

[103] D. W. Scott. *Kernel Density Estimation*, pages 1–7. John Wiley and Sons, Ltd, 2018.

[104] B. E. Shapiro, A. Levchenko, E. M. Meyerowitz, B. J. Wold, and E. Mjolsness. Cellerator: extending a computer algebra system to include biochemical arrows for signal transduction simulations. *Bioinform.*, 19(5):677–678, 2003.

[105] B. E. Shapiro and E. Mjolsness. Pycellerator: an arrow-based reaction-like modelling language for biological simulations. *Bioinformatics*, 32(4):629–631, 10 2015.

[106] S. L. Shaw. Reorganization of the plant cortical microtubule array. *Current Opinion in Plant Biology*, 16(6):693–697, 2013. Cell biology.

[107] S. L. Shaw, R. Kamyar, and D. W. Ehrhardt. Sustained microtubule treadmilling in arabidopsis cortical arrays. *Science*, 300(5626):1715–1718, 06 2003.

[108] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Pearson Education, 01 2002.

[109] S. Slattery, S. T. Reeve, C. Junghans, D. Lebrun-Grandié, R. Bird, G. Chen, S. Fogerty, Y. Qiu, S. Schulz, A. Scheinberg, A. Isner, K. Chong, S. Moore, T. Germann, J. Belak, and S. Mniszewski. Cabana: A performance portable library for particle-based simulations. *Journal of Open Source Software*, 7(72):4115, 2022.

[110] A. Spicher and O. Michel. Declarative modeling of a neurulation-like process. *Biosystems*, 87(2):281–288, 2007. Papers presented at the Sixth International Workshop on Information Processing in Cells and Tissues, York, UK, 2005.

[111] J. R. Stiles, T. M. Bartol, et al. Monte carlo methods for simulating realistic synaptic microphysiology using mcell. *Computational neuroscience: realistic modeling for experimentalists*, pages 87–127, 2001.

[112] J. Strikwerda. *Finite Difference Schemes and Partial Differential Equations, Second Edition*, chapter 1, pages 1–36. Wadsworth and Brooks/Cole, 2nd edition, 1989.

[113] D. Thoms, L. Vineyard, A. Elliott, and S. L. Shaw. CLASP Facilitates Transitions between Cortical Microtubule Array Patterns. *Plant Physiology*, 178(4):1551–1567, 10 2018.

[114] S. Tindemans, E. Deinum, J. Lindeboom, and B. Mulder. Efficient event-driven simulations shed new light on microtubule organization in the plant cortical array. *Frontiers in Physics*, 2, 2014.

[115] S. Tindemans, R. Hawkins, and B. Mulder. Survival of the aligned: Ordering of the plant cortical microtubule array. *Physical review letters*, 104:058103, 02 2010.

[116] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817, 2022.

[117] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, jan 1976.

[118] J. R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *ACM J. Exp. Algorithmics*, 15, feb 2011.

[119] G. Valiente. *Graph Isomorphism*, pages 255–285. Springer International Publishing, Cham, 2021.

[120] L. Vineyard, A. Elliott, S. Dhingra, J. R. Lucas, and S. L. Shaw. Progressive Transverse Microtubule Array Organization in Hormone-Induced Arabidopsis Hypocotyl Cells. *The Plant Cell*, 25(2):662–676, 02 2013.

[121] J. W. Vos, M. Dogterom, and A. M. C. Emons. Microtubules become more dynamic but not shorter during preprophase band formation: A possible "search-and-capture" mechanism for microtubule translocation. *Cell Motility*, 57(4):246–258, 01 2004.

[122] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock. *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb 2016.

[123] G. O. Wasteneys. Microtubule organization in the green kingdom: chaos or self-order? *Journal of Cell Science*, 115(7):1345–1354, 04 2002.

[124] B. Wheatman and H. Xu. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, 2018.

[125] R. Wightman, G. Chomicki, M. Kumar, P. Carr, and S. Turner. Spiral2 determines plant microtubule organization by modulating microtubule severing. *Current Biology*, 23(19):1902–1907, 2013.

[126] R. Wightman and S. R. Turner. Severing at sites of microtubule crossover contributes to microtubule alignment in cortical arrays. *The Plant Journal*, 52(4):742–751, 2007.

[127] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[128] Wolfram Research, Inc. Mathematica, 2021.

[129] G. Yosiphon. *Stochastic Parameterized Grammars : Formalization, Inference and Modeling Applications*. PhD thesis, UC Irvine, 06 2009.

[130] G. Yosiphon and E. Mjolsness. *Towards the Inference of Stochastic Biochemical Network and Parameterized Grammar Models*. MIT Press, 12 2009.

[131] W. M. Young and E. W. Elcock. Monte carlo studies of vacancy migration in binary ordered alloys: I. *Proceedings of the Physical Society*, 89(3):735–746, 04 1966.

Appendix A

Minimal CMA DGG

A.1 Overview and Definitions

There are two different classes of rules, *continuous* and *discrete*. A continuous rule is a rule that updates parameters by solving an ordinary differential equation (ODE). A discrete rule is a rule that rewrites the system graph and may modify parameters by sampling new ones from a factored propensity function. In the main text, we discussed the factorized form of rules and the general form for rules. The discrete rules for this grammar are written in a more compact shorthand form. Newly sampled output values and variables used in rate calculations are sub-parts of a *where* clause. Parameters are exactly sampled to be those values by using a Dirac delta distribution. As stated in the results section, artificial parameters are chosen to evaluate the simulation algorithm and code, rather than ones chosen to represent biophysical knowledge.

Table A.1 lists the symbols used to represent nodes types in the DGG CMA rules. Table A.2 lists the commonly used rule parameters in the grammar, and includes a brief description of their meaning. In table A.3 we do the same, but for model parameters. Finally, table A.4

Graph Symbol	Type Name
●	growing
○	intermediate
■	retraction
▲	zipper
⧫	junction

Table A.1: Graph Node Type Symbol Table

Rule Parameter	Description
x_n	a point in $\mathbb{R}^2$
u_n	a unit vector in $\mathbb{R}^2$
L	current length of the MT segment
β	minimum distance from collision point
I	an intersection point

Table A.2: Rule Parameter Definitions

lists commonly used functions and a description as well.

Model Parameter	Descpription	Value
L_{div}	the maximal dividing length of a segment	1.2 units
L_{min}	the minimal length of a segment	0.125 units
$[Y_g]$	tubulin concentration	1.0 units
v_{plus}	growth rate	1.0 units per time
v_{minus}	retraction rate	0.25 units per time
γ	parametric line parameter	1.0
ϵ	maximal reaction radius	$2.0 * L_{div}$
θ_{crit}	critical angle	$\frac{2\pi}{9}$ radians
c	a constant factor	1.0
k	sigmoid factor	10
$\hat{\rho}_{\text{retract}\leftarrow\text{growth}}$	retraction to growth conversion rate	0.01
$\hat{\rho}_{\text{growth}\leftarrow\text{retract}}$	growth to retraction conversion rate	0.01

Table A.3: Model Parameter Definitions

The cortical microtubule array (CMA) grammar presented is a minimal set of rules capable of leading to the long-time behavior of network formation. The grammar consists of six rules: two for growth, two for retraction, a collision rule with three different outcomes, and a reversible state change of growing to retracting. We introduce and briefly discuss these in the following subsections.

Function	Description
$\hat{\rho}_{\text{grow}}(x) = (x)(v_{plus})$	growth rate function
$\hat{\rho}_{\text{retract}}(x) = (x)(v_{minus})$	retraction rate function
$\hat{\rho}_{\text{zipper}}(x) = x$	zipper rate function
$\hat{\rho}_{\text{junction}}(x) = x$	junction rate function
$\sigma(x;k) \equiv \frac{1}{1+e^{-kx}}$	sigmoid function
$H(x;a) = \begin{cases} 1 & x \geq a \\ 0 & x < a \end{cases}$	unit step function
$\Theta(\text{stmt}) = \begin{cases} 1 & \text{if stmt is true} \\ 0 & \text{if stmt is false} \end{cases}$	indication function

Table A.4: Function Descriptions

A.2 Positive MT Growth

The positive growth rule models polymerization and effectively elongates the MT. The rule in equation A.1 describes MT growth (polymerization), and is a continuous rule. The position of x_2 is updated by solving an ordinary differential equation limited by the dividing length in the direction of the growing node. The function $\hat{\rho}_{\text{grow}}$ (table A.4) can be any function of tubulin concentration, but is set to be a linear function with a constant rate of growth v_{plus} and $[Y_g]$ both of which are set in table A.3.

$$\begin{aligned}
&(\bigcirc_1 \,—\, \bullet_2)\langle\!\langle(\boldsymbol{x}_1, \boldsymbol{u}_1)(\boldsymbol{x}_2, \boldsymbol{u}_2)\rangle\!\rangle \\
&\longrightarrow (\bigcirc_1 \,—\, \bullet_2)\langle\!\langle(\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2 + d\boldsymbol{x}_2, \boldsymbol{u}_2)\rangle\!\rangle \\
&\qquad \textbf{solving} \quad d\boldsymbol{x}_2/dt = \hat{\rho}_{\text{grow}}([Y_{\text{g}}])(1 - L/L_{\text{div}})\boldsymbol{u}_2
\end{aligned} \tag{A.1}$$

A.3 Positive MT Overgrowth

Equation A.2 is a discrete rule. When a MT segment becomes long enough, a new node is added in between the previous two nodes and splits the edge. This process is called “overgrowth” with respect to the CMA DGG. A sigmoid function provides a fully activated rate when a dividing threshold has been reached. The new node, $\bigcirc_3$, is placed at a point controlled by the γ parameter on the line between x_1 and x_2. An additional constant c can be used to further scale the rate, but for these simulations it is set to unity.

$$
\begin{aligned}
&(\bigcirc_1 — \bullet_2)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2)\rangle\!\rangle\\
&\longrightarrow (\bigcirc_1 — \bigcirc_3 — \bullet_2)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3)\rangle\!\rangle\\
&\qquad \textbf{with}\ \ c\,\sigma\Big(\frac{\|\boldsymbol{x}_2-\boldsymbol{x}_1\|}{L_{\text{div}}};k=10\Big)\\
&\qquad \textbf{where}\ \begin{cases}\boldsymbol{x}_3=\boldsymbol{x}_2-(\boldsymbol{x}_2-\boldsymbol{x}_1)\gamma\\ \boldsymbol{u}_3=\frac{\boldsymbol{x}_3-\boldsymbol{x}_2}{\|\boldsymbol{x}_3-\boldsymbol{x}_2\|}\end{cases}
\end{aligned}
\tag{A.2}
$$

A.4 Negative MT Retraction

Negative MT retraction is a continuous rule used to model depolymerization. Equation A.3 is similar to the growth rule in A.1, but instead functions to limit how much the MT segment can retract. The function $\hat{\rho}_{\text{retract}}$ (table A.4) can be any function of tubulin concentration, but is set to be a linear function with a constant rate of growth $v_{retract}$ and $[Y_g]$ both of which are set in table A.3. The limit, L_{min}, is primarily added to prevent instability from round off errors in the ODE solver caused by two points being too close to one another.

$$
\begin{aligned}
&(\blacksquare_1 — \bigcirc_2)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2)\rangle\!\rangle\\
&\longrightarrow (\blacksquare_1 — \bigcirc_2)\,\langle\!\langle(\boldsymbol{x}_1+d\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2)\rangle\!\rangle\\
&\qquad \textbf{solving}\ \ d\boldsymbol{x}_1/dt=\hat{\rho}_{\text{retract}}([Y_{\text{g}}])(L/L_{\text{min}})\boldsymbol{u}_1
\end{aligned}
\tag{A.3}
$$

A.5 Negative MT Undergrowth

The rule in equation A.4 is a discrete rule used to shorten a MT. In the context of the CMA DGG, the process of shortening is "undergrowth". The rule in equation A.2 inserts more intermediate nodes into the system and the negative MT undergrowth rule removes them. Here, a Heaviside function H scaled by a constant c set to unity is used. When the retraction node gets close enough to its intermediate neighbor, the rate function activates. If this rule is selected to occur, the neighboring intermediate node is removed.

$$\begin{aligned}&(\blacksquare_1 - \bigcirc_2 - \bigcirc_3)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3)\rangle\!\rangle\\ &\longrightarrow (\blacksquare_1 - \bigcirc_3)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),\varnothing,(\boldsymbol{x}_3,\boldsymbol{u}_3)\rangle\!\rangle\\ &\qquad\textbf{with}\ \ c\,H(-\|\boldsymbol{x}_2-\boldsymbol{x}_1\|;-0.2*L_{\text{div}})\end{aligned} \tag{A.4}$$

A.6 MT Collision

The collision rule is more combinatorially complex than the rest of the rules, having three alternative discrete outcomes. The LHS of the grammar rule can be seen in equation A.5. It represents the case where a growing end of a MT is close enough to the interior of another MT.

$$\left(\begin{matrix}\bigcirc_1 - \bigcirc_2 - \bigcirc_3\\ \bigcirc_4 - \bullet_5\end{matrix}\right)\ \begin{matrix}\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_3),(\boldsymbol{x}_3,\boldsymbol{u}_3),\\ (\boldsymbol{x}_4,\boldsymbol{u}_4),(\boldsymbol{x}_5,\boldsymbol{u}_5)\rangle\!\rangle\end{matrix} \tag{A.5}$$

Each of the three discrete outcomes makes use of locally defined rule specific parameters from the where clause in equation A.6. Normally, the where clause would follow the with clause like in equation A.8, but it is separated out for clarity. In equation A.6, I_1, I_2 are intersection points of lines parameterized by $\alpha_1, \alpha_2 \in (-\infty, \infty)$ starting at the point x_2 in the directions u_1, u_2. The parameter α_1 indicates where the growing edge $(\bigcirc_1, \bullet_5)$ intersects edge $(\bigcirc_1, \bigcirc_2)$ and α_2 indicates where the growing edge $(\bigcirc_1, \bullet_5)$ intersects edge $(\bigcirc_2, \bigcirc_3)$.

$$\textbf{where} \quad \begin{cases} \alpha_1 = (\boldsymbol{x}_5 \cdot (\boldsymbol{x}_1 - \boldsymbol{x}_2) - \boldsymbol{x}_2 \cdot (\boldsymbol{x}_1 - \boldsymbol{x}_2))/||\boldsymbol{x}_1 - \boldsymbol{x}_2||^2 \\ \alpha_2 = (\boldsymbol{x}_5 \cdot (\boldsymbol{x}_3 - \boldsymbol{x}_2) - \boldsymbol{x}_2 \cdot (\boldsymbol{x}_3 - \boldsymbol{x}_2))/||\boldsymbol{x}_3 - \boldsymbol{x}_2||^2 \\ I_1 = \boldsymbol{x}_2 + \alpha_1(\boldsymbol{x}_1 - \boldsymbol{x}_2) \\ I_2 = \boldsymbol{x}_2 + \alpha_2(\boldsymbol{x}_3 - \boldsymbol{x}_2) \\ \beta_1(\alpha_1) = \begin{cases} ||\boldsymbol{x}_5 - I_1||^2, & 0 < \alpha_1 < 1 \\ ||\boldsymbol{x}_5 - \boldsymbol{x}_1||^2, & \alpha_1 \geq 1 \\ ||\boldsymbol{x}_5 - \boldsymbol{x}_2||^2, & 0 \leq \alpha_1 \end{cases} \\ \beta_2(\alpha_2) = \begin{cases} ||\boldsymbol{x}_5 - I_2||^2, & 0 < \alpha_2 < 1 \\ ||\boldsymbol{x}_5 - \boldsymbol{x}_3||^2, & \alpha_2 \geq 1 \\ ||\boldsymbol{x}_5 - \boldsymbol{x}_2||^2, & 0 \leq \alpha_2 \end{cases} \\ \beta = min(\beta_1, \beta_2) \end{cases} \tag{A.6}$$

The functions β_1, β_2 are used to calculate the minimum distance to intersection along the respective segments. Since the growing edge $(\bigcirc_1, \bullet_5)$ could intersect $(\bigcirc_1, \bigcirc_2)$ or $(\bigcirc_2, \bigcirc_3)$, we must find the closest of the two. Hence, we use $\beta = min(\beta_1, \beta_2)$.

The first discrete outcome is zippering (equation A.7), with additional parameters defined in the where clause in equation A.6. The rate function primarily depends on the incoming MT being at a critical angle and the incoming MT being on a collision trajectory with another MT in the reaction radius. If these conditions are met, the rate function becomes non-zero and increases as the collision gap narrows. The rewrite on the right hand side (RHS) attaches the growing end to the segment and transforms it into a zippering node.

$$\longrightarrow \left(\begin{array}{ccccc} \bigcirc_1 & \text{—} & \blacktriangle_2 & \text{—} & \bigcirc_3 \\ & \diagup & & & \\ \bigcirc_4 & & & & \end{array}\right) \begin{array}{l} \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), \\ (\boldsymbol{x}_4, \boldsymbol{u}_4), \varnothing \rangle\!\rangle \end{array}$$

$$\begin{array}{ll} & \hat{\rho}_{\text{zipper}}(|\boldsymbol{u}_2 \cdot \boldsymbol{u}_4|/|\cos(\theta_{\text{crit}})|) \exp(-\beta^2/2\epsilon^2) \\ \textbf{with} & \times(\Theta(0 < \alpha_1 < 1) + \Theta(0 < \alpha_2 < 1) \\ & +\Theta(\alpha_1 = \alpha_2)) \\ \textbf{where} & \begin{cases} \boldsymbol{u}_4 = \frac{\boldsymbol{x}_2 - \boldsymbol{x}_4}{\|\boldsymbol{x}_2 - \boldsymbol{x}_4\|} \end{cases} \end{array} \tag{A.7}$$

The second case is junction formation (equation A.8), with additional parameters defined in the where clause in equation A.6. The rate function, like zippering, depends on the critical angle and the collision trajectory. If the conditions are met, the propensity increases and a junction is formed if the rule is selected to occur. The rewrite effectively crosses the growing MT over the intermediate MT segment.

$$\longrightarrow \left(\begin{array}{ccccc} & & & & \bullet_5 \\ & & & \diagup & \\ & & & \bigcirc_6 & \\ & & \diagup & & \\ \bigcirc_1 & \text{—} & \blacklozenge_2 & \text{—} & \bigcirc_3 \\ & \diagup & & & \\ \bigcirc_4 & & & & \end{array}\right) \begin{array}{l} \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), \\ (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4), \\ (\boldsymbol{x}_5, \boldsymbol{u}_4), (\boldsymbol{x}_6, \boldsymbol{u}_4) \rangle\!\rangle \end{array}$$

$$\begin{array}{ll} & \hat{\rho}_{\text{junction}}(|\boldsymbol{u}_2 \cdot \boldsymbol{u}_4|/|\cos(\theta_{\text{crit}})|) \exp(-\beta^2/2\epsilon^2) \\ \textbf{with} & \times(\Theta(0 < \alpha_1 < 1) + \Theta(0 < \alpha_2 < 1) \\ & +\Theta(\alpha_1 = \alpha_2)) \\ \textbf{where} & \begin{cases} \boldsymbol{u}_4 = \frac{\boldsymbol{x}_2 - \boldsymbol{x}_4}{\|\boldsymbol{x}_2 - \boldsymbol{x}_4\|} \\ \boldsymbol{x}_6 = \boldsymbol{x}_2 + \frac{L_{max}}{4} \boldsymbol{u}_4 \\ \boldsymbol{x}_5 = \boldsymbol{x}_2 + \frac{L_{max}}{2} \boldsymbol{u}_4 \end{cases} \end{array} \tag{A.8}$$

The third discrete outcome is catastrophe (equation A.9), with additional parameters defined in the where clause in equation A.6. In this case, we have the propensity increasing as long as the MT is on a collision course with the nearby MT segment. The rewrite causes the incoming MT to change from a growing to a retracting state, simulating catastrophe.

$$
\begin{aligned}
\longrightarrow \quad & \begin{pmatrix} \bigcirc_1 - \bigcirc_2 - \bigcirc_3 \\ \bigcirc_4 - \blacksquare_5 \end{pmatrix} \begin{array}{l} \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_3), (\boldsymbol{x}_3, \boldsymbol{u}_3), \\ (\boldsymbol{x}_4, \boldsymbol{u}_4), (\boldsymbol{x}_5, \boldsymbol{u}_5) \rangle\!\rangle \end{array} \\
\textbf{with} \quad & \begin{array}{l} c\,(\exp(-\beta^2/2\epsilon^2))\,(\Theta(0 < \alpha_1 < 1) \\ +\Theta(0 < \alpha_2 < 1) + \Theta(\alpha_1 = \alpha_2)) \end{array}
\end{aligned}
\tag{A.9}
$$

A.7 State Changes

The state change rule in equation A.10 is a bi-directional discrete rule. Growing ends become retracting ends at a rate of $\hat{\rho}_{\text{retract}\leftarrow\text{growth}}$. Retracting ends become growing ends at a rate of $\hat{\rho}_{\text{growth}\leftarrow\text{retract}}$. These rates are typically chosen to be near zero, so that they occur less frequently than the other discrete rules [126]. More rates can be found in [107].

$$
\begin{aligned}
& (\bullet_1)\,\langle\!\langle \boldsymbol{x}_1, \boldsymbol{u}_1 \rangle\!\rangle \quad \longleftrightarrow \quad (\blacksquare_1)\,\langle\!\langle \boldsymbol{x}_1, \boldsymbol{u}_1 \rangle\!\rangle \\
& \quad \textbf{with} \;\; (\hat{\rho}_{\text{retract}\leftarrow\text{growth}}, \hat{\rho}_{\text{growth}\leftarrow\text{retract}})
\end{aligned}
\tag{A.10}
$$

A.8 Calculation of correlation length difference z-score

The first and second experiments produced angle correlation lengths as detailed in the main text of $\xi_1 \pm \sigma_1 = 1.34 \pm 0.039$ and $\xi_2 \pm \sigma_2 = 3.14 \pm 0.06$. We estimate the z-score for the difference $\xi_2 - \xi_1$ as

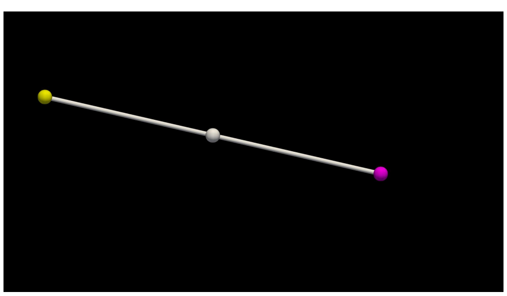
$$
Z = \frac{\xi_2 - \xi_1}{\sigma_2 + \sigma_1} \simeq \frac{3.14 - 1.34}{0.06 + 0.039} \simeq 18.2
$$

The two experiments resulted in different numbers of microtubule segments. If we zoom in on the first experiment window by a factor of 1.6 this population becomes equal, although other parameters of the model become different. So a very generous interpretation of ξ_1 is to multiply it by 1.6 yielding $\hat{\xi}_1 \pm \hat{\sigma}_1 = 1.34 * 1.6 \pm 0.039 * 1.6$ and a z-score of

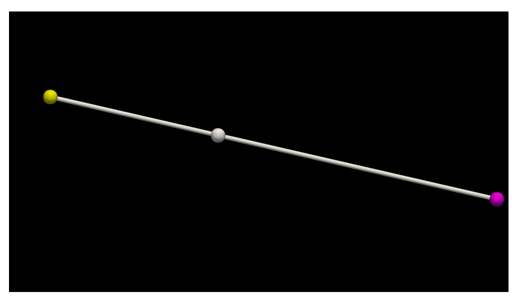
$$\hat{Z} = \frac{\xi_2 - \hat{\xi}_1}{\sigma_2 + \hat{\sigma}_1} \simeq \frac{3.14 - 1.34 * 1.6}{0.06 + 0.039 * 1.6} \simeq 8.14$$

This is larger than the $Z = 5$ standard deviations sometimes used for high certainty because the corresponding p-value for a Gaussian distribution is roughly 3×10^{-7}. Such p-values are $\ll 0.05$, a commonly used threshold for "statistical significance". Of course, we may not have a Gaussian distribution. The worst case distributions however must still obey the Chebyshev inequality under which the p-value must be at most $1/Z^2 \simeq .003 \ll .05$ or even $1/\hat{Z}^2 \simeq .015 \ll .05$, in an extremely conservative calculation.

A.9 Rule Firing Examples



(a) Starting State, No Growth ODE Solving

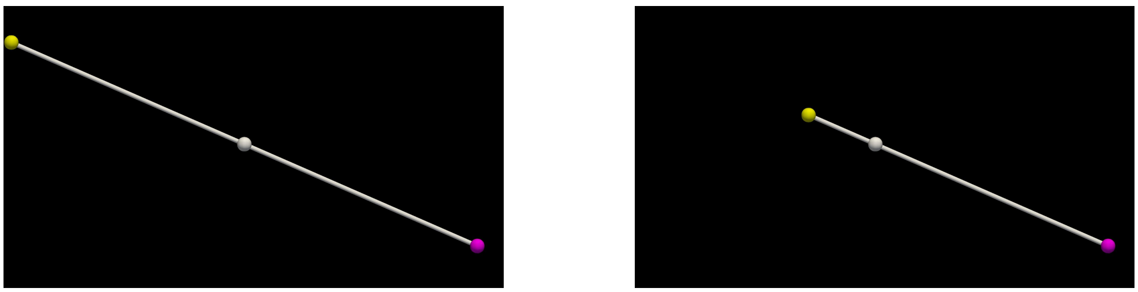


(b) Growth ODE has been solved for Δt

Figure A.1: Example of MT Growth with ODE

(a) Starting State, No Growth Rewrite (b) Ending State, Growth Rewrite Occurred

Figure A.2: Example of MT Growth with Stochastic Polymerization

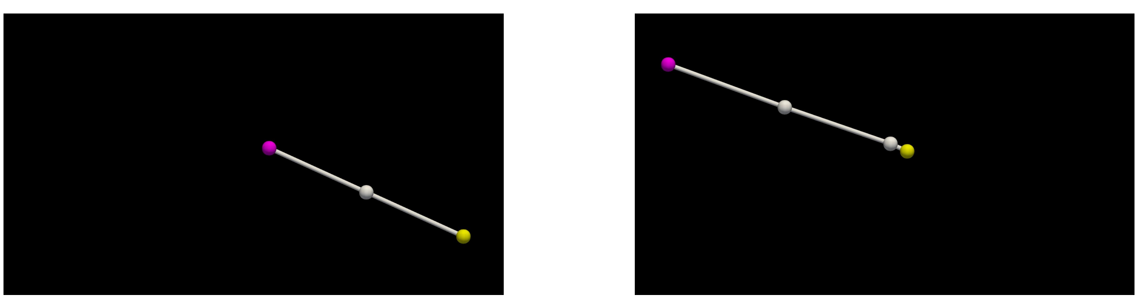


(a) Starting State, No Retraction ODE Solving (b) Retraction ODE has been solved for Δt

Figure A.3: Example of MT Retraction with ODE

(a) Starting State, No Retraction Rewrite (b) Ending State, Retraction Rewrite Occurred

Figure A.4: Example of MT Retraction with Stochastic Depolymerization



(a) Treadmilling IC (b) Treadmilling has Occured

Figure A.5: Example of MT Stochastic and ODE Growth and Retraction Rules Leading to Treadmilling

Appendix B

Periclinal CMA DGG

B.1 Overview and Definitions

There are two different classes of rules, *continuous* and *discrete*. A continuous rule is a rule that updates parameters by solving an ordinary differential equation (ODE). A discrete rule is a rule that rewrites the system graph and may modify parameters by sampling new ones from a factored propensity function. In the main text, we discussed the factorized form of rules and the general form of rules. The discrete rules for this grammar are written in a more compact shorthand form. Newly sampled output values and variables used in rate calculations are sub-parts of a *where* clause. Parameters are exactly sampled to be those values by using a Dirac delta distribution unless otherwise stated. The *where* clause also contains other calculated parameters for convenience.

Rule Parameter	Description
x_n	a point in $\mathbb{R}^2$
u_n	a unit vector in $\mathbb{R}^2$
L	current length of the MT segment

Table B.1: Rule Parameter Definitions

Graph Symbol	Type Name
●	growing
○	intermediate
■	retraction
▲	zipper
⧫	junction
⧄	nucleator
⊡	boundary

Table B.2: Graph node type symbols for the periclinal cortical microtubule array (PCMA) grammar.

Table B.2 lists the symbols used to represent node types in the periclinal cortical microtubule array (PCMA) dynamical graph grammar (DGG) rules. Table B.1 lists the commonly used rule parameters in the grammar and includes a brief description of their meaning. In table B.3 we do the same, but for model parameters. Finally, table B.5 lists commonly used functions and a description as well. Where appropriate, relevant sources for parameters are cited. Otherwise, they are declared as estimated with a citation to where the estimation came from, if applicable.

Model Parameter	Description	Value	Source
L_{div}	the maximal dividing length of a segment	0.075 μm	Estimated [19, 39]
v_{plus}	growth rate	0.0615 μm/sec	[37, 3]
L_{min}	the minimal length of a segment	0.0025 μm	Estimated [19, 39]
v_{minus}	retraction rate	0.00883 μm/sec	[37, 3]
θ_{cic}	CIC angle threshold	40°	Estimated [37, 3, 24]
ϵ	maximal reaction radius	0.1 μm	Estimated [19, 39]
θ_{crit}	critical zippering angle threshold	40°	[37, 6]
σ_{sep}	MT separation distance	0.025 μm	[25]
θ_{cross}	crossover angle threshold	40°	Estimated [37, 3, 24]
θ_{exit}	CLASP entry angle threshold	15°	Estimated [120]
θ_{angle}	CLASP exit angle threshold	15°	Estimated [120]
σ_{col}	MT collision distance	0.025 μm	Estimated [24, 39]
s_{min}	Minimum MT segment initialization length	0.005 μm	Estimated [39]
s_{max}	Minimum MT segment initialization length	0.01 μm	Estimated [39]

Table B.3: Model parameter definitions with default values.

In an alternative model, the rates in table B.4 could also be replaced by propensity functions. The rates themselves are propensities per unit of time, and so in the context of the PCMA (in which the unit of time is seconds) a rate of $40,000$ means that given the rate is active, the rule will fire nearly instantaneously. On the other hand, a rate of 1 means the event occurs on average once per second.

Model Parameter	Description	Propensity Rate	Source
$\hat{\rho}_{grow}$	growth rate factor	100.0	Estimated
$\hat{\rho}_{retract}$	retraction rate factor	10.0	Estimated
$\hat{\rho}_{bnd\_cic\_std}$	boundary CIC standard rate factor	40,000	Estimated
$\hat{\rho}_{bnd\_cic\_clasp}$	boundary CIC for CLASP rate factor	40,000	Estimated
$\hat{\rho}_{int\_cic}$	intermediate CIC rate factor	12,000	Estimated [37]
$\hat{\rho}_{grow\_cic}$	growing end CIC rate factor	12,000	Estimated [37]
$\hat{\rho}_{retract\_cic}$	retracting end CIC rate factor	12,000	Estimated [37]
$\hat{\rho}_{zip\_hit}$	zippering entrainment rate factor	4,000	Estimated [37]
$\hat{\rho}_{zip\_guard}$	zippering guard rate factor	12,000	Estimated [37]
$\hat{\rho}_{cross}$	crossover rate factor	200	Estimated [37]
$\hat{\rho}_{uncross}$	uncrossover rate factor	0.01	Estimated
$\hat{\rho}_{clasp\_entry}$	clasp entry rate factor	0.001	Estimated [120, 97]
$\hat{\rho}_{clasp\_exit}$	clasp exit rate factor	40,000	Estimated [120, 97]
$\hat{\rho}_{clasp\_cat}$	clasp catastrophe rate factor	1,000	Estimated
$\hat{\rho}_{clasp\_detach}$	clasp detachment rate factor	0.016	Estimated
$\hat{\rho}_{destruct}$	destruction rate factor	0.0026	Estimated
$\hat{\rho}_{create}$	creation rate factor	0.0026	Estimated [3]
$\hat{\rho}_{retract\leftarrow \text{growth}}$	retraction to growth conversion rate	0.016	Estimated [126]
$\hat{\rho}_{growth\leftarrow \text{retract}}$	growth to retraction conversion rate	0.016	Estimated [126]

Table B.4: A table of model rate factors with default values.

Function	Description
$M_d(x_1, x_2, x_3)$	Minimum distance from x_1 to the line through x_2 and x_3
$M_\theta(u_1, x_1, u_2)$	Given a point x_1 in the direction u_1 find the angle made with u_2
$H(x;a) = \begin{cases} 1 & x \geq a \\ 0 & x < a \end{cases}$	unit step function
$rotate(x,\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \cdot x$	rotation function
$\Theta(\text{stmt}) = \begin{cases} 1 & \text{if stmt is true} \\ 0 & \text{if stmt is false} \end{cases}$	indication function
$I_p(p_1, u_1, p_2, p_3) = \left[\begin{pmatrix} u_{1x} & -(p_{3x} - p_{2x}) \\ u_{1y} & -(p_{3y} - p_{2y}) \end{pmatrix}^{-1} \cdot \begin{pmatrix} p_{2x} - p_{1x} \\ p_{2y} - p_{1y} \end{pmatrix} \right] = \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix}$	Line intersection

Table B.5: Function Descriptions

B.2 Growing Rules

The positive growth rule models polymerization and effectively elongates the MT. The rule in equation B.1 describes MT growth (polymerization) and is a continuous rule. The position of x_2 is updated by solving a linear ordinary differential equation for constant growth in the direction of the growing node. The velocity for growth, v_{plus}, is set in table B.3.

$$
\begin{aligned}
&(\bigcirc_1 - \bullet_2) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1)(\boldsymbol{x}_2, \boldsymbol{u}_2) \rangle\!\rangle \\
&\longrightarrow (\bigcirc_1 - \bullet_2) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2 + d\boldsymbol{x}_2, \boldsymbol{u}_2) \rangle\!\rangle \\
&\qquad \textbf{solving} \quad d\boldsymbol{x}_2/dt = v_{plus}\, \boldsymbol{u}_2
\end{aligned}
\tag{B.1}
$$

Equation B.2 is a discrete rule. When an MT segment becomes long enough, a new node is added in between the previous two nodes and splits the edge. This process is called "overgrowth". A Heaviside function provides a fully activated rate when a dividing threshold has been reached, and it is multiplied by a growth rate factor. The new node, $\bigcirc_3$, is placed at a point controlled by the γ parameter on the line between x_1 and x_2 and here it is set to $\gamma = 0.01$ to place the new segment right before.

$$
\begin{aligned}
&(\bigcirc_1 - \bullet_2) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2) \rangle\!\rangle \\
&\longrightarrow (\bigcirc_1 - \bigcirc_3 - \bullet_2) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3) \rangle\!\rangle \\
&\qquad \textbf{with} \;\; \hat{\rho}_{\text{grow}}\, H(\|\boldsymbol{x}_2 - \boldsymbol{x}_1\|; L_{\text{div}}) \\
&\qquad \textbf{where} \;\; \begin{cases} \boldsymbol{x}_3 = \boldsymbol{x}_2 - (\boldsymbol{x}_2 - \boldsymbol{x}_1)\gamma \\ \boldsymbol{u}_3 = \frac{\boldsymbol{x}_3 - \boldsymbol{x}_2}{\|\boldsymbol{x}_3 - \boldsymbol{x}_2\|} \end{cases}
\end{aligned}
\tag{B.2}
$$

B.3 Retraction Rules

Negative MT retraction is a continuous rule used to model depolymerization. Equation B.3 is similar to the growth rule in B.1, but instead functions to limit how much the MT segment can retract. The ODE has a constant rate of retraction velocity, v_{minus}, which are set in table B.3. The limit, L_{min}, is primarily added to prevent instability from round-off errors in the ODE solver caused by two points being too close to one another.

$$\begin{aligned}
&(\blacksquare_1 - \bigcirc_2)\langle\!\langle(\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2)\rangle\!\rangle \\
&\longrightarrow (\blacksquare_1 - \bigcirc_2)\langle\!\langle(\boldsymbol{x}_1 + d\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2)\rangle\!\rangle \\
&\quad\ \textbf{solving} \ \ d\boldsymbol{x}_1/dt = v_{minus}(L/L_{\min})\boldsymbol{u}_1
\end{aligned} \tag{B.3}$$

The rule in equation B.4 is a discrete rule used to shorten an MT. In the context of this DGG, the process of shortening is "undergrowth". The rule in equation B.2 inserts more intermediate nodes into the system and the negative MT undergrowth rule removes them. Here, a Heaviside function H scaled by the rate $\hat{\rho}_{retract}$ (table B.4) is used. When the retraction node gets close enough to its intermediate neighbor, i.e. the distance is less than L_{min}, the rate function activates. If this rule is selected to occur, the neighboring intermediate node is removed.

$$\begin{aligned}
&(\blacksquare_1 - \bigcirc_2 - \bigcirc_3)\langle\!\langle(\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3)\rangle\!\rangle \\
&\longrightarrow (\blacksquare_1 - \bigcirc_3)\langle\!\langle(\boldsymbol{x}_1, \boldsymbol{u}_1), \varnothing, (\boldsymbol{x}_3, \boldsymbol{u}_3)\rangle\!\rangle \\
&\quad\ \textbf{with} \ \ \hat{\rho}_{retract}\, H(\|\boldsymbol{x}_2 - \boldsymbol{x}_1\|; L_{min})
\end{aligned} \tag{B.4}$$

B.4 Boundary Catastrophe Rules

The boundary in the PCMA grammar is also represented as a graph. To ensure that MTs cannot grow outside the boundary, we implement two rules. The first rule is the standard boundary collision-induced catastrophe (CIC):

$$
\begin{aligned}
&(\, \bigcirc_1 \text{ — } \bullet_2 \,,\ \boxdot_3 \text{ — } \boxdot_4) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3)(\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\longrightarrow (\, \bigcirc_1 \text{ — } \blacksquare_2 \,,\ \boxdot_3 \text{ — } \boxdot_4) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\qquad \textbf{with } \ \hat{\rho}_{bnd\_cic\_std}\, \Theta(M_d(x_2, x_3, x_4) \leq \sigma_{col}) \\
&\qquad \textbf{where } \ \boldsymbol{u}_2 = -\boldsymbol{u}_2
\end{aligned} \tag{B.5}
$$

If the minimum distance from the incoming MT to the boundary segment is less than the collision distance, the incoming MT enters a catastrophe state. Node $\bullet_2$ changes its type to $\bigcirc_2$. The corresponding unit vector is reversed. The rate $\hat{\rho}_{bnd\_cic\_std}$ is set to a large value so that given this condition, the rule will fire nearly instantaneously. Low rates, would lead to a situation where the "window of opportunity" is missed and the MT exits the domain.

The second rule is another CIC rule, but for the case when a grow MT gets too close to a boundary segment with an attached MT:

$$
\begin{aligned}
&(\, \bigcirc_1 \text{ — } \bullet_2 \,,\ \boxdot_3 \text{ — } \bigcirc_4) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\longrightarrow (\, \bigcirc_1 \text{ — } \blacksquare_2 \,,\ \boxdot_3 \text{ — } \bigcirc_4) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\qquad \textbf{with } \ \hat{\rho}_{bnd\_cic\_clasp}\, \Theta(M_d(x_2, x_3, x_4) \leq \sigma_{col}) \\
&\qquad \textbf{where } \ \boldsymbol{u}_2 = -\boldsymbol{u}_2
\end{aligned} \tag{B.6}
$$

If the minimum distance from the incoming MT to the boundary with an attached MT is less than the collision distance, the incoming MT enters a catastrophe state. Node $\bullet_2$ changes its type to $\bigcirc_2$. The corresponding unit vector is reversed. The rate $\hat{\rho}_{bnd\_cic\_std}$ is set to a large value so that given this condition, the rule will fire nearly instantaneously. Low rates again, would lead to a situation where the "window of opportunity" is missed and the MT exits the domain. If the reader were instead looking to create rules for a particle grammar, it would be worth considering different types of deflections that can occur upon collision. See the zippering rule (equation B.12) for inspiration.

B.5 MT Collision Induced Catastrophe Rules

When two MTs (or possibly itself) hit each other, one outcome that occurs is collision-induced catastrophe (CIC). In the PCMA, there are three distinct types of MT-MT interactions. The following rule is a collision rule that induces catastrophe when a growing end hits an intermediate segment:

$$
\begin{aligned}
&(\ \bigcirc_1 - \bullet_2,\ \bigcirc_3 - \bigcirc_4)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4)\rangle\!\rangle\\
&\longrightarrow (\ \bigcirc_1 - \blacksquare_2,\ \bigcirc_3 - \bigcirc_4)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4)\rangle\!\rangle\\
&\qquad \textbf{with}\ \ \hat{\rho}_{int\_cic}\,\Theta(M_d(x_2,x_3,x_4) \leq \sigma_{col})\,\Theta(\alpha_1 \geq 0)\,\Theta(M_\theta(u_2,x_2,u_3) \geq \theta_{cic})\\
&\qquad \textbf{where}\ \ \boldsymbol{u}_2 = -\boldsymbol{u}_2,\ \text{and}\ \alpha = I_p(x_2,u_2,x_3,x_4)
\end{aligned}
\tag{B.7}
$$

In equation B.7, three conditions determine if the rule can be activated. One, the incoming MT is within the minimum distance for a collision to occur. The expanded cell complex in DGGML enables nearby components to be found, but σ_{col} is used to enforce that they must be even closer. Two, the growing MT must be on a collision course with the intermediate segment, denoted by $\alpha_1 \geq 0$. Three, the angle formed between the incoming MT and the segment it collides with must be above an angle threshold. If all three conditions are true, the rate is then $\hat{\rho}_{int\_cic}$ otherwise it is zero.

The following rule is a collision rule that induces catastrophe when a growing end hits another growing end:

$$
\begin{aligned}
&(\bigcirc_1 — \bullet_2 ,\ \bigcirc_3 — \bullet_4) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\longrightarrow (\bigcirc_1 — \blacksquare_2 ,\ \bigcirc_3 — \bullet_4) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\qquad \textbf{with} \quad \hat{\rho}_{grow\_cic}\, \Theta(M_d(x_2, x_3, x_4) \leq \sigma_{col}/2) \\
&\qquad \textbf{where} \quad \boldsymbol{u}_2 = -\boldsymbol{u}_2
\end{aligned} \tag{B.8}
$$

The logic for a growing end colliding with another growing end in equation B.8 is similar to equation B.7, but instead, we only require the first condition to be true. Given the max length L_{div} of a segment is already close to the actual width of a MT, we make the modeling choice to only implement catastrophe for grow-grow interactions. Further, we set it to be half the distance previously used. We half the collision distance because not doing so would cause "false" collisions to occur since the separation distance of zippered MTs is $\sigma_{sep} = \sigma_{col}$ for this particular model. If the minimum distance condition is true, the rate is then $\hat{\rho}_{grow\_cic}$ otherwise it is zero.

The following rule is a collision rule that induces catastrophe when a growing end hits a retracting end:

$$
\begin{aligned}
&(\bigcirc_1 — \bullet_2 ,\ \bigcirc_3 — \blacksquare_4) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\longrightarrow (\bigcirc_1 — \blacksquare_2 ,\ \bigcirc_3 — \blacksquare_4) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\qquad \textbf{with} \quad \hat{\rho}_{retract\_cic}\, \Theta(M_d(x_2, x_3, x_4) \leq \sigma_{col}/2) \\
&\qquad \textbf{where} \quad \boldsymbol{u}_2 = -\boldsymbol{u}_2
\end{aligned} \tag{B.9}
$$

The logic for a growing end colliding with another growing end in equation B.9 is similar to equation B.8. Again, given the max length, L_{div} of a segment, we make the modeling choice to only implement catastrophe for grow-retract interactions. If the minimum distance condition is true, the rate is then $\hat{\rho}_{retract\_cic}$ otherwise it is zero.

B.6 Crossover Rules

When two MTs (or possibly itself) hit each other, one outcome that occurs is crossover. The following is a rule for crossover:

$$
\begin{aligned}
&(\ \bigcirc_1 - \bullet_2,\ \bigcirc_3 - \bigcirc_4)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4)\rangle\!\rangle \\
&\longrightarrow \left(\begin{array}{ccccc} & & & & \bullet_6 \\ & & & / & \\ & & & \bigcirc_5 & \\ & & / & & \\ \bigcirc_3 & - & \blacklozenge_2 & - & \bigcirc_4 \\ & / & & & \\ \bigcirc_1 & & & & \end{array}\right)
\begin{array}{l}\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3), \\ (\boldsymbol{x}_4,\boldsymbol{u}_4),(\boldsymbol{x}_5,\boldsymbol{u}_2),(\boldsymbol{x}_6,\boldsymbol{u}_2)\rangle\!\rangle\end{array}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{with} \quad \begin{array}{l}\hat{\rho}_{cross}\ \Theta(M_d(x_2,x_3,x_4) \le \sigma_{col})\ \Theta(M_\theta(u_2,x_2,u_3) \ge \theta_{cross}) \\ \times\Theta(\alpha_1 \ge 0)\ \Theta(0 < \alpha_2 < 1)\end{array} \\
&\textbf{where} \quad \begin{cases}\alpha = I_p(x_2,u_2,x_3,x_4) \\ \boldsymbol{x}_2 = \boldsymbol{x}_4 + \alpha_2(\boldsymbol{x}_3 - \boldsymbol{x}_4),\ \boldsymbol{x}_5 = \boldsymbol{x}_2 + 0.01\,\boldsymbol{x}_2,\ \boldsymbol{x}_6 = \boldsymbol{x}_2 + 0.011\,\boldsymbol{x}_2\end{cases}
\end{aligned} \tag{B.10}
$$

In the crossover rule (equation B.10) a junction, $\blacklozenge_2$, is formed. The crossover will only occur if four conditions are met. One, the growing end is within a minimum distance of the intermediate segment. Two, the collision angle must be above an angle threshold. Crossover uses a different angle θ_{cross} than the angle θ_{cic} in the intermediate CIC rule (equation B.7). This allows for the angle that allows the crossover to be set differently than CIC. However, for the default parameters, they are the same. Next, the third and fourth conditions are that a growing end must be growing in the direction of the intermediate segment and will intersect with it given the current trajectory. The condition $\Theta(\alpha_1 \geq 0)$, guarantees the MT is growing towards the intermediate segment and not away. The condition $\Theta(0 < \alpha_2 < 1)$ guarantees the intersection point is on the segment between the points $\boldsymbol{x}_3$ and $\boldsymbol{x}_4$. If the four conditions are met, activating the rate $\hat{\rho}_{cross}$, then the new nodes are placed just ahead of the collision point in the direction of the collision.

For completeness, we have also included a rule to allow MTs to uncross:

$$
\left(\begin{array}{ccccc} & & & & \bigcirc_3 \\ & & & & | \\ \blacksquare_1 & — & \bigcirc_2 & — & \blacklozenge_5 \\ & & & & | \\ & & & & \bigcirc_4 \end{array}\right)
\begin{array}{l} \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), \\ (\boldsymbol{x}_4, \boldsymbol{u}_4), (\boldsymbol{x}_5, \boldsymbol{u}_5) \rangle\!\rangle \end{array}
$$

$$
\longrightarrow \left(\begin{array}{ccccc} & & \bigcirc_3 & — & \bigcirc_4 \\ \blacksquare_1 & — & \bigcirc_2 & — & \bigcirc_5 \end{array}\right)
\begin{array}{l} \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), \\ (\boldsymbol{x}_4, \boldsymbol{u}_4), (\boldsymbol{x}_5, \boldsymbol{u}_2) \rangle\!\rangle \end{array}
$$

$$
\textbf{with} \quad \hat{\rho}_{\text{uncross}}\, \Theta(|\boldsymbol{u}_2 \cdot \boldsymbol{u}_4| \leq 0.95 \text{ and } \|\boldsymbol{x}_4 - \boldsymbol{x}_3\| \leq L_{div}) \tag{B.11}
$$

This uncrossover rule (equation B.11) should not be thought of as katanin severing [35], but rather an example of how to write a rule that reverses the effect of crossover in the case that we don't want the crossover sites to persist. The rule works by finding a retracting end attached to a junction. If the intermediate segment of the MT perpendicular to the retracting end is nearly parallel, we know that it is the crossed MT. A check like this is needed because the LHS pattern could match an intermediate node that is part of the segment associated with the retracting end. So, this allows an invalid pattern to be rejected. We then check if the retracting segment is short enough. If both conditions pass, then the rate for the rule is $\hat{\rho}_{\text{uncross}}$ otherwise it is zero.

B.7 Zippering Rules

When two MTs (or possibly itself) hit each other, one outcome that occurs is zippering. The following is a rule for zippering, where an incoming MT is oriented parallel to the MT it collides with:

$$
\begin{aligned}
&\left(\bigcirc_1 - \bullet_2\,,\ \bigcirc_3 - \bigcirc_4 \right) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\longrightarrow \left(\begin{array}{llll} & \bigcirc_3 & \text{———————} & \bigcirc_4 \\ & \blacktriangle_2 & - \bigcirc_6 - & \bullet_5 \\ / & & & \\ \bigcirc_1 & & & \end{array} \right) \begin{array}{l} \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), \\ \quad (\boldsymbol{x}_4, \boldsymbol{u}_4), (\boldsymbol{x}_5, \boldsymbol{u}_5), (\boldsymbol{x}_6, \boldsymbol{u}_6) \rangle\!\rangle \end{array} \\
&\quad \textbf{with} \quad \hat{\rho}_{\text{zip\_hit}}\, \Theta(M_d(x_2, x_3, x_4) \leq \sigma_{sep})\, \Theta(M_\theta(u_2, x_2, u_3) \leq \theta_{crit})\, \Theta(\alpha_1 \geq 0) \\
&\quad \textbf{where} \quad \begin{cases} \alpha = I_p(x_2, u_2, x_3, x_4) \\ \boldsymbol{u}_2 = \boldsymbol{u}_3,\ \boldsymbol{u}_6 = \boldsymbol{u}_3,\ \boldsymbol{u}_5 = \boldsymbol{u}_3,\ \boldsymbol{x}_6 = \boldsymbol{u}_2 + 0.005\, \boldsymbol{u}_3,\ \boldsymbol{x}_5 = \boldsymbol{u}_2 + 0.01\, \boldsymbol{u}_3 \end{cases}
\end{aligned}
\tag{B.12}
$$

In the zippering rule (equation B.12) the incoming MT only becomes entrained with the other MT if three conditions are met. One, they must be separated by a minimum distance. In this case, σ_{sep}, the separation distance. Second, the angle between them must be less than the critical angle, θ_{crit}. Both of the default values for these parameters can be found in table B.3. Third, the incoming MT must be on a collision course with the other MT i.e. the condition on α_1. If all three conditions are true, then the rate at which zippering occurs is $\hat{\rho}_{\text{zip\_hit}}$ otherwise zero. The growing node ($\bullet_2$) is then transformed into a zippering node ($\blacktriangle_2$) and then two new nodes and edges are added. The new nodes along with node 2 are then oriented to be parallel to the MT they are entrained with.

In a real system, when zippering occurs, the MTs are linked by proteins [106]. Therefore, the gap in-between them can't be entered. The following is a zippering rule that guards the space between two MTs in the absence of modeling linker proteins:

$$
\begin{aligned}
&(\bigcirc_1 — \bullet_2\,,\ \bigcirc_4 — \blacktriangle_3 — \bigcirc_5)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4),(\boldsymbol{x}_5,\boldsymbol{u}_5)\rangle\!\rangle\\
&\longrightarrow\ (\bigcirc_1 — \blacksquare_2\,,\ \bigcirc_4 — \blacktriangle_3 — \bigcirc_5)\ \langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4),(\boldsymbol{x}_5,\boldsymbol{u}_5)\rangle\!\rangle\\
&\quad\textbf{with}\quad \hat{\rho}_{\text{zip\_guard}}\,\Theta(\|\boldsymbol{x}_3-\boldsymbol{x}_2\|\le\sigma_{sep})\\
&\quad\textbf{where}\quad \begin{cases}\boldsymbol{u}_2=-\boldsymbol{u}_2\end{cases}
\end{aligned}
\tag{B.13}
$$

The zippering guard rule (equations B.13) is fairly straightforward. If an incoming MT is within the separation distance of the zippering node, then the rate is $\hat{\rho}_{\text{zip\_guard}}$ otherwise zero. Accordingly, if the rule fires, the incoming MT enters a catastrophe state and the unit vector is reversed.

Finally, we have included a rule that allows an MT to effectively unzipper from another. The following is the unzippering rule:

$$
\begin{aligned}
&(\blacksquare_1 — \bigcirc_2 — \blacktriangle_3)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3)\rangle\!\rangle\\
&\longrightarrow (\blacksquare_1 — \bigcirc_3)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),\varnothing,(\boldsymbol{x}_3,\boldsymbol{u}_3)\rangle\!\rangle\\
&\qquad\textbf{with}\quad \hat{\rho}_{retract}\,H(\|\boldsymbol{x}_2-\boldsymbol{x}_1\|;L_{min})
\end{aligned}
\tag{B.14}
$$

The unzippering rule (equation B.14) works the same as the retraction rule in equation B.4. The difference is we have a retracting node attached to an intermediate node attached to a zippering node. When this rule fires, the zippering node is simply removed and the MT continues depolymerizing in parallel to the MT it had entrained with.

B.8 CLASP Rules

The PCMA model has four rules to approximate the effects CLASP may have on the boundary of the periclinal face. The first rule, models MTs entering the periclinal face from the anticlinal face:

$$
\begin{aligned}
&\left(\boxdot_1 \;\text{———}\; \boxdot_2 \right) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2) \rangle\!\rangle \\
\longrightarrow\; &\left(\begin{array}{ccccc} \boxdot_1 & \text{—} & \bigcirc_3 & \text{—} & \boxdot_2 \\ & & | & & \\ & & \bullet_4 & & \end{array} \right) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\quad \textbf{with} \quad \hat{\rho}_{\text{claps\_entry}} \\
&\quad \textbf{where} \quad \begin{cases} \boldsymbol{x}_3 = (\boldsymbol{x}_1 + \boldsymbol{x}_2)/2, \; \boldsymbol{u}_4 = \boldsymbol{u}_3 \\ \theta_{rot} \sim \mathcal{U}(-\theta_{entry}, \theta_{entry}), \; \boldsymbol{x}_4 = rotate(\boldsymbol{x}_3 + \sigma_{col}\, \boldsymbol{u}_3, \theta_{rot}) \end{cases}
\end{aligned}
\tag{B.15}
$$

The CLASP entry rule (equation B.15) is fairly straightforward. If there is an available boundary edge for a new MT to emerge from it occurs at a rate of $\hat{\rho}_{\text{claps\_entry}}$. The new MT is perpendicular to the boundary and the growing end is set at a distance of σ_{col} away. The segment is then rotated by a randomly sampled entry angle.

The second rule models MT exiting the periclinal face into the anticlinal face:

$$
\begin{aligned}
&\left(\begin{array}{ccc} \boxdot_1 & \text{———} & \boxdot_2 \\ \bigcirc_3 & \text{—} & \bullet_4 \end{array}\right) \langle\!\langle (\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4) \rangle\!\rangle \\
\longrightarrow\ &\left(\begin{array}{ccccc} \boxdot_1 & \text{—} & \bigcirc_3 & \text{—} & \boxdot_2 \\ & & \mid & & \\ & & \bigcirc_4 & & \end{array}\right) \langle\!\langle (\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4) \rangle\!\rangle \\
&\quad \textbf{with} \quad \hat{\rho}_{\text{clasp\_exit}}\, \Theta(M_d(x_4,x_1,x_2) \le \sigma_{col})\, \Theta(M_\theta(x_4,u_4,u_1) \le \theta_{exit}) \\
&\qquad\qquad \times\Theta(\alpha_1 \ge 0)\, \Theta(0 < \alpha_2 < 1) \\
&\quad \textbf{where} \quad \begin{cases} \alpha = I_p(x_4,u_4,x_1,x_2),\ \boldsymbol{x}_3 = \boldsymbol{x}_1 + \alpha_2(\boldsymbol{x}_1 - \boldsymbol{x}_2) \end{cases}
\end{aligned}
\tag{B.16}
$$

The CLASP exit rule (equation B.16) is similar to the crossover rule in equation B.10. There are four conditions required for the rate to be $\hat{\rho}_{\text{clasp\_exit}}$ otherwise the rate is zero. First, the incoming MT must be within the minimum distance of the boundary. Second, the incoming MT must be within the exit angle threshold. In the third and fourth conditions, the MT must be on a collision path with the boundary. The MT then "exits" the boundary at the location where it would have collided.

$$
\begin{aligned}
&\left(\begin{array}{ccccc} \boxdot_1 & \text{—} & \bigcirc_3 & \text{—} & \boxdot_2 \\ & & \mid & & \\ & & \bigcirc_4 & & \end{array}\right) \langle\!\langle (\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4) \rangle\!\rangle \\
\longrightarrow\ &\left(\begin{array}{ccc} \boxdot_1 & \text{———} & \boxdot_2 \\ \blacksquare_3 & \text{—} & \bigcirc_4 \end{array}\right) \langle\!\langle (\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4) \rangle\!\rangle \\
&\quad \textbf{with} \quad \hat{\rho}_{\text{clasp\_detach}}
\end{aligned}
\tag{B.17}
$$

The third rule models an MT depolymerizing in the anticlinal plane and entering the periclinal in a catastrophe state. The CLASP detachment rule (equation B.17), is very simple. Given a rate $\hat{\rho}_{\text{clasp\_detach}}$ the MT will detach from the boundary every so often with a frequency directly related to how small or large the rate is.

The fourth rule models the opposite of equation B.17 and functions to model an MT depolymerizing from the periclinal plane into the anticlinal:

$$
\begin{aligned}
&\left(\begin{array}{ccccc} \boxdot_1 & \text{——} & \bigcirc_3 & \text{——} & \boxdot_2 \\ & & | & & \\ & & \blacksquare_4 & & \end{array}\right) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\longrightarrow \;\; (\, \boxdot_1 \text{————} \boxdot_2) \;\; \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\quad \textbf{with} \quad \hat{\rho}_{\text{clasp\_cat}}\, \Theta(\|\boldsymbol{x}_4 - \boldsymbol{x}_3\| \leq \epsilon)
\end{aligned}
\tag{B.18}
$$

The CLASP catastrophe rule (equation B.18), is also very simple. The MT will detach from the boundary under the condition that the retracting MT is short enough. Given the condition is met, the rate of removal is $\hat{\rho}_{\text{clasp\_cat}}$ and the MT will be removed.

B.9 Destruction and Creation Rules

The PCMA grammar includes two destruction rules and one creation rule. The following is the destruction rule to remove an MT that becomes too short:

$$
\begin{aligned}
&(\, \blacksquare_1 \text{——} \bigcirc_2 \text{——} \blacksquare_3) \langle\!\langle (\boldsymbol{x}_1, \boldsymbol{u}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3) \rangle\!\rangle \longrightarrow \varnothing \\
&\quad \textbf{with} \quad \hat{\rho}_{\text{destruct}}\, \Theta(\|\boldsymbol{x}_1 - \boldsymbol{x}_2\| \leq s_{min})\, \Theta(\|\boldsymbol{x}_3 - \boldsymbol{x}_2\| \leq s_{min})
\end{aligned}
\tag{B.19}
$$

The short MT destruction rule (equation B.19) will occur when two conditions are met. One, the left segments are shorter than a minimum length s_{min}. The second condition is identical, but for the other segment. If both conditions are met the rate is $\hat{\rho}_{\text{destruct}}$ otherwise it is zero. If the destruction rate is small, the MT will exist for a longer period of time before removal when compared to a large value for the rate. Also, as a modeling choice, the destruction rule is specific to the case when an MT has two retracting ends. We consider the MT unrecoverable if the two segments become short enough.

Before describing the second destruction rule, we describe the creation rule and introduce the nucleator node:

$$
\begin{aligned}
&(\boxslash_1)\, \langle\!\langle (\boldsymbol{x}_1) \rangle\!\rangle \longrightarrow (\boxslash_1\,,\ \blacksquare_2 \text{ --- } \bigcirc_3 \text{ --- } \bullet_4)\, \langle\!\langle (\boldsymbol{x}_1), (\boldsymbol{x}_2, \boldsymbol{u}_2), (\boldsymbol{x}_3, \boldsymbol{u}_3), (\boldsymbol{x}_4, \boldsymbol{u}_4) \rangle\!\rangle \\
&\quad \textbf{with} \quad \hat{\rho}_{\text{create}} \\
&\quad \textbf{where} \begin{cases} \boldsymbol{x}_3 \sim \mathcal{U}([\boldsymbol{x}_{1x} - 0.5\epsilon,\ \boldsymbol{x}_{1x} + 0.5\epsilon]\,[\boldsymbol{x}_{1y} - 0.5\epsilon,\ \boldsymbol{x}_{1y} + 0.5\epsilon]) \\ \theta_{rot} \sim \mathcal{U}(0, 2\pi), \text{ and } d \sim \mathcal{U}(s_{min}, s_{max}) \text{ so } \boldsymbol{x}_4 = rotate(\boldsymbol{x}_3 + d, \theta_{rot}) \\ \boldsymbol{x}_2 = \boldsymbol{x}_4 - 2d,\ \boldsymbol{u}_4 = \frac{\boldsymbol{x}_3 - \boldsymbol{x}_4}{\|\boldsymbol{x}_3 - \boldsymbol{x}_4\|},\ \boldsymbol{u}_3 = \boldsymbol{u}_4,\ \boldsymbol{u}_2 = \boldsymbol{u}_4 \end{cases}
\end{aligned}
\tag{B.20}
$$

The creation rule (equation B.20) matches a nucleator node. The nucleator node is a graph node introduced into the PCMA grammar to allow new MTs to be created in the absence of modeling concentrations by using reaction-diffusion equations [129]. The rule uses the position of the nucleator, x_1, as the center of a uniform sampling box with the diameter of half of the reaction radius, ϵ. The intermediate node position, x_3 is sampled from this box. The point x_4 is put a distance away from x_3 based on the minimum and maximum length an MT segment can be initialized to and then it is randomly rotated. The retracting end node position x_2 is calculated using x_4. Following the initialization of the positions, the correct unit vectors are created.

The following rule is the second destruction rule and works to remove an MT that is created too close to an existing MT:

$$
\begin{aligned}
&(\ \blacksquare_1 - \bigcirc_2 - \bullet_3,\ \bigcirc_4)\,\langle\!\langle(\boldsymbol{x}_1,\boldsymbol{u}_1),(\boldsymbol{x}_2,\boldsymbol{u}_2),(\boldsymbol{x}_3,\boldsymbol{u}_3),(\boldsymbol{x}_4,\boldsymbol{u}_4)\rangle\!\rangle \longrightarrow \varnothing \\
&\quad \textbf{with} \quad \hat{\rho}_{\text{destruct}}\,\Theta(\|\boldsymbol{x}_4 - \boldsymbol{x}_3\| \leq \sigma_{col})
\end{aligned} \tag{B.21}
$$

The nearby MT destruction rule (equation B.21) removes nearby MTs with a rate of $\hat{\rho}_{\text{destruct}}$ if a single condition is met. Otherwise, the propensity for the nearby MT destruction rule is zero. The condition for removal is met when the newly created MT is within the collision distance of another intermediate node not included in the new MT. There is evidence that new MTs are created near existing MTs [3, 39], but for the PCMA grammar these effects are not included.

B.10 State Changes

The state change rule in equation B.22 is a bi-directional discrete rule. Growing ends become retracting ends at a rate of $\hat{\rho}_{\text{retract}\leftarrow\text{growth}}$. Retracting ends become growing ends at a rate of $\hat{\rho}_{\text{growth}\leftarrow\text{retract}}$. These rates are typically chosen to be near zero so that they occur less frequently than the other discrete rules [126]. More rates can be found in [107].

$$
\begin{aligned}
&(\bullet_1)\,\langle\!\langle\boldsymbol{x}_1,\boldsymbol{u}_1)\rangle\!\rangle \quad \longleftrightarrow \quad (\blacksquare_1)\,\langle\!\langle\boldsymbol{x}_1,\boldsymbol{u}_1)\rangle\!\rangle \\
&\quad \textbf{with} \quad (\hat{\rho}_{\text{retract}\leftarrow\text{growth}}, \hat{\rho}_{\text{growth}\leftarrow\text{retract}}) \\
&\quad \textbf{where} \quad \begin{cases} \boldsymbol{u}_1 = -\boldsymbol{u}_1 \end{cases}
\end{aligned} \tag{B.22}
$$

B.11 A Note on Rates, Boundary, and Creation

This section discusses how rates work within the model. A rule with a propensity of 1 means given the condition a rule firing can occur, it will with an average of 1 per unit of time. In the case of the PCMA, time is measured in seconds. Therefore, a rate of 1 gives us an event occurrence of once a second. A rate of 60 is then 60 times per minute. A rate of $1/60 \approx 0.016$ means the event will occur on average once per minute. When rates are set very high, i.e. $12,000$, the rule fires nearly instantaneously. When two rules could occur at the same time, such as crossover or zippering, having an equally high rate means they are both equally likely outcomes. Lowering one or the other will then shift the odds.

There are other cases, where we have included boundary nodes and edges to allow new MTs to enter or exit. In this case, the spacing between these nodes is set to be the reaction radius of 0.1. Meaning, that at any one of the edges, the default entry rate is 0.001 or approximately every 16 minutes. In the square case, where we have a $5\mu\text{m} \times 5\mu\text{m}$ domain, there are 195 boundary nodes and 194 edges. So, the rate of entry is $0.001 \times 194 \approx 0.194$ or about every 11 seconds.

For the nucleators, they have been initialized uniformly within the domain. There are $46 \times 46 = 2116$ nucleators. Each one is meant to cover a $0.1\mu\text{m} \times 0.1\mu\text{m}$ region. Using the default rate allows for new MTs to be nucleated at a rate of $10/\mu\text{m}^2$ as estimated in [3].

B.12 Additional Plots

B.12.1 Curve Fits

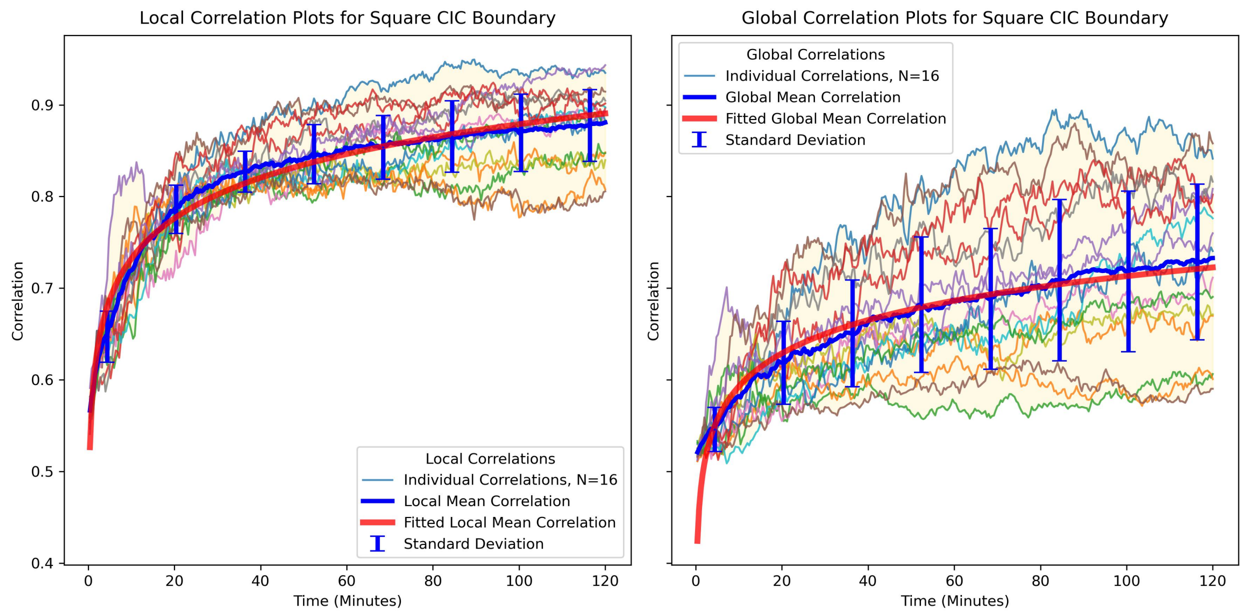


Figure B.1: For the square experiment with CIC Boundary, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

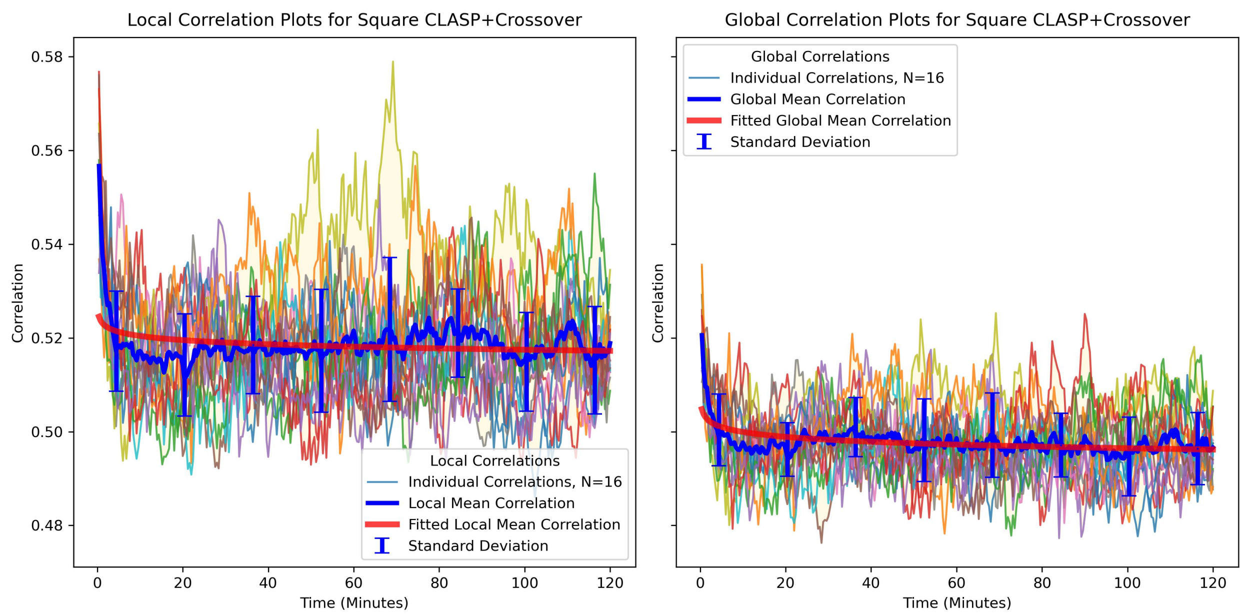


Figure B.2: For the square with CLASP+Crossover where the crossover rate is $8,000$, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

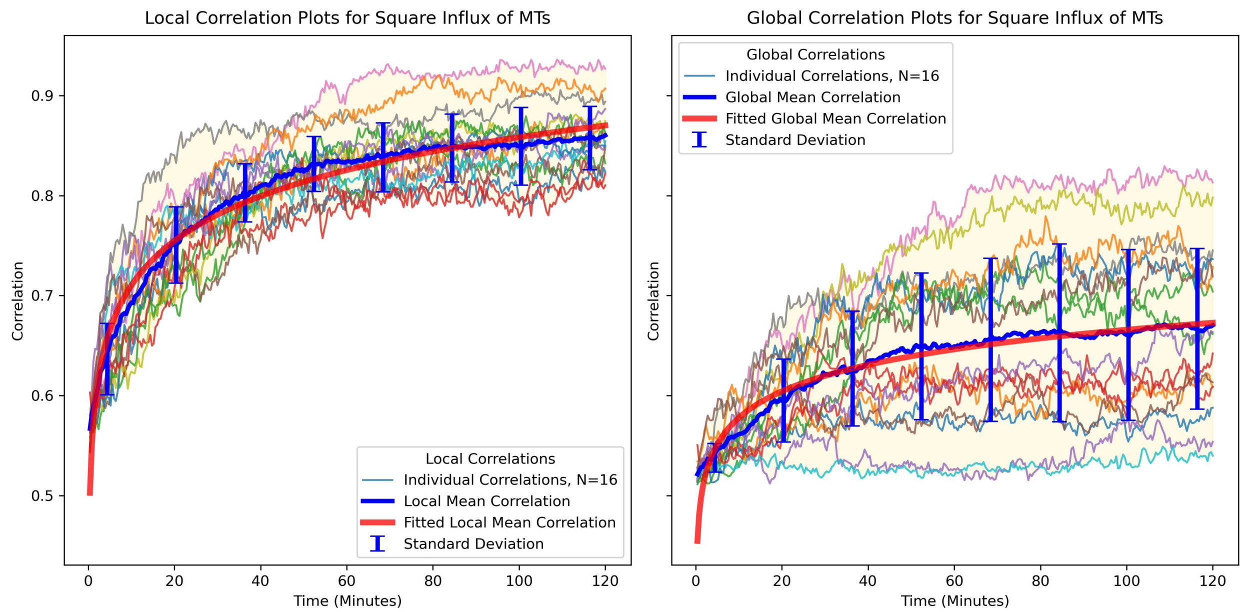


Figure B.3: For the square experiment with an influx of MTs experiment where the entry rate is 0.008, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

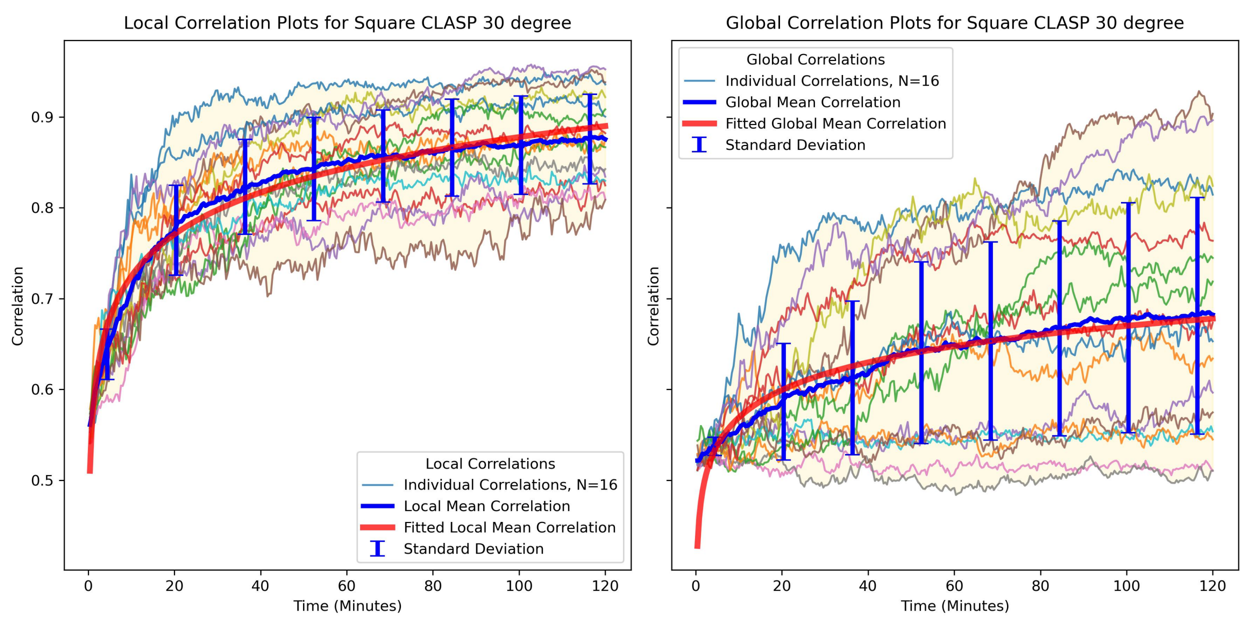


Figure B.4: For the square CLASP 30-degree experiment, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

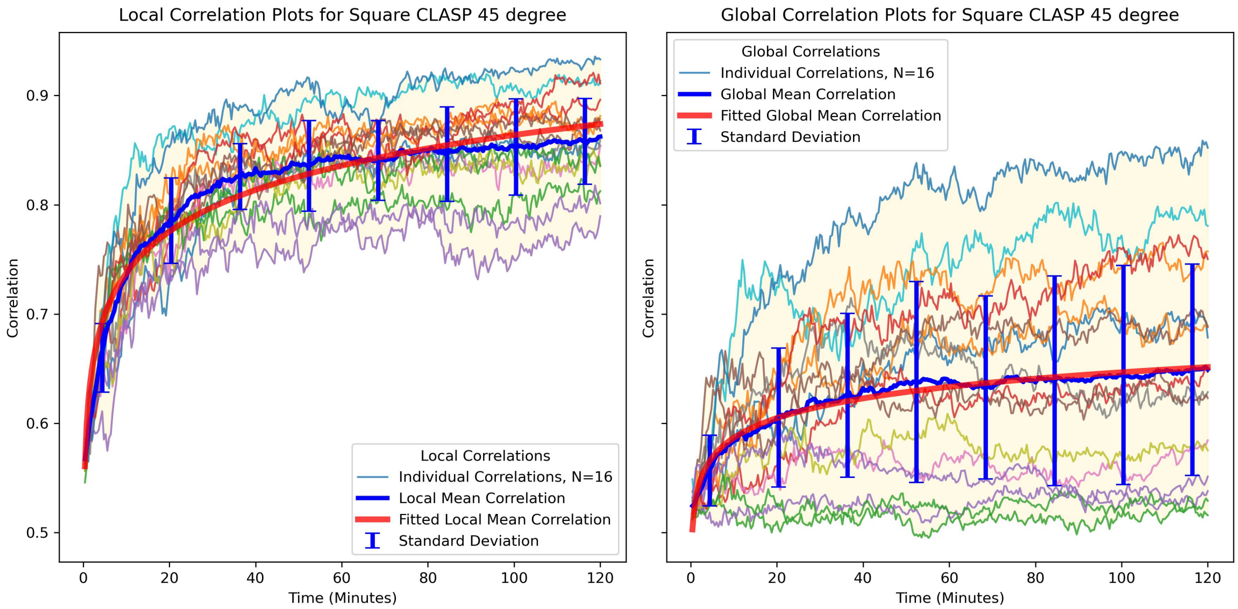


Figure B.5: For the square CLASP 45-degree experiment, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

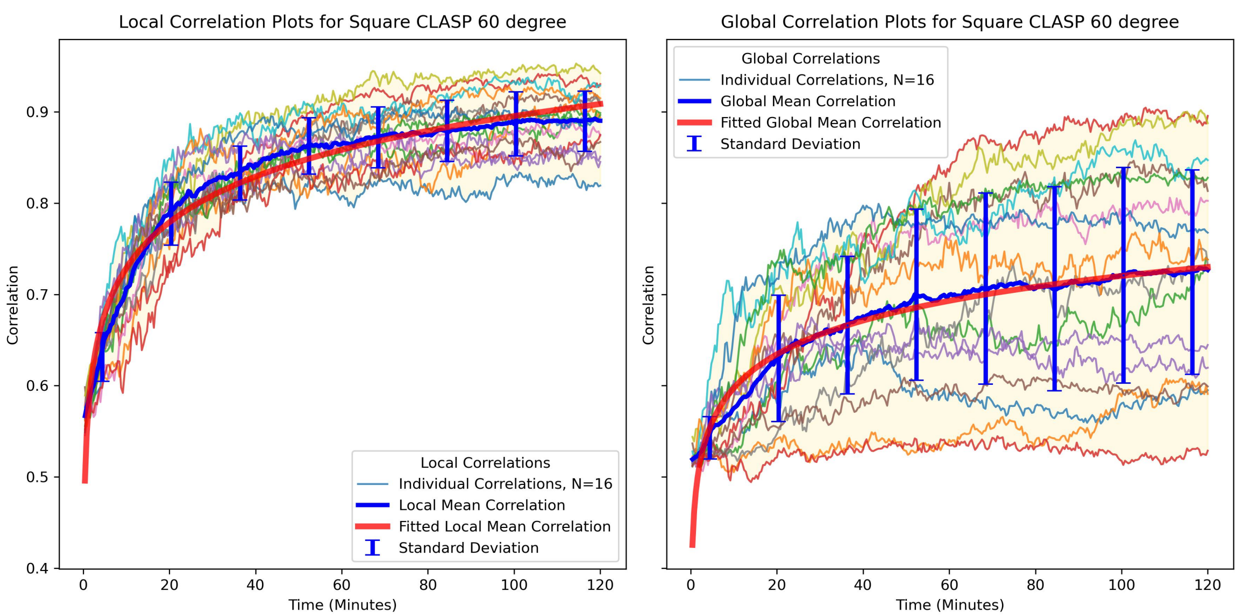


Figure B.6: For the square CLASP 60-degree experiment, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

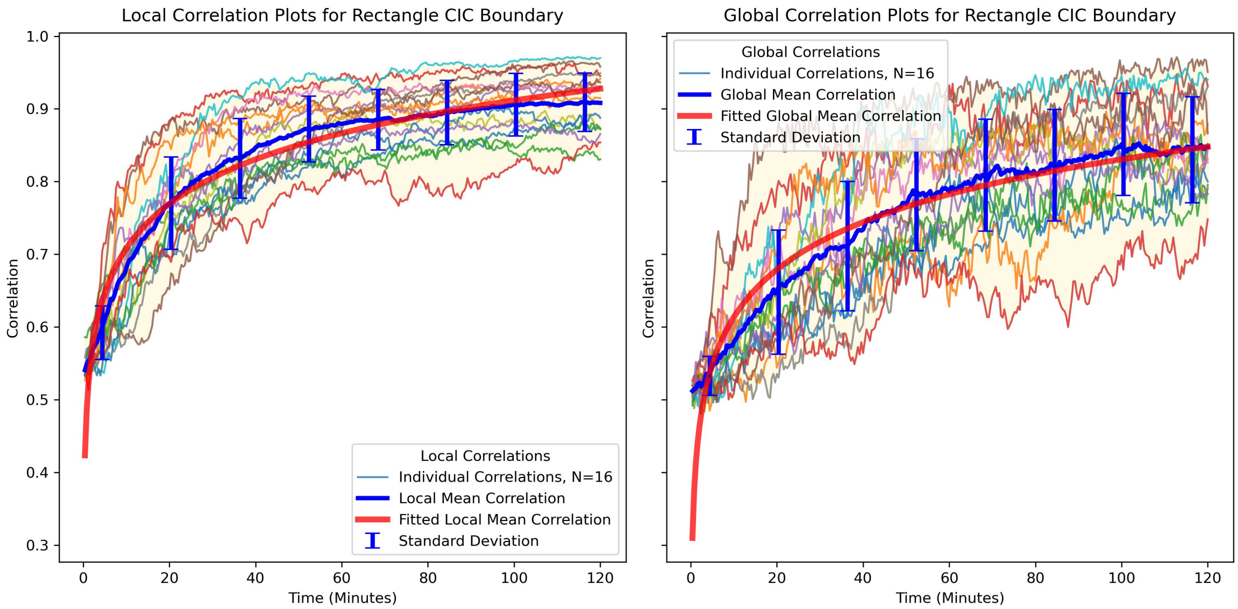


Figure B.7: For the rectangular experiment with CIC Boundary, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

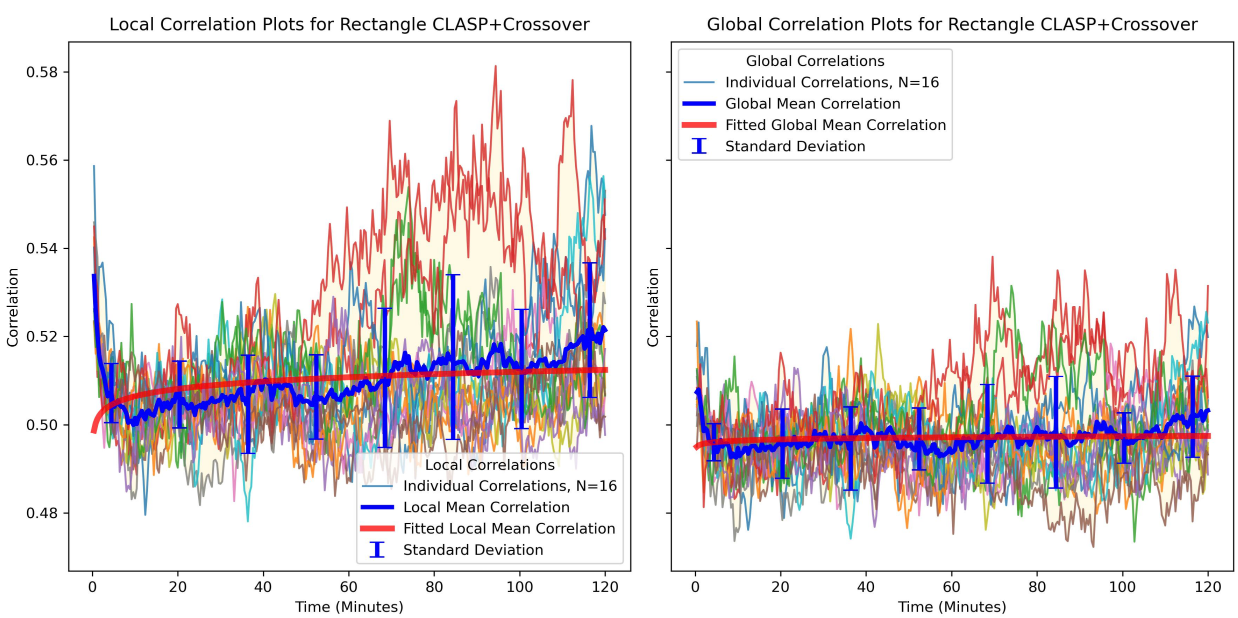


Figure B.8: For the rectangular experiment with CLASP+Crossover where the crossover rate is $8,000$, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

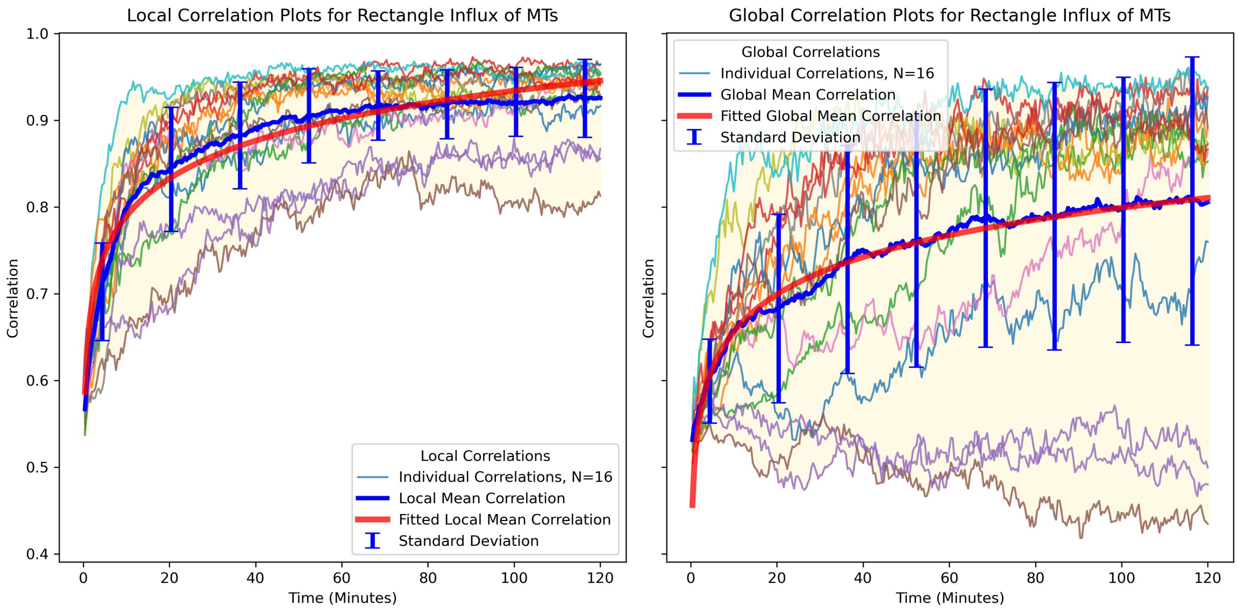


Figure B.9: For the rectangular influx of MTs experiment where the entry rate is 0.008, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

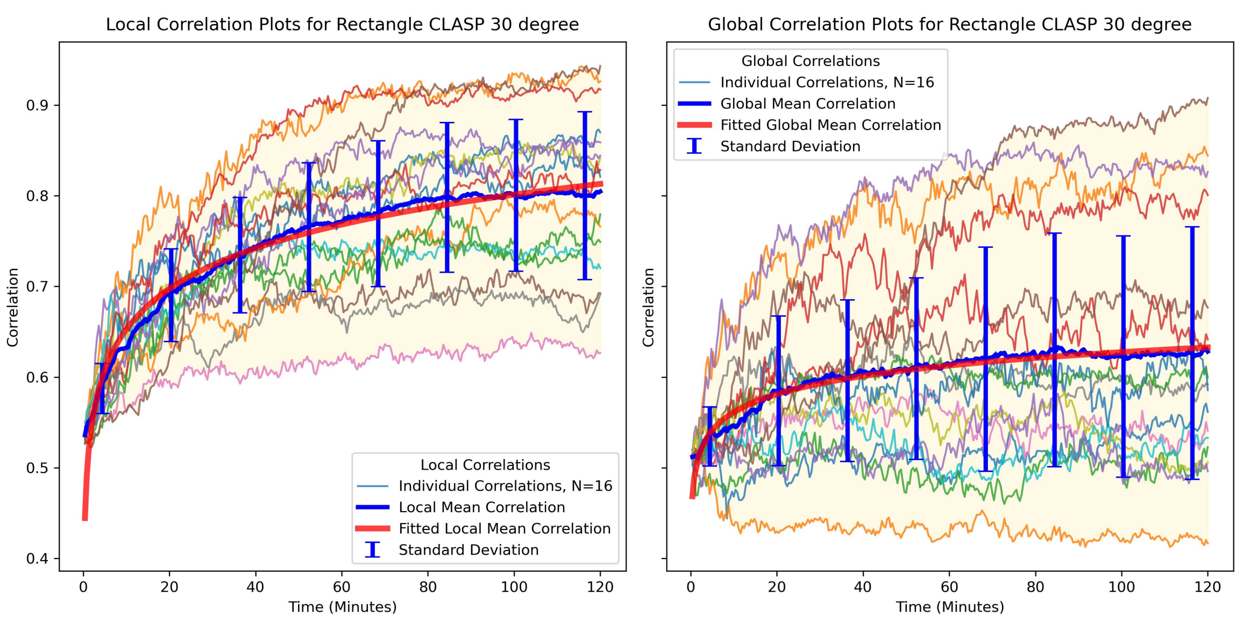


Figure B.10: For the rectangular CLASP 30-degree experiment, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

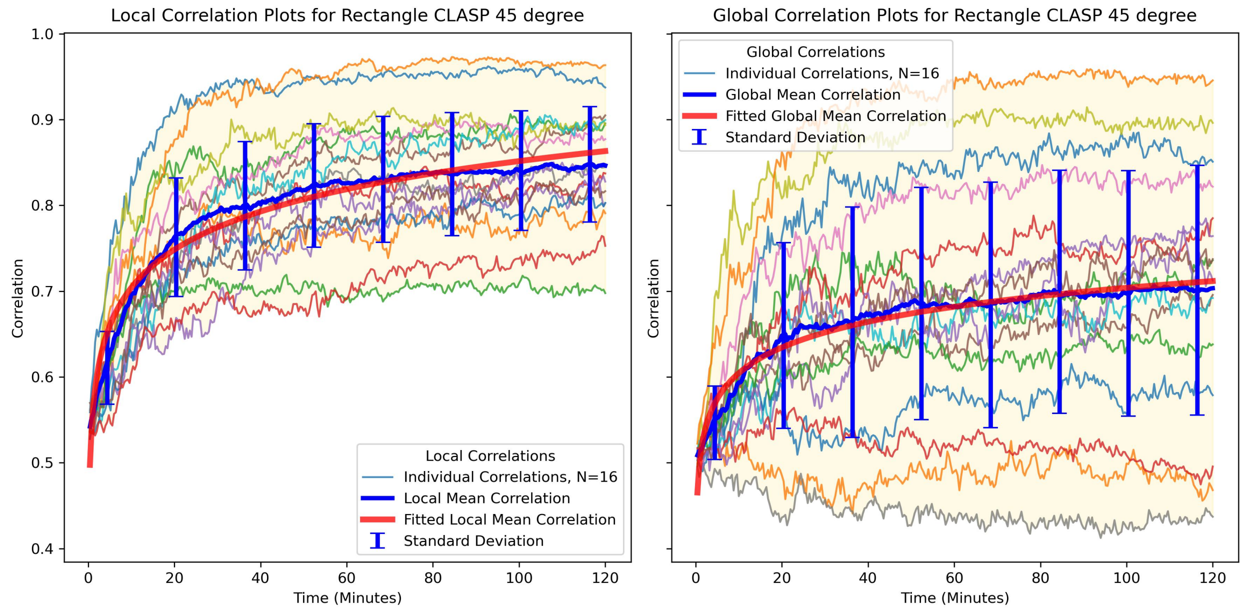


Figure B.11: For the rectangular CLASP 45-degree experiment, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

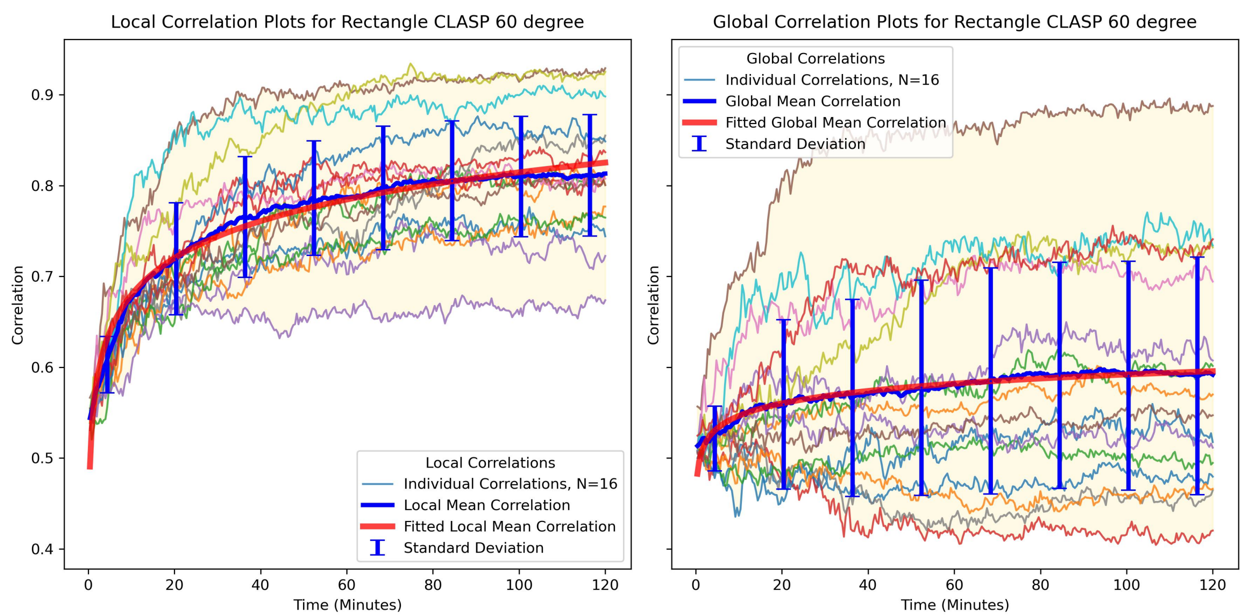
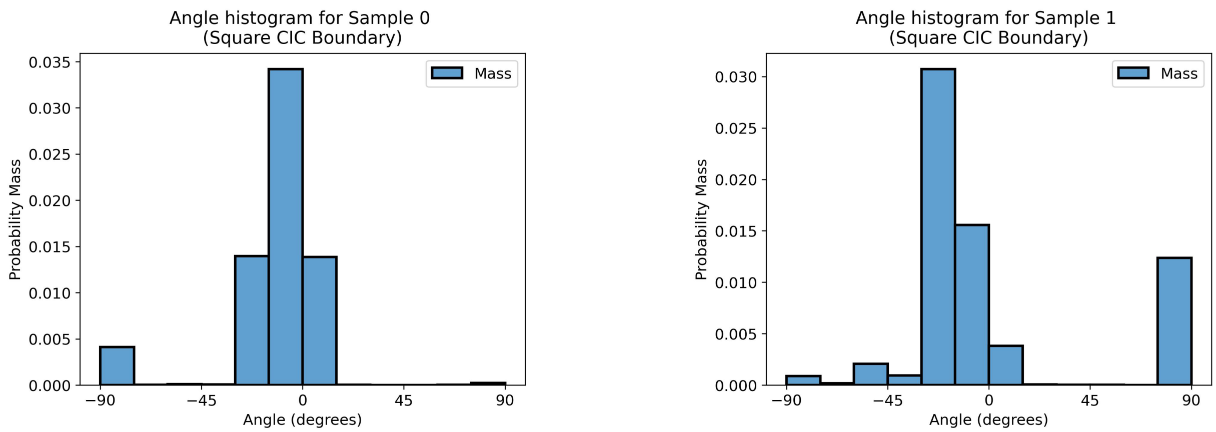


Figure B.12: For the rectangular CLASP 60-degree experiment, the plot contains the following: all $N = 16$ samples of local correlations (left) and global correlations (right), the corresponding mean of all the samples with the standard deviations, and the best-fit curve for the mean.

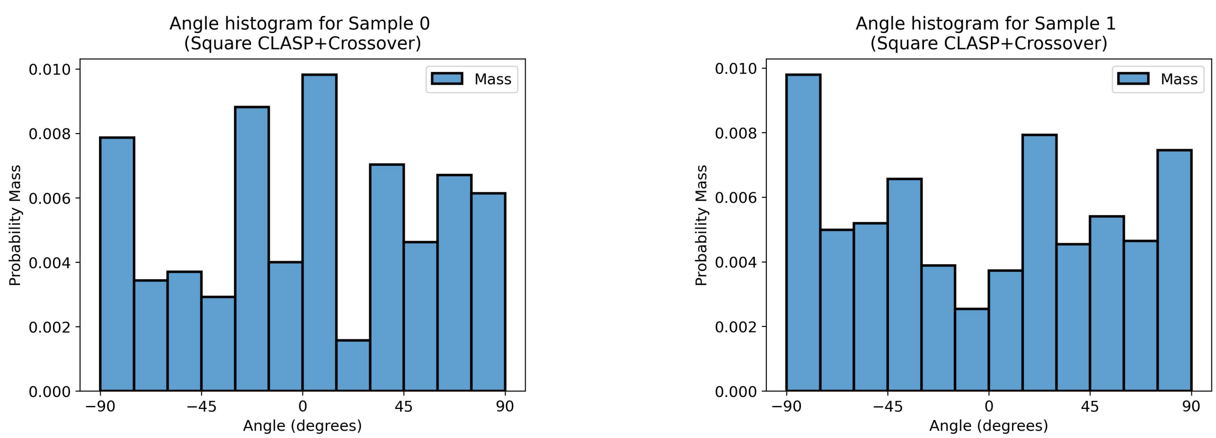
B.12.2 Selected Histogram Examples Fits



(a) Histogram of array orientation for sample 0 of the experiment square domain with CIC boundary.

(b) Histogram of array orientation for sample 1 of the experiment square domain with CIC boundary.

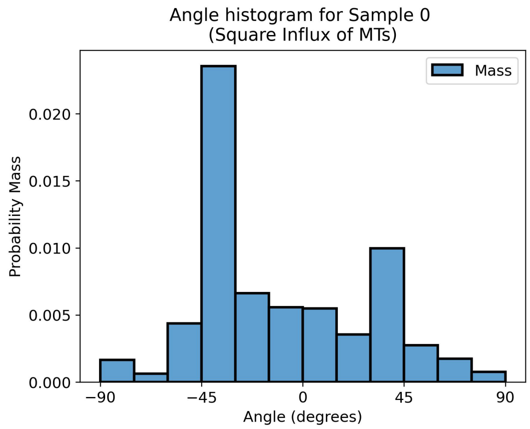
Figure B.13: Two samples of histograms used to calculate the mean histogram for the experiment square domain with CIC boundary.



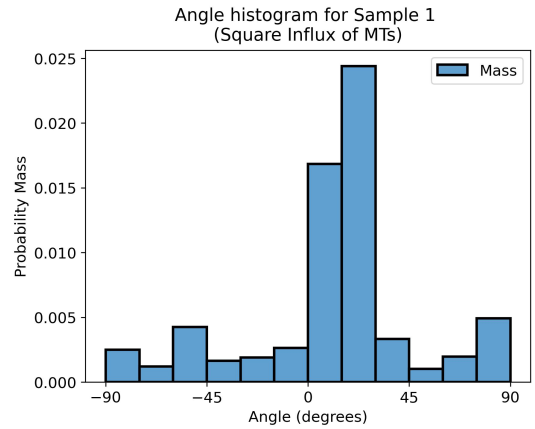
(a) Histogram of array orientation for sample 0 of the experiment square domain with CLASP+Crossover

(b) Histogram of array orientation for sample 1 of the experiment square domain with CLASP+Crossover

Figure B.14: Two samples of histograms used to calculate the mean histogram for the experiment square domain with CLASP+Crossover

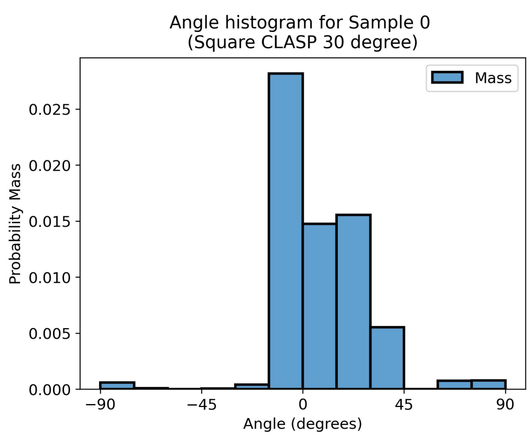


(a) Histogram of array orientation for sample 0 of the experiment square domain with influx of MTs.

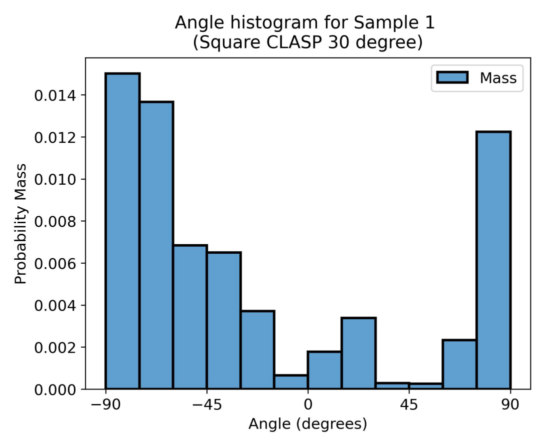


(b) Histogram of array orientation for sample 1 of the experiment square domain with influx of MTs.

Figure B.15: Two samples of histograms used to calculate the mean histogram for the experiment square domain with influx of MTs.

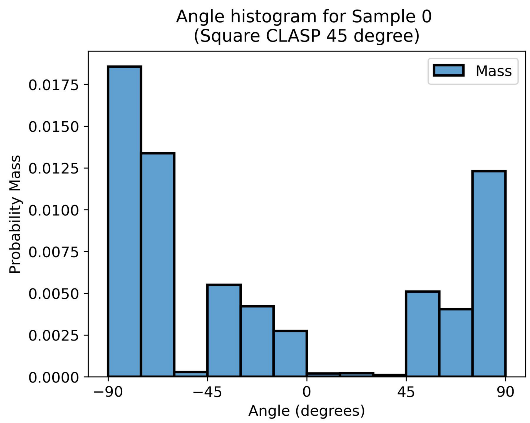


(a) Histogram of array orientation for sample 0 of the experiment square domain with CLASP 30 degrees.

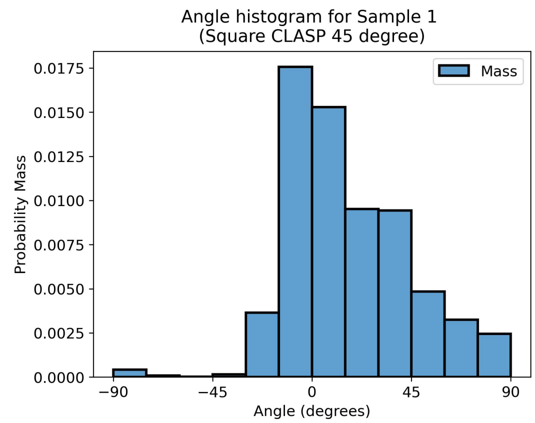


(b) Histogram of array orientation for sample 1 of the experiment square domain with CLASP 30 degrees.

Figure B.16: Two samples of histograms used to calculate the mean histogram for the experiment square domain with CLASP 30 degrees.

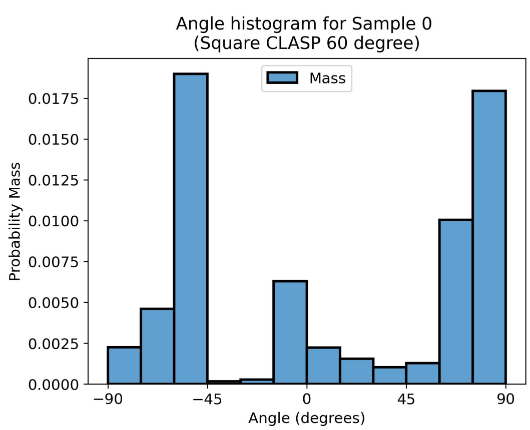


(a) Histogram of array orientation for sample 0 of the experiment square domain with CLASP 45 degrees.

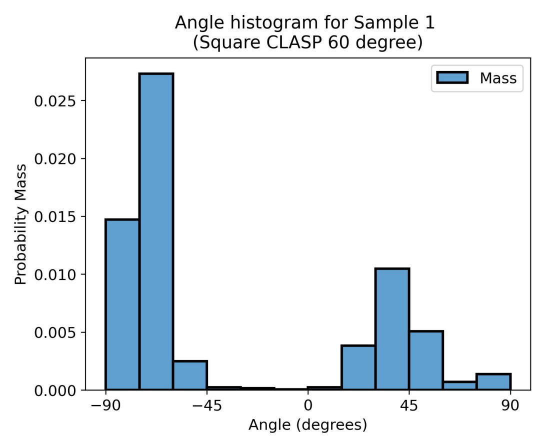


(b) Histogram of array orientation for sample 1 of the experiment square domain with CLASP 45 degrees.

Figure B.17: Two samples of histograms used to calculate the mean histogram for the experiment square domain with CLASP 45 degrees.

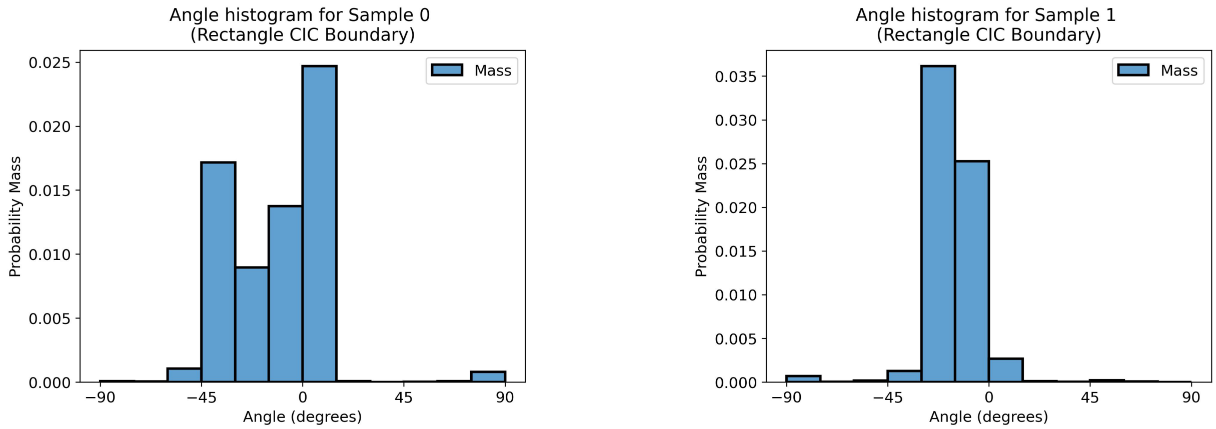


(a) Histogram of array orientation for sample 0 of the experiment square domain with CLASP 60 degrees.



(b) Histogram of array orientation for sample 1 of the experiment square domain with CLASP 60 degrees.

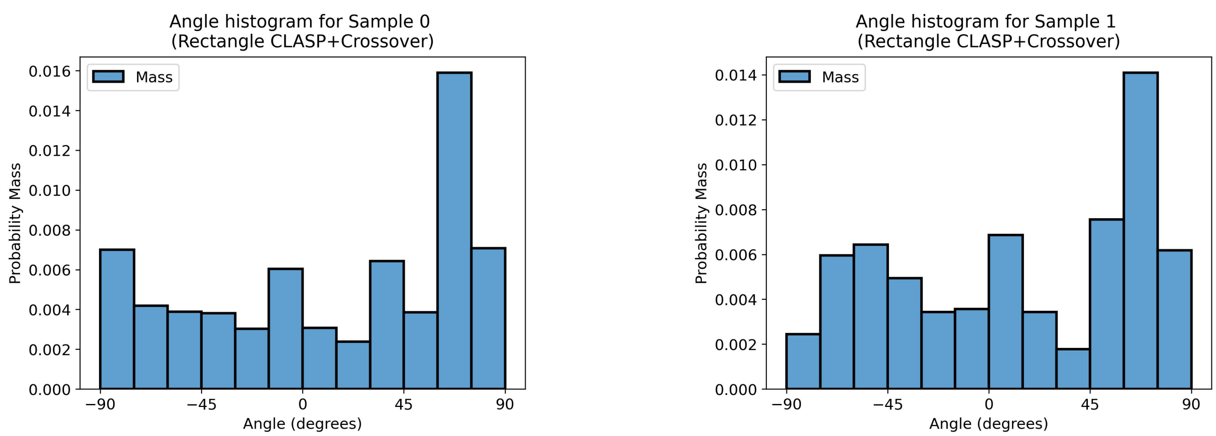
Figure B.18: Two samples of histograms used to calculate the mean histogram for the experiment square domain with CLASP 60 degrees.



(a) Histogram of array orientation for sample 0 of the experiment rectangular domain with CIC boundary.

(b) Histogram of array orientation for sample 1 of the experiment rectangular domain with CIC boundary.

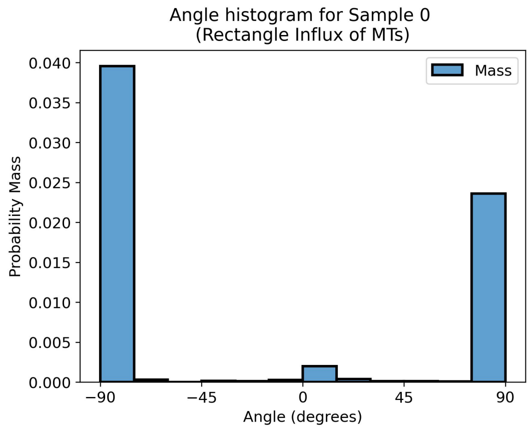
Figure B.19: Two samples of histograms used to calculate the mean histogram for the experiment rectangular domain with CIC boundary.



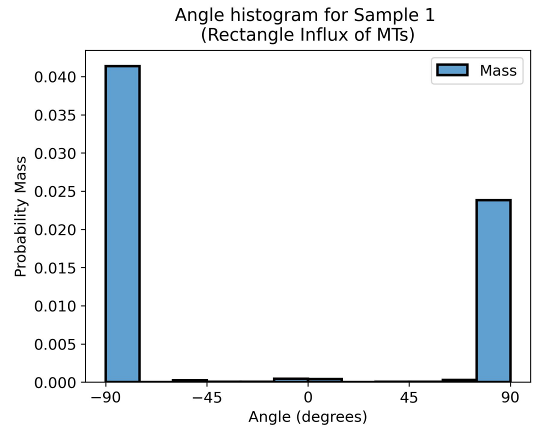
(a) Histogram of array orientation for sample 0 of the experiment rectangular domain with CLASP+Crossover.

(b) Histogram of array orientation for sample 1 of the experiment rectangular domain with CLASP+Crossover.

Figure B.20: Two samples of histograms used to calculate the mean histogram for the experiment rectangular domain with CLASP+Crossover.

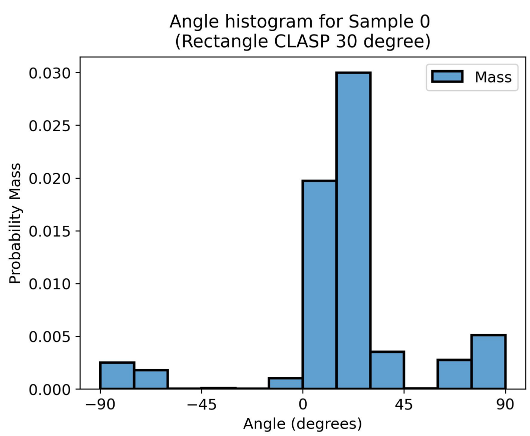


(a) Histogram of array orientation for sample 0 of the experiment rectangular domain with influx of MTs.

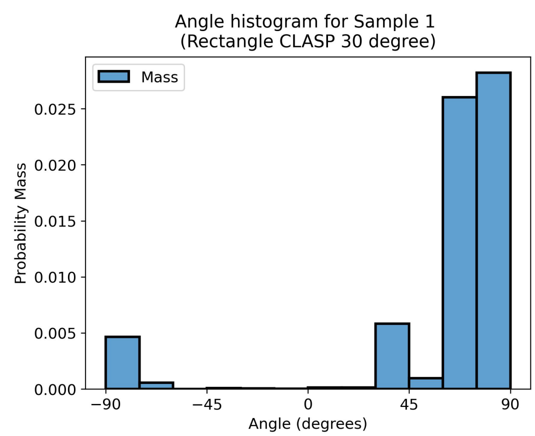


(b) Histogram of array orientation for sample 1 of the experiment rectangular domain with influx of MTs.

Figure B.21: Two samples of histograms used to calculate the mean histogram for the experiment rectangular domain with influx of MTs.

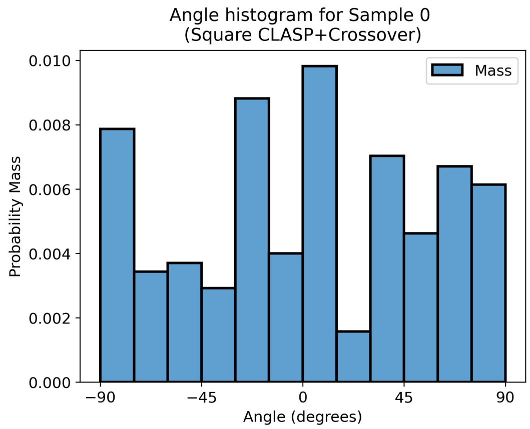


(a) Histogram of array orientation for sample 0 of the experiment rectangular domain with CLASP 30 degrees.

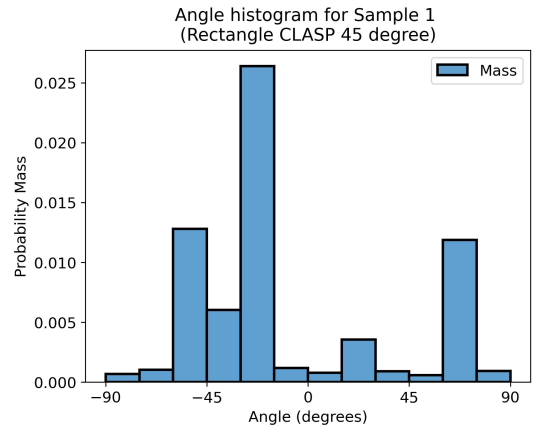


(b) Histogram of array orientation for sample 1 of the experiment rectangular domain with CLASP 30 degrees.

Figure B.22: Two samples of histograms used to calculate the mean histogram for the experiment rectangular domain with CLASP 30 degrees.

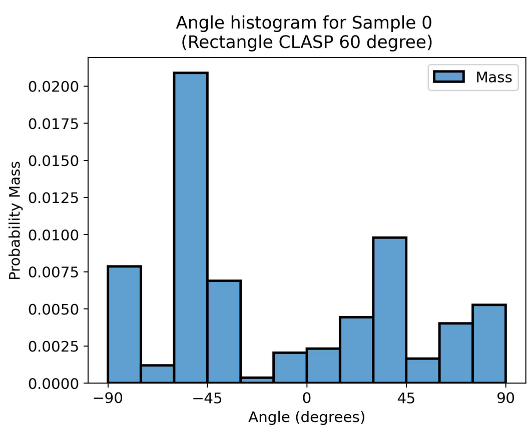


(a) Histogram of array orientation for sample 0 of the experiment rectangular domain with CLASP 45 degrees.

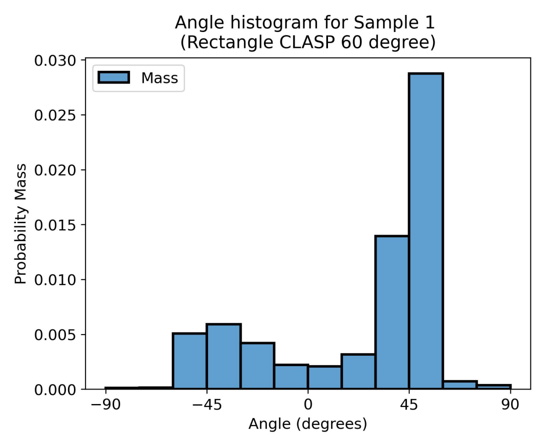


(b) Histogram of array orientation for sample 1 of the experiment rectangular domain with CLASP 45 degrees.

Figure B.23: Two samples of histograms used to calculate the mean histogram for the experiment rectangular domain with CLASP 45 degrees.



(a) Histogram of array orientation for sample 0 of the experiment rectangular domain with CLASP 60 degrees.



(b) Histogram of array orientation for sample 1 of the experiment rectangular domain with CLASP 60 degrees.

Figure B.24: Two samples of histograms used to calculate the mean histogram for the experiment rectangular domain with CLASP 60 degrees.

Appendix C

Approximate Algorithm Derivation

C.1 Approximate Spatially Embedded Hybrid ODE SSA

C.1.1 Overview

In this section we discuss a foundational framework and introduce some of the theory necessary to derive an approximate algorithm. The work in Section C.1 is taken from an unpublished manuscript [78] with theory derived by Eric Mjolsness and transcribed and further extended by myself, with slight modifications added for clarity. The theoretical portion of this work is still in progress. A proof in future work would be required to define an exact error bound of the approximate algorithm. The first subsection includes necessary notation for what follows. The second subsection introduces assumptions. We conclude this section with a strategy for how the algorithm can be justified, the algorithm itself, and a brief discussion of the prospects it has for parallelization.

C.1.2 Notation

- $\{\!\!\{ \cdot \}\!\!\} \equiv$ a multi-set, where the center dot is used as an arbitrary placeholder.

- $i, j \equiv$ object instance number (hidden j could be OID (Object ID) or reordering thereof)

- Let $n \in \mathbb{Z}^{+}$ such that $\mathbb{R}^{n}$ is a high dimensional real space. For our case, we typically set $n = 2$ or $n = 3$

- We further define $D = \{n, n-1, \ldots, 0\}$ such that $d \equiv$ cell dimension and $d \in D$ is the dimension of a cell's level number in an abstract cell complex? For example, in the practical case of $\mathbb{R}^{2}$ we have $D = \{2, 1, 0\}$

- $b \equiv$ bin number $\equiv$ number of $d \equiv d_{\max}$ i.e. the highest dimensional cell

- $c \equiv$ cell number

- $\vec{x} \equiv$ position of the object (centered or other representation)

- $\vec{\alpha} \equiv$ all other parameters associated with the object, discrete or continuous. Includes OIDs for graph grammar implementation(SGGs)

- $\tau \equiv$ type of object (typeterm in Γ) a.k.a. species

- $r \equiv$ reaction/rule number

- $R \equiv$ reaction/rule firing/instantiation/channel $\longleftrightarrow (r, [\vec{x}_{r\,p}, \vec{\alpha}_{r\,p}]_{p}, [\vec{y}_{r\,q}, \vec{\beta}_{r\,q}]_{q})$ - excludes i, j information so these are summed over.

- $\partial \equiv$ boundary relationship (unsigned) $\equiv$ map from a singleton d-cell to a set of $(d-1)$ cells
  - $\Longrightarrow$ map from a set of d-cells to a multi-set of $(d-1)$ cells
  - $\partial^{-1} \equiv$ inverse relationship.
  - signed version has $\partial^{2} = 0$

- $In(r)$, $Out(r) \equiv$ range of p, q in rule r syntax $\equiv$ number of terms on the LHS, RHS
- $Reactants(R) \equiv \{(\vec{x}_{r\,p}, \vec{\alpha}_{r\,p})\big|p \in In(r)\} \cup \{(\vec{y}_{r\,q}, \vec{\beta}_{r\,q})\big|q \in Out(q)\}\}$
  - $\equiv$ set of complete attributes of object (defining them up to renumbering of i, j) of a rule firing.
- $\rho(\cdot) \equiv$ a generic propensity function s.t. $\rho_r(\cdot) \equiv$ propensity for a rule r to fire.
- $C_{r,r'} \equiv$ commutation structure of the grammar such that:

$$C_{r,r'} \equiv \begin{cases} 0, & [W_r, W_{r'}] = 0, \text{ alternatively if } \int \Big|[W_r, W_{r'}]\Big| < \epsilon \cdot O(1) \\ 1, & otherwise \end{cases}$$

- $C_{R,R'} \equiv$ same as above.

We can then denote a Reaction/Rule r as follows:

$$\{\tau^{in}_{r,p}(\vec{x}_{r\,p}, \vec{\alpha}_{r\,p})\}_{p \in In(r)} \longrightarrow \{\tau^{out}_{r,q}(\vec{y}_{r\,q}, \vec{\beta}_{r\,q})\}_{q \in Out(r)}$$
$$\textbf{with} \quad \rho_r([\vec{x}_{r\,p}.\vec{\alpha_{r\,p}}]_p), [\vec{y}_{r\,q}, \vec{\beta}_{r\,q}]_q) = \tilde{\rho}_r([\vec{x}_{r\,p}, \vec{\alpha}_{r\,p}]_p) \times \rho_r\big([\vec{y}_{r\,q}, \vec{\beta}_{r\,q}]\big|[\vec{x}_{r\,p}, \vec{\alpha}_{r\,p}]\big)$$

C.1.3 Assumptions

A1: Spatially Local

Assumption 1. Our (graph) grammar rules are spatially-local by virtue of their propensity functions ρ

High level explanation:

We want to define some neighborhood where we can count the reactions and rule out the ones that don't fit. We let $\epsilon \in [0,1]$ and be a user chosen threshold for the minimum probability a rule must have to be considered possible to fire - i.e. our "fall off" distance.

Mathematical: $\forall\ \epsilon > 0 \quad \exists \quad \delta_r(\epsilon) = \epsilon\text{-diameter}_r > 0$, and $p\,,\,q\ \equiv$ indexes of slots.

$$\begin{aligned} &(\ \forall\ p,p' \quad \|x_{r\,p} - x_{r\,p'}\| < \delta_r \quad \wedge \quad \forall\ p,q' \quad \|x_{r\,p} - y_{r\,q}\| < \delta_r \quad \wedge \quad \forall\ q,q' \quad \|y_{r\,q} - y_{r\,q'}\| < \delta_r) \\ &\quad \vee \quad (\ \rho([\vec{x},\vec{\alpha}],[\vec{y},\vec{\beta}])\ <\ \epsilon\ ) \end{aligned} \tag{C.1}$$

Alternatively,

$$\int_{V=\{\ \|\cdot\|\geq\delta_r\ \}} \prod_p d\vec{x} \prod_q d\vec{y}\ \rho([\vec{x},\vec{\alpha}],[\vec{y},\vec{\beta}]) < \epsilon \cdot O(1)$$

may be needed, if volume $|V|$ is infinite(usually we can bound it). We let $\|\cdot\|$ be the norm of $\vec{x}$, $\vec{y}$. The integral is a less strict case of equation C.1. It ensures that for any volume larger than δ_r, all probabilities/propensities in that volume must be much less than the user chosen ϵ.

Consequently, rule firings R can be ϵ-valid or ϵ-invalid:

$$\begin{aligned} \epsilon\text{-valid}(\ R\ =\ (r,[\vec{x}_{r\,p},\vec{\alpha}_{r\,p}]_p,[\vec{y}_{r\,q},\vec{\beta}_{r\,q}]_q)\ ) &\equiv \rho_r(\ [\vec{x}_{r\,p},\vec{\alpha}_{r\,p}]_p,[\vec{y}_{r\,q},\vec{\beta}_{r\,q}]_q\ ) \geq \epsilon \\ \neg\epsilon\text{-valid} \equiv \epsilon\text{-invalid}(\ R\ =\ (r,[\vec{x}_{r\,p},\vec{\alpha}_{r\,p}]_p,[\vec{y}_{r\,q},\vec{\beta}_{r\,q}]_q)\ ) &\equiv \rho_r(\ [\vec{x}_{r\,p},\vec{\alpha}_{r\,p}]_p,[\vec{y}_{r\,q},\vec{\beta}_{r\,q}]_q\ ) < \epsilon \end{aligned} \tag{C.2}$$

We can filter out some invalid R using the predicate:

$$\epsilon\text{-OK}(R) \quad \equiv \quad \forall\ p,p',q,q' \quad \|\vec{x}_{r\,p}-\vec{x}_{r\,p'}\| < \delta_r \quad \wedge \quad \|\vec{x}_{r\,p}-\vec{y}_{r\,q}\| < \delta_r \quad \wedge \quad \|\vec{y}_{r\,q}-\vec{y}_{r\,q'}\| < \delta_r \tag{C.3}$$

Then we can deduce:

$$\begin{aligned} \neg\epsilon\text{-OK}(R) &\implies \neg\epsilon\text{-valid}(R) \\ \epsilon\text{-valid}(R) &\implies \epsilon\text{-OK}(R) \end{aligned} \tag{C.4}$$

It is sufficient to apply an even weaker predicate (Here we introduce the idea of an anchor point):

$$\epsilon\text{-OK}'(R,p) \quad \equiv \quad \forall\ p', q \quad \|\vec{x}_{r\,p} - \vec{x}_{r\,p'}\| < \delta_r \quad \wedge \quad \|\vec{x}_{r\,p} - \vec{y}_{r\,q}\| < \delta_r \tag{C.5}$$

Then by using the triangle inequality:

$$\begin{aligned} \epsilon\text{-OK}'(R, p \in In(r(R))) \quad &\implies \quad \epsilon^*\text{-OK}(R) \\ \epsilon\text{-OK}(R) \quad &\implies \quad \epsilon\text{-OK}'(R,p) \end{aligned} \tag{C.6}$$

Where $\epsilon^*(\epsilon) = \max_r \delta_r^{-1}(2\delta_r(\epsilon))$ and:

$$\epsilon\text{-valid}(R) \quad \implies \quad \epsilon\text{OK}'(R,p) \quad \forall\ p \in In(r) \tag{C.7}$$

A2: Collars

Assumption 2. Every dimension $d < d_{\max}$ has a $d_{\max}$ dimensional collar

Definition C.1

$$\begin{aligned} &Collar(c \mid \dim(c) = d) = \{\vec{x} \mid \inf_{z \in c} \|x - z\| < \Delta_d\} \setminus \bigcup_{c' \mid \dim(c') < d} Collar(c) \\ &\qquad where \quad \Delta_d > 0 \quad and \quad \Delta_{d' < d} \geq \Delta_d \end{aligned} \tag{C.8}$$

We further assume that **collars do not overlap**:

$$c \neq c' \quad \implies \quad Collar(c) \cap Collar(c') = \varnothing \tag{C.9}$$

This is trivial for $d \neq d'$, but a nontrivial geometric constraint for $d = d'$.

A3: Well Separated

Assumption 3. Collars of the same dimension such that $d < d_{\max}$ are well-separated.

We can represent assumption 3 as follows:

$$c \neq c' \quad \wedge \quad \dim(c) = \dim(c') \quad \implies \quad \inf_{\vec{x}\in c\,,\,\vec{z}\in c'} \|\vec{x} - \vec{z}\| \geq \Delta_d \tag{C.10}$$

From this we can conclude that cells of a lower dimension are subject to the constrain:

$$\Delta_{d=0} \geq c_{0,1}\,\Delta_{d=1} \tag{C.11}$$

With $c_{0,1}$ from a geometric calculation.

Note: collars could be even bigger as long as the inequalities are satisfied. E.G. 0-dim collars could be $\|x - z\|_\infty < \Delta_d \implies$ square enclosing a circle.

A4: Bins Are Large Enough

Assumption 4. Bins (= max-dim cells) are large enough.

We define a bin as the max-dimensional cell. We can represent assumption 4 as:

$$\begin{aligned}
&\forall\, \vec{x} \in b \quad \forall\, \vec{y}\; [||\vec{x} - \vec{y}|| < \Delta_{d_{\max}}(b) \quad \Longrightarrow \quad b(\vec{y}) = b \quad \vee \quad b(\vec{y}) \in \mathrm{nbrs}_{d_{\max}}(b)] \\
&\qquad \text{for some } \Delta_{d_{\max}}(b) > 0 \;\sim\; \min(\mathrm{diameter}(b), \mathrm{diameter}(\mathrm{nbrs}(b)))
\end{aligned} \tag{C.12}$$

In equation C.12 in the mapping $\Delta_{d_{\max}}(b)$ we can vary the output i.e. the size with the bin number, for future adaptive/multi-scale partitioning.

Here the mapping $\mathrm{nbrs}(c)$, which represents a function for determining the neighbors for some cell c could be:

$$\begin{aligned}
&\mathrm{nbrs}(c) = \partial^{-1} \circ \partial(\{c\}) \quad \text{//main neighbors only} \\
&\qquad \textbf{or} \\
&\mathrm{nbrs}(c) = \left[\quad \bigcup_{d < d_{\max}} \quad \partial^{-1\,(d_{\max}-d)} \partial^{(d_{\max}-d)}(\{c\}) \quad \right] \quad \text{//includes all corner neighbors}
\end{aligned} \tag{C.13}$$

A5: Cells Are Large Enough

Assumption 5. Constraints on $\Delta_{d_{\max}}$ is large enough.

$$\begin{aligned}
&\delta_r(\epsilon) \le \Delta_{d_{\max}}(b)\ \forall\ b \\
&\text{i.e.} \quad \forall\ b(\Delta_{d_{\max}}(b) \ge \delta_r(\epsilon))
\end{aligned} \tag{C.14}$$

A6: Constraint on $dim(c) < d_{\max}$

Assumption 6. We have a constraints on lower dimensional cells.

$$d < d_{\max} \quad \Longrightarrow \quad \Delta_d \quad \ge \quad \gamma_d \max_r \delta_r(\epsilon), \quad \gamma_d \ge 1 \tag{C.15}$$

A7: Minimum Scaling

Assumption 7. We set a minimum scaling factor.

$$\forall\ d \in \{0, \ldots, d_{max-1}\} \quad \gamma_d \geq 4 \tag{C.16}$$

C.1.4 Strategy

We use this setting and the assumptions to decompose operator W into dimension-d slices each of which can be parallellized over well-separated cells of dim d:

$$\begin{aligned}
W_{tot} &= \sum_r W_r & &\equiv \sum_r \int_{R|r} W(R) \\
&= \sum_d \sum_r W_r^{(d)} & &\equiv \sum_d (W_{(d)} = \sum_r W_r^{(d)}) \\
&= \sum_d \sum_{c|d} \sum_r W_{r,collar(C)} & &\equiv \sum_d \sum_{c|d} W_{(c,d)} \\
W_{tot} &= \sum_d \sum_{c|d} \sum_r \int (\ R\ \times\ |\varphi(R)| = c\ ) dV\ W_r(R|c,d)
\end{aligned} \tag{C.17}$$

where:

$$\begin{aligned}
&\forall\ c, c' \quad | \quad dim(c) = dim(c') = d \\
&\forall\ r, r' \quad [W_{r,collar(c)}, W_{r',collar(c')}] = 0
\end{aligned} \tag{C.18}$$

Here we say that if cells are of the same dimension, then the commutator should be zero. So, due to well separatedness of collars and bins (implicit for ∂ widths $\Delta_{d'} \geq \Delta_{d_{\max}-1}$) on the scale of reactions R ($\sim \delta$) as established by the ϵ-OK$'$ filter.

Then, each $\sum_{c|d}$ can be parallelized! By which we mean:

$$
\begin{aligned}
e^{tW} &\approx \left(\prod_{d\downarrow} e^{\frac{t}{n} W_{(d)}} \right)^{n\to\infty} && \text{//1st order operator splitting - go 2nd order instead} \\
e^{t' W_d} &= \prod_{c\text{ (any order)}} e^{t' W_{(c,d)}} && \text{where} \quad [W_{(c,d)}, W_{(c',d)}] = 0 \quad \text{and} \quad t' \equiv \frac{t}{n} \\
W_{(c,d)} &= \sum_r W_{r,\text{collar}(c)} && \equiv \sum_r \sum_{R\,|\,\varphi(R)=c} W_r(\,R \mid c, d)
\end{aligned}
\tag{C.19}
$$

Given the current strategy, we need to find $\varphi(R) \to c$. So, we must find the function that maps a rule firing R to a cell c.

The strategy is then as follows:

(a) Show, based on proximity, there is only one candidate of each dimensionality of $(0 \leq d \leq d_{\max})$.

(b) Choose the highest dimension compatible with separation into parallel executable rule firing sets. This will also be the lowest dimension containing certain R, R' interactions to be defined.

(c) Key sepeartion criterion for $\varphi(R)$:

$$
\begin{aligned}
&\textbf{IF} \quad d(\varphi(R_1)) = d(\varphi(R_2)), \quad \textbf{THEN} \\
&\quad (\text{Reactants}(R_1) \cap \text{Reactants}(R_2) = \varnothing) \quad \vee \quad (\varphi(R_1) = \varphi(R_2)) \\
&\textbf{i.e.} \quad (\text{Reactants}(R_1) \cap \text{Reactants}(R_2) \neq 0) \quad \Longrightarrow \quad (\varphi(R_1) = \varphi(R_2)) \\
&\textbf{i.e.} \quad (\varphi(R_1) \neq \varphi(R_2)) \quad \Longrightarrow \quad (\text{Reactants}(R_1) \cap \text{Reactants}(R_2) = \varnothing)
\end{aligned}
\tag{C.20}
$$

(d) So if the previous is established:

$$\begin{aligned}&(c_1 \neq c_2) \quad \Longrightarrow \quad [W_{(c_1,d)}(R_1|c,d), W_{(c_2,d)}(R_2|c_2,d)] = 0 \\ &\qquad \text{where} \quad c_1 = \varphi(R_1) \quad \wedge \quad c_2 = \varphi(R_2)\end{aligned} \tag{C.21}$$

//Note: why: $c_1 \neq c_2$: see point (c) on line 4? Need to refine this

(e) For ϵ-valid firing R define(Key step in defining $\varphi(R)$):

$$\text{Binset}(R) = b(\text{Reactants}(R)) \quad \cup \bigcup_{R' \epsilon \text{Overlapset}(R)} b(\text{Reactants}(R)) \tag{C.22}$$

$$\text{Overlapset}(R) \equiv \{\ R' \mid \epsilon\text{-valid}(R') \wedge \text{Reactants}(R)\ \cap\ Reactants(R') \neq \varnothing\} \tag{C.23}$$

Actually, we will replace ϵ-valid with ϵ-OK$'(R, p(r))$ obtaining $\widehat{\text{Binset}}$ and $\widehat{\text{Overlapset}}$ once $p(r)$ has been defined. Define the "home bin" of rule firing R as:

$$b_R \ \equiv\ b(\vec{X}_{r(R),p(R)}) \quad \text{where} \quad p(R) \in In(r)\ \cup\ Out(r) \tag{C.24}$$

It's preferred to use $In(r)$ unless $\varnothing$ i.e. we prefer to use the incoming objects to find the center. It is the geometrically central "on average". Thus, $p(R)$ is the "home object slot" of rule r (also a definition). We can use $p(R) \in \vec{x}_{r(R),p(R)}$ in the triangle inequality to bound the distances between other objects in R.

Now, define predicate as seen in equation C.5:

$$\begin{aligned}&\epsilon\text{-OK}'(R) \equiv \epsilon\text{-OK}'(R,p) \\ &\qquad = \forall\ p', q \quad \|\vec{x}_{r(R),p(R)} - \vec{x}_{r\,p'}\| < \delta_r(\epsilon) \quad \wedge \quad \|\vec{x}_{r(R),p(R)} - \vec{y}_{r\,q}\| < \delta_r(\epsilon)\end{aligned} \tag{C.25}$$

Combining equations C.25, C.12, C.7 as follows:

$$\epsilon\text{-valid}(R) \quad \overset{(C.7)}{\Longrightarrow} \quad \epsilon\text{-OK}'(R)$$

Using assumption 5: $\delta_r(\epsilon) \leq \Delta_{d_{\max}}(b_R)$

Then,

$$\begin{aligned}
\epsilon\text{-OK}'(R) \implies \forall\ p', q \qquad & \|\vec{x}_{r(R),p(R)} - \vec{x}_{r\,p'}\| < \delta_r(\epsilon) \quad \wedge \quad \|\underbrace{\vec{x}_{r(R),p(R)}}_{\in b_R} - \vec{y}_{r\,q}\| < \delta_r(\epsilon) \\
\underset{(C.12)}{\Longrightarrow} \forall\ p', q \qquad & (\ b(\vec{x}_{r(R)p'}) = b_R \ \vee\ b(\vec{x}_{r(R)p'}) \in \text{nbrs}_{d_{\max}}(b)\ ) \\
& \wedge \quad (\ b(\vec{y}_{r(R)q}) = b_R \ \vee\ b(\vec{y}_{r(R)q}) \in \text{nbrs}_{d_{\max}}(b)\ )
\end{aligned}$$

So,

$$\epsilon\text{-valid}(R) \implies \epsilon\text{-OK}'(R) \implies b(\text{ Reactants}(R)\ ) \subseteq \{b_R\} \cup \{\text{nbrs}_{d_{\max}}(b_R)\} \tag{C.26}$$

Now, let R be $\epsilon\text{-OK}'(R) = \textbf{true}$, and define $\widehat{Binset}$ and $\widehat{\text{Overlapset}}$ as promised:

$$\begin{aligned}
\widehat{Binset} &= b(\text{ Reactants}(R)\ ) \cup \bigcup_{R' \in \text{Overlapset}(R)} b(\text{ Reactants}(R')\ ) \\
\widehat{\text{Overlapset}} &= \{\ R' \mid \epsilon\text{-OK}'(R') \ \wedge\ \text{Reactants}(R) \cap \text{Reactants}(R')\} \neq \varnothing
\end{aligned} \tag{C.27}$$

Since $\epsilon\text{-OK}'(R) \implies \epsilon^*\text{-OK}(R)$ ($\epsilon^*(\epsilon) = \max_r \delta_r^{-1}(2\delta_r(\epsilon))$), all $Reactancts(R)$ are clustered to within $2\delta_r(\epsilon)$ of one another:

$$
\begin{aligned}
&\|\vec{x}_{r(R)\,p} - \vec{x}_{r(R)\,p'}\| < 2\delta_r(\epsilon)\\
&\|\vec{x}_{r(R)\,p} - \vec{y}_{r(R)\,q'}\| < 2\delta_r(\epsilon)\\
&\|\vec{y}_{r(R)\,q} - \vec{y}_{r(R)\,q'}\| < 2\delta_r(\epsilon)
\end{aligned}
\tag{C.28}
$$

Then, there is an object at position $\vec{x}_{r(R)\tilde{p}} = \vec{x}_{r(R')\tilde{p}'}$ or $\vec{y}_{r(R)\tilde{q}} = \vec{y}_{r(R')\tilde{q}'}$ in the nonempty intersection Reactants(R) $\cap$ Reactants(R), and:

$$
\begin{aligned}
\forall\, p, q \in In(r(R)) \cup Out(r(R)), \quad \wedge \quad \forall\, p', q' \in In(r(R')) \cup Out(r(R'))&\\
\|\underbrace{\vec{x}_{r(R)p}}_{\text{or y or q}} - \vec{x}_{r(R')p'}\| \le \|\vec{x}_{r(R)\,p} - \vec{x}_{r(R)\,p'}\| + \|\vec{x}_{r(R)\tilde{p}} - \vec{x}_{r(R')\tilde{p}'}\| = 0&\\
+\|\vec{x}_{r(R')\tilde{p}'} - \vec{x}_{r(R')p'}\| < 2(\delta_{r(R)}(\epsilon) + \delta_{r(R')}(\epsilon))&
\end{aligned}
\tag{C.29}
$$

Either way:

$$
\begin{aligned}
&\text{all reactants of overlapping } \epsilon\text{OK}' \text{ rule firings } R_1 \textit{ and } R_2 \text{ lie within a bounded distance}\\
&\qquad 2\delta_{r(R_1)}(\epsilon) \textit{ or } 2(\delta_{r(R_1)}(\epsilon) + \delta_{r(R_2)}(\epsilon)) \text{ of one another.}
\end{aligned}
\tag{C.30}
$$

We seek to use this "clustering" of all reactants in any two overlapping ϵ-OK$'$ rule firings (say R and R') to ensure that all such reactants fall within at most one collar of each dimension $d < d_{\max}$.

To this end assume:

$$
\Delta_{d<d_{\max}} > \gamma_d \max_r \delta_r(\epsilon), \quad \text{such that} \quad \gamma_d \ge 1 \tag{C.31}
$$

We can see that $\gamma_d \geq 1$ because $\Delta_{d<d_{\max}} \geq \Delta_{d_{\max}} \geq 1$. In view of the previous equations, take $\gamma_{d<d_{\max}} \geq 4$. Then any two reactants in R and R' must have distance $\geq \Delta_{d=1}$ and hence by assumption 3 cannot be contained in tow differnt collars at the same dimension $d < d_{\max}$.

For $d = d_{\max}$ we could define the "collar" of a bin b = cell c to be an entire $d_{\max}$-cell = bin b $\setminus \bigcup_{\text{all lower dim collars}}$. Similar to an earlier equation. And, take $\Delta_{d_{\max}} = \Delta_{d_{\max}-1}$ (in which case that earlier equation may literally hold). Then, $\gamma_d \geq 4 \implies$ only two reactants in R and R' must have distance $< \Delta_{d_{\max}}$. If they were in different $d = d_{\max}$ bin interiors, then they must be separated by some collar whose minimum traversed distance is $2\Delta_d$ which is not $< \Delta_d$ - so, they must be in at most a single bin interior.

Definition C.2

$\forall\ d < d_{\max}$, $\gamma_d \geq 4$ the previous equation $\implies$ $\forall$ ϵ-OK' R, R' All reactants in R and/or its overlapping R' fall within at most one collar (including bin interiors for $d = d_{\max}$) of each dimension d, $0 \leq d \leq d_{\max}$.

Definition C.3

$\varphi(R) \equiv$ the (unique) cell of lowest dimension d among those whose collar (or bin "interior", for $d > d_{\max}$) contains any of the reactants in Participants(R).

Where:

$$Participants(R) \equiv Reactants(R) \cup \bigcup_{R' \in \overbrace{Overlap}set(R)} Reactants(R') \tag{C.32}$$

Assuring A3 will require $\Delta_0 \geq \Delta_1 \geq \Delta_2 \ldots$ by a constant factor in each case:

$$\underbrace{\Delta_d}_{>\gamma_d\ max_r\ \delta_r} \geq c_d \underbrace{\Delta_{d+1}}_{>\gamma_{d+1}\ max_r\ \delta_r} \geq \left(\prod_d c_d\right) \underbrace{\Delta_{d_{\max}-1}}_{>\gamma_{d_{\max}}\ max_r\ \delta_r(\epsilon)}$$

where $c_d > 1$ is chosen by geometry $\implies$ a pure number(no units). So, it suggests:

$$\forall\ d \in \{0, \ldots, d_{\max} - 2\} \quad \gamma_d \geq c_d \gamma_{d+1},\ \text{with } c_d > 1, \gamma_{d_{\max}-1} \geq 4 \text{ suffices} \tag{C.33}$$