The Local Rules Dynamics Model for Self-Assembly Simulation

by

Russell S. Schwartz

S.B., Massachusetts Institute of Technology (1996)
M.Eng., Massachusetts Institute of Technology (1996)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2000

Author . . . .
Department of Electrical Engineering and Computer Science
January 7, 2000

Certified by . . . .
Bonnie Berger
Samuel A. Goldblith Associate Professor of Applied Mathematics
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The Local Rules Dynamics Model for Self-Assembly Simulation

by

Russell S. Schwartz

Submitted to the Department of Electrical Engineering and Computer Science
on January 7, 2000, in partial fulfillment of the
requirements for the degree of Doctor of Philosophy

Abstract

Biochemical self-assembly is an important but still poorly understood phenomenon. Self-assembly occurs in many natural systems and may be useful in designing novel materials and nanometer-scale machines. A particularly important self-assembly system about which many important questions remain unanswered is the viral protein shell. Answering these questions may be crucial in understanding the viral life cycle and developing treatments for viral illnesses.

The local rules theory of virus shell assembly [8] has proved useful in gaining insight into virus shell assembly and other self-assembly problems. The theory by itself, however, does not apply to certain key problems in areas such as binding kinetics and assembly pathways. A simulation model was constructed to expand upon the local rules framework and produce qualitative simulations capable of extending the range of phenomena that can be examined. This model has since been applied to a variety of problems in virus shell assembly and related fields.

This thesis presents the local rules dynamics simulation model, which combines the local rules abstraction with a molecular dynamics-like framework to allow computationally tractable simulations of self-assembly processes. The thesis additionally covers several applications of the model, including exploration of the effects of parameter variations on virus shell assembly, the merits of different models for the source of nucleation-limited assembly kinetics, and strategies for disrupting assembly. Other applications demonstrate how local rules dynamics can apply to related systems, such as glasses, protein aggregates, and hypothetical artificial self-assembling machines.

The thesis also describes a particular implementation of the local rules dynamics model designed for virus shell assembly simulations. It first covers a detailed physical model based on the local rules dynamics idea. It then describes two numerical methods, one based on Richardson extrapolation [90] and the other on waveform relaxation [64], some key discrete algorithms, and the simulator's user interface.

Thesis Supervisor: Bonnie Berger
Title: Samuel A. Goldblith Associate Professor of Applied Mathematics

Acknowledgments

I would first like to thank my thesis supervisor, Bonnie Berger. Over the years, she has given me a great deal of freedom to pursue this work but has also kept me from making too many mistakes along the way. Her advice and guidance on this project and on my other work throughout the time I have worked for her have been invaluable.

I would also like to thank the other members of my thesis committee both for their assistance in their capacity as thesis readers and for their other help. Jacob White has been a great help in suggesting the use of a waveform-relaxation method, in providing guidance on efficient use of the numerical methods, and in suggesting improvements on the presentation of this work. Charles Leiserson has been similarly helpful over the years with his suggestions about how to use Cilk multithreading most effectively on this project. I am also grateful for his extensive comments on the preparation of this thesis document.

I also thank my collaborators from the biology community for all of their help. Peter Prevelige has been consistently supportive of this work and has provided extensive feedback and discussions regarding how to improve upon this work and keep it relevant to the biology. Jonathan King has provided valuable advice on which avenues to pursue in advancing this work and on how to present it to the biology community. Bob Garcea has also been a great help in discussions on extending this work to other systems and presenting it to biologists.

I would like to thank the many other colleagues from the Theory of Computation group and the Cilk group at the MIT Lab for Computer Science who have shared their expertise on various aspects of this project at one time or another. Doug Muir, Esther Jesurum, Vinay Pulim, and Alex Coventry have been especially helpful to me over the years that I have been working on this project in suggesting improvements to this work and providing guidance on their own areas of expertise.

I am also grateful to the other colleagues who have supported and guided my work as a graduate student. Sorin Istrail has been a great source of advice and support on this project and my research as a whole. Peter Shor also provided many helpful suggestions regarding the pursuit of this project. Linn Hobbs has provided much advice and feedback on the applications portion of the project.

I also want to thank those organizations that provided financial support for this work. Aspects of this work were supported under National Science Foundation grant DBI9711234 and Department of Energy grant DEFG0295ER25253.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Finally, I would like to thank my family for their consistent support throughout my education.

Contents

1 Introduction **11**

1.1 Background 14

1.2 Motivation for the Present Work 23

1.3 Summary of Major Revisions 25

1.4 Contributions of this Thesis 27

2 The Local Rules Dynamics Model **29**

2.1 Other Methods for Self-Assembly Simulation 30

2.2 The Value of Local Rules Dynamics 37

2.3 Implementing and Applying the Local Rules Dynamics Model 41

3 Applications **42**

3.1 Shell Geometries 43

3.2 Parameter Variations 45

3.3 Comparing Assembly Models 53

3.4 Disrupting Growth 68

3.5 Non-Virus Systems 72

3.5.1 Actin 72

3.5.2 Borosilicate Glasses 77

3.5.3 Protein Aggregation 81

3.5.4 Novel Self-Assembly Systems 85

4 Physical Model **91**

4.1 Protein Model 92
4.2 Binding Interactions 94
4.3 Collision Interactions 99
4.4 Thermodynamic Constraints 100

5 Numerical Methods 108
5.1 Extrapolation-Based Method 110
5.2 Waveform Relaxation-Based Method 116
5.3 Comparing the Methods 123

6 Algorithms 135
6.1 Formation of Binding Interactions 135
6.2 Collision Detection 140

7 User Interface 151
7.1 Command Interpreter 152
7.2 Graphical Interface 157
7.3 Network Interface 162
7.4 Evaluation 163

8 Conclusions and Discussion 166
8.1 Summary 166
8.2 Open Problems and Future Work 167
8.3 Conclusions 169

A Control Language Reference 186
A.1 Special Forms 188
A.2 Type Manipulation 189
A.3 Graphics 191
A.4 Communication 193
A.5 Arithmetic and Logic 195
A.6 Interacting with the Operating System 198

A.7 Pre-Defined Non-Primitive Procedures 199

B Network Interface Protocols **203**

List of Figures

1-1 T=7 Local Rules 15

1-2 Simulated T=7 Shell 16

1-3 Simulated Spiral Structure 17

1-4 Non-Quasi-Equivalent Shells 18

2-1 Hierarchy of Simulation Methods 36

2-2 Adding Local Rules Dynamics to the Hierarchy 39

3-1 Simulated Virus Shells 45

3-2 Growth as a Function of Binding Energy 48

3-3 Growth as a Function of Binding Tolerance 50

3-4 Growth as a Function of Concentration 51

3-5 Malformed Clusters 53

3-6 Simulation of Model One 59

3-7 Simulation of Model Two 60

3-8 Simulation of Model Three 60

3-9 Simulation of Model Four 61

3-10 Comparison of Growth for Three Nucleation Models 66

3-11 Growth in the Presence of "Poisoned Subunits" 70

3-12 Growth in the Absence and Presence of Poisoned Subunits 71

3-13 Actin Simulation 76

3-14 Actin Dynamics 76

3-15 Cooling Schedule Code 79

3-16 Borosilicate Glass 80

3-17 Three-Helix Bundle 84

3-18 Simulated Protein Aggregates 84

3-19 Aggregation Simulations 86

3-20 Self-Assembling Circuits 88

3-21 An Error in Circuit Assembly 90

4-1 Node Types 93

5-1 Pseudo-code for the Waveform-Relaxation Method 122

5-2 Performance Tests 124

5-3 Performance Test One 126

5-4 Performance Test Two 127

5-5 Performance Test Three 129

5-6 Performance Test Four 130

5-7 Performance Test Five 132

6-1 Bounding Volume of Spheres Whose Edges End in a Given Bucket . . 138

6-2 Pseudo-code for Assigning Particles to Time-Stamped Buckets 147

7-1 Sample Control Language Code 152

7-2 Rules File 155

7-3 Save File 156

7-4 The Graphical User Interface 158

List of Tables

3.1 Original Nucleation and Growth Rates 58

3.2 Revised Nucleation and Growth Rates 63

Chapter 1

Introduction

Macromolecular self-assembly is an increasingly important area of study, yet it remains poorly understood. One of the most significant examples of self-assembly, in terms of both its sophistication and its practical importance, is the formation of viral protein shells. Forming a viral protein shell requires the spontaneous self-assembly of typically several hundred coat proteins into a complicated but regular structure. While some viral coat proteins require interaction with other proteins or the viral genome for assembly, others are capable of assembling complete shells independently. Although experimental studies of virus assembly are progressing rapidly, important questions about protein shell self-assembly remain beyond their reach.

Computer simulations provide a potential means of broadening the range of questions that can be explored regarding how these shells assemble. I believe that the insights that simulations can provide will lead to a greater practical understanding of the process and to a deeper theoretical grasp of the problems and constraints of molecular self-assembly. Such an understanding will in turn likely yield practical advances in the treatment of viral diseases and applications to other self-assembly systems, both natural and artificial.

A better understanding of viral behavior will yield great practical value. Viruses are a major cause of illness for which there is often no adequate treatment. Viral diseases affecting humans include the common cold, influenza, herpes, some kinds of hepatitis, polio, and AIDS. Viral diseases also affect other animals and plants, making the diseases a problem for agriculture. Developing treatments for viral illnesses will require a better understanding of how viruses actually assemble and function and how their behavior can be manipulated. Furthermore, viruses have the potential to teach us about other problems. The laboratory study of bacteriophages, viruses that infect bacteria, has already led to crucial insights in a variety of other fields. A deeper understanding of their behavior will continue to yield benefits to the study of other areas of biology and chemistry. Furthermore, virus self-assembly has lessons to teach human engineers practicing in materials design, bio-engineering, and the emerging field of nanotechnology. Viruses are nanometer-scale self-assembly systems far more complex, robust, and versatile than anything human engineers can currently produce. Understanding how they accomplish what they do will provide crucial insights into how to advance our own technology.

Unfortunately, understanding virus assembly has proven to be extremely difficult. The quasi-equivalence theory of Caspar and Klug [14], which examines the global symmetries of protein coats, added considerably to understanding of the nature of shell geometries. Quasi-equivalence provided a rationale for many observed structures. In addition, it established a taxonomy of icosahedral shell geometries in terms of triangulation, or "T," numbers. Quasi-equivalence does not provide much insight into the actual process of assembly, however. The small scale and high speed of virus shell assembly has confounded analysis through current experimental approaches. While great advances have been made, fundamental questions remain unanswered. Such questions include the role and timing of conformational shifting (in which proteins alter their shapes and binding configurations), the constraints on assembly pathways, and the roles of auxiliary "scaffolding" proteins in shell assembly. It is unclear when or how experimental approaches may be able to provide definitive answers to these and other unsolved problems.

Modeling and simulation can provide an important complement to experimental and purely theoretical approaches. Simulations depend on experiment as a source of raw data and on theory as a source of hypotheses, which form the bases for the abstractions that make a simulation work. Simulation, however, provides a source of information that neither theory nor experiment can produce in its absence. Simulations provide a means of confirming theoretical hypotheses and experimental observations; a simulation can show that the hypotheses alone adequately account for observed behaviors or they can suggest unanticipated problems that come up when the theory is applied, in turn suggesting a need to question or supplement existing theoretical models. The ability to explore the behavior of a model is particularly important for highly complex systems, such as virus shell assembly, in which the effects of the interactions of different components may not be obvious from theoretical models alone. Simulations thus provide a means of validating what is known about the systems that are being simulated and illuminating what is not known. These insights can guide further avenues of experimental and theoretical work, providing a valuable feedback mechanism in which both the traditional avenues of exploration and the newer simulation models augment one another.

Furthermore, simulation work on this kind of complex biological system will be useful to computer science as well as biology. Efficiently simulating a macromolecular assembly system requires developing algorithms and numerical methods for fast particle simulations that are likely to have broader applicability. The high cost of such simulations makes them a good candidate for parallelization, meaning that work on them can contribute to parallel algorithm development as well. Perhaps more significant than the specific techniques required, though, is the value of the problem as a "real world" test case for the fields of algorithms, numerical methods, and parallel processing. Although each of these fields is relevant to the problem, the fact that the problem itself is motivated by a complicated and irregular physical system means that it is not an ideal showcase for any of them. The algorithms required will tend to lack the elegance of those developed for the abstract theoretical problems generally explored in algorithm design. The numerical methods will have to handle discontinuities and other irregularities that are generally avoided in formal treatments of numerical techniques. The parallel methods will have to deal with problems that are far from the "embarrassingly parallel" examples frequently used to test parallel systems. Precisely because the problem is so far from ideal, it is a valuable test case for how broadly all of these techniques can be applied and how they will need to be adapted to extend their applicability.

This thesis explores an application of computer simulation to the study of self-assembly, particularly of viral protein shells. The focus is both on the development of a simulation tool for this task and on applications of that tool. The project is a continuation of earlier work [8, 77, 94], and the thesis therefore begins by summarizing that work and its relevance to the present work. It next discusses the basis of the simulator's design: the local rules dynamics simulation model. It then describes applications; these are primarily problems in virus shell assembly, but also include applications of the simulation tool to other self-assembly systems. Next, it discusses the functionality of the simulator. It first describes the physical model the simulator implements. The thesis then details the construction of the simulator, describing the major numerical methods that implement the physical model, some key discrete algorithms used in its implementation, and the design of its user interface. Finally, the thesis concludes with a discussion of the simulation model, its effectiveness, and avenues for future work.

1.1 Background

This section presents some background on local rules-based approaches to modeling virus shell assembly. In the first subsection, it explains local rules and describes several applications of early static local rules models. In the second subsection, it discusses prior work on kinetic local rules simulations, the focus of my Master of Engineering thesis [94]. This discussion covers the design goals of the simulator, the design chosen to satisfy those goals, and an evaluation of the design conducted after its original implementation, which concluded that the simulator had achieved most of its design goals, but had significant room for improvement.

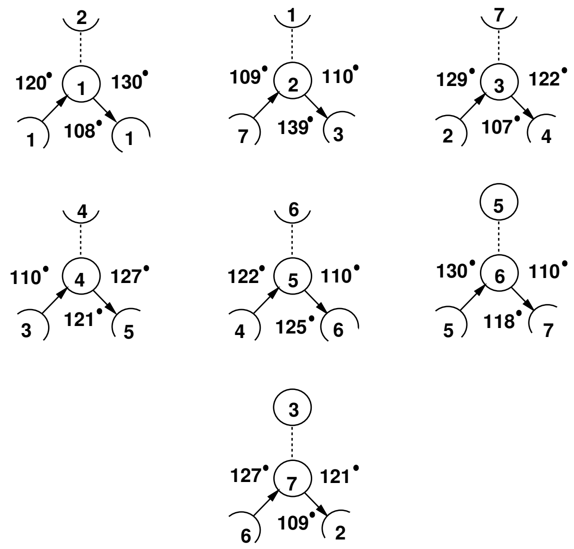

Figure 1-1: Local rules for a T=7 shell geometry

Local Rules

The local rules theory of Berger *et al.* [8] proposes that virus protein shell assembly is directed by the interactions of coat proteins with their immediate neighbors in the shell. These coat proteins are proposed to take on distinct shapes, or conformations, each of which conforms to a "local rule" specifying the conformations of its neighbors and their relative positions in the shell. Figure 1-1, for example, provides rules specifying what is called a T=7 shell in terms of seven coat protein conformations. A computer simulator developed by Berger and Muir [77] makes it possible to construct simulated shells in accordance with specific rule sets. Berger and Muir's simulator provides a method to test and refine rules sets and explore their tolerances to perturbations. Figure 1-2 shows a shell constructed by Berger and Muir's simulator using the rules of figure 1-1.

Local rules models have helped to explain several previously puzzling observations. Local rules have provided a possible explanation for malformed shells that appear as spirals made up of multiple layers of coat proteins. These spirals are observed during normal shell growth and have been found to be the predominant form for the bacter-

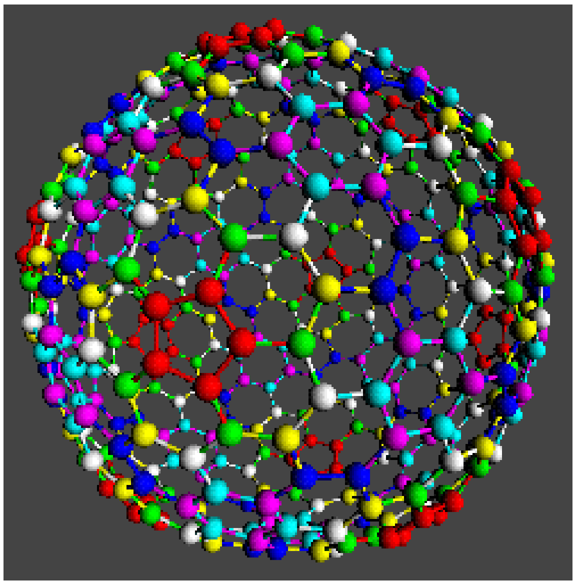

Figure 1-2: Simulated T=7 shell constructed from the rules of figure 1-1.

iophage P22 when an auxiliary scaffolding protein is missing [31, 110]. It was found in local rules simulations that a simple local error in assembly — placing a hexagonal structure in place of a pentagonal structure at the beginning of assembly — could lead to similar spiral structures [8], such as that shown in figure 1-3. Local rules have also helped to explain some shells that do not fit into the binding patterns predicted by the quasi-equivalence theory. For example, local rules provided a simple model for the "T=6" papovavirus geometry, which uses a T=7 lattice but contains only pentagons, rather than the pentagons and hexagons seen in a normal T=7 shell [88, 4]. Local rules also predicted the "T=2" geometry [6], since observed in the *Reoviridae* family of viruses [36, 42]. In addition, local rules models have been constructed for hypothetical "T=5" and "T=8" shells, which appear theoretically possible but are not known to exist. These four geometries are illustrated in figure 1-4.

The basic local rules model was also extended to allow several other applications. The model was given a limited means of favoring the formation of some binding interactions over others in order to allow examination of pathways of assembly. Particular conformations were allowed to bind to multiple different neighboring conformations from a given binding site in order to support "under-determined" local rules sets in

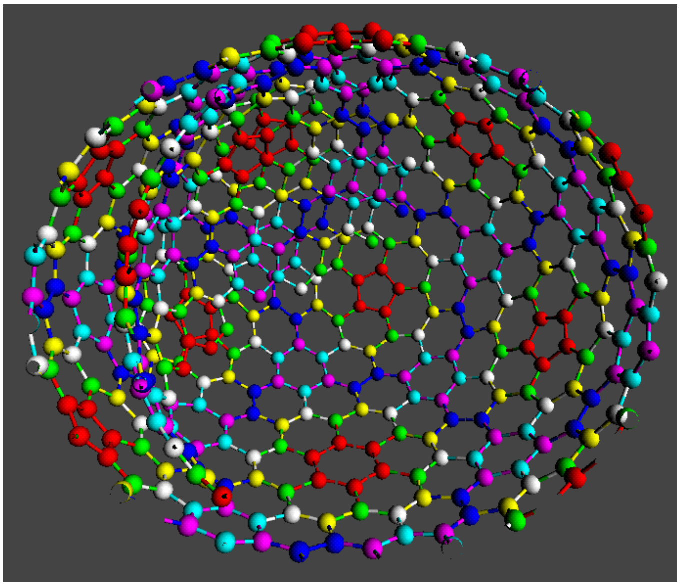

Figure 1-3: Spiral structure created by substituting a hexagon for a pentagon at the beginning of assembly. The image shows a cut-away view in order to make the structure clearer.

which some binding interactions can link to one of several neighbor types. In addition, an extension allowed nodes to alter their conformations during assembly in response to detection of particular neighbor conformations; this extension was meant to model a hypothesis that proteins entering a growing shell initially assume undetermined conformations until they have sufficient neighbors to uniquely specify their conformations. Such extensions allow for a limited examination of problems regarding assembly pathways and conformational switching that could not be fully captured by the earliest conception of local rules.

Static local rules are still being applied to a variety of problems. Local rules models have provided possible explanations for a link between the T=4 and T=7 geometries that has been observed in several virus assembly systems. For instance, a mutant of the T=7 bacteriophage lambda coat protein produces T=4 shells [56]. In addition, the T=7 bacteriophage P2 is sometimes found in conjunction with a "satellite" phage called P4, which uses P2's coat proteins to build its own T=4 shell [70]. It had previously been found that a P4-specific scaffolding protein, gpSid, controls

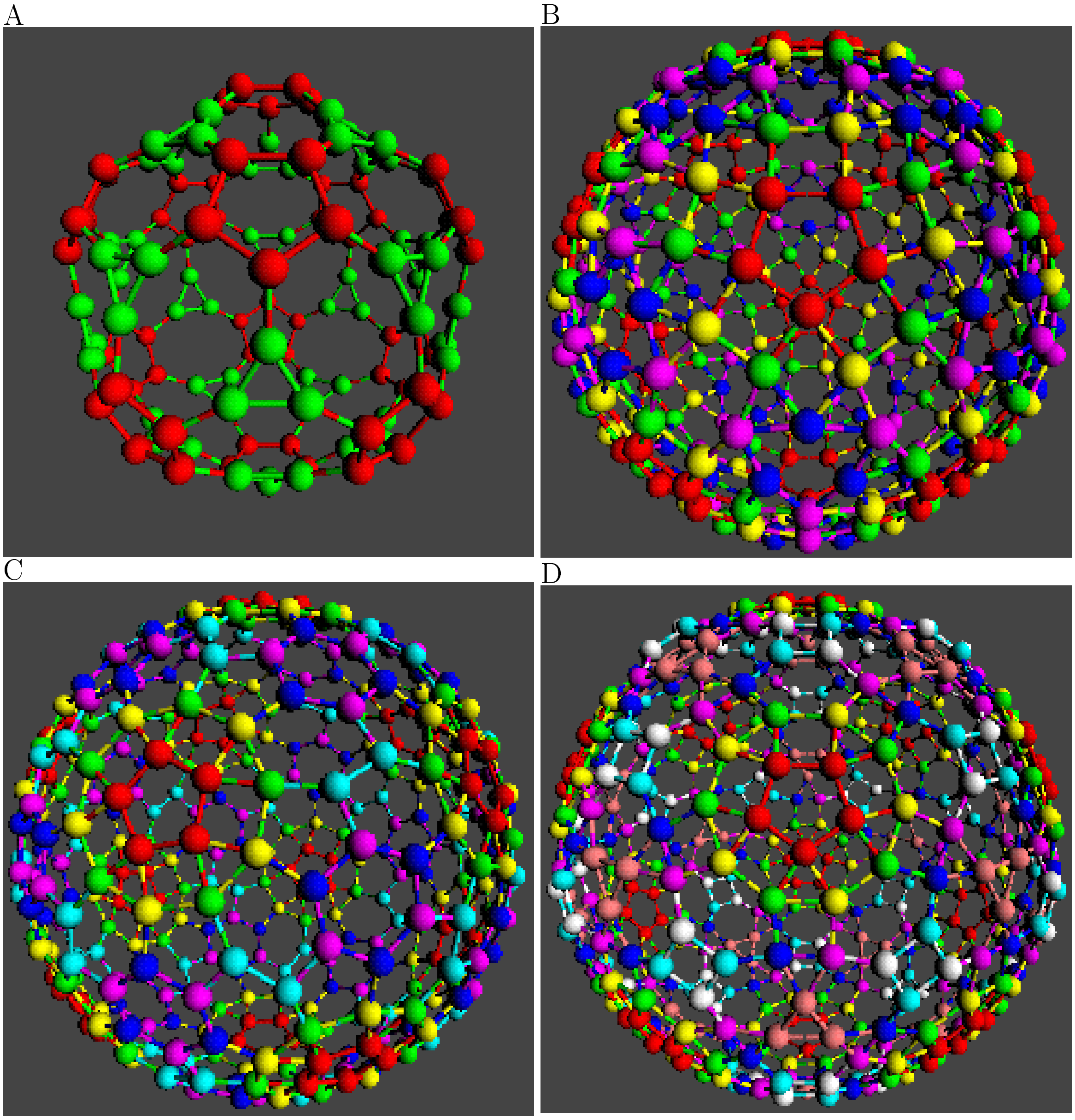

Figure 1-4: A variety of non-quasi-equivalent shell geometries. The figures show A: "T=2," B: "T=5," C: "T=6," and D: "T=8" shells.

selection between the two different shell sizes [101, 1]. Furthermore, the T=7 bacteriophage P22 can produce T=4 shells when it is grown without its scaffolding protein [31]. Experimental work suggests that scaffolding protein could play a crucial role in switching between these two geometries in the case of P22 as well [110, 109]. Local rules models have previously shown how a simple switching mechanism can select between these two geometries [7]. More recent work with static local rules models, utilizing the extensions for dynamic conformational shifting and under-determined binding, has shown how scaffolding proteins might act within a local rules framework to implement such a switch [5].

Static local rules work has also recently been applied to studying aberrant structures produced by papovavirus coat proteins in response to mutations or changes in solvent conditions. Although papovavirus normally produces icosahedral shells with an all-pentamer T=7 lattice, its coat protein can be induced to produce a variety of other forms when assembled *in vitro* (outside its normal cellular environment) including different sizes of icosahedron and rhomboid forms [93, 65]. Baculovirus expression of papovavirus coat proteins, in which the coat proteins are added to the genome of another virus in order to cause their expression at high levels in insect cells, can also induce different types of tube-shaped structures [76]. In addition, several mutant coat proteins have been found to induce such structures as abnormally small icosahedra, misshapen spirals, and long ribbon-like forms [95]. Local rules models have been applied to explain why these forms can result from variations on a single coat protein; while some have eluded explanations, others appear to result from simple adjustments of ordinary papovavirus local rules [95]. Hypotheses into how variations on one set of local rules can give rise to multiple structures may help in analyzing the assembly pathways of the normal virus and the mechanisms by which it selects its form.

Static local rules models have even proven useful for analyzing crystals and glasses, showing that local rules have applications beyond virus assembly. Extensive work has been conducted on adapting local rules principles and simulation tools to the study of silica glasses [46, 45, 50, 51]. These models applied local rules to structures of only a few atoms that form the repeating units of many crystals and glasses. Such local rules models have been used to study the structural properties of these materials, and in particular to examine how they respond to radiation damage. Studying the responses of glasses to radiation is important to finding safe long-term methods for containing nuclear waste [118]. Work with static local rules models has continued in trying to develop techniques to better predict how glasses respond to radiation damage that disrupts their bonding structure and forces bond rearrangements. A technique using maximal matching algorithms to chose patterns of bond rearrangement in a local rules-based system was recently developed to better match known properties of glass systems [44].

Prior Kinetics Simulation Work

Despite the many applications of static local rules models, there are some important processes they cannot capture. These include the effects of populations of subunits on one another and the role of dynamic mechanisms, such as conformational shifting and the association and dissociation of subunits. In order to address such concerns, work was begun on a newer simulation model to more realistically represent aspects of the dynamic nature of the assembly of populations of shells. This newer simulation tool was meant to be used in conjunction with theoretical modeling and with the earlier static local rules simulator to extend the range of phenomena that could be described. Earlier work on this simulation tool was the subject of my Master of Engineering thesis [94].

The simulator was originally intended to meet several pre-specified design goals. It was designed to implement a physical model meeting certain requirements: a "soup" of proteins that could assemble without constraint on order or rate of assembly; a realistic notion of binding kinetics capturing the association and dissociation of proteins; a model of proteins with adjustable sizes, shapes, and masses; and variability of some physical quantities such as temperature. The simulator was also meant to be easy to use, in the sense that users with limited experience should be able to modify simple parameters, run simulations, and examine the results. It was considered acceptable, however, for less frequent or more specialized operations to require greater user knowledge. In addition, the simulator had to be fast enough to simulate about one thousand particles on a time scale long enough to grow shells. Moreover, it had to do so while remaining accurate enough to give reasonable Newtonian dynamics on short time scales and reasonable global thermodynamic behavior over arbitrarily long times. The simulator had to accomplish these goals while retaining a fast enough user interface to be interactive. In addition, it could not unreasonably burden shared resources, such as memory and network bandwidth. The simulator also had to run efficiently on the available hardware: a Connection Machines CM-5 parallel supercomputer and various serial Unix machines. Finally, the project had to be sufficiently limited that a working simulator could be completed in about one year.

The overall design of the simulator consisted of two parts: a parallel program implementing the physical model and a serial user interface. The parallel program established a hard-coded physical model maintaining particle state and governing inter-particle interactions. It was meant to run on fast, parallel computers to give high performance to the most computationally intensive simulation routines. The user interface was meant to provide a configurable, easy-to-use local program that would control interactions with the parallel routines. Separating the two programs was intended in part to allow them to take advantage of different kinds of hardware relevant to their distinct operations: high-performance supercomputers for numerically intensive computations and workstations containing sophisticated rendering hardware for implementing a graphical user interface.

The physical model created a representation of assembly subunits in a simulated solution. Subunits could have different shapes and binding configurations and could shift between different configurations dynamically. Subunits interacted with one another through collisions and through binding interactions, modeled with spring forces between bound subunits, using a Monte Carlo model of the forming and breaking of interactions. Subunits moved in a simulated solution that acted on them through a combination of damping and random perturbations. Users could control some properties of the solution, such as the temperature and the strength of the damping interaction.

The user interface consisted of a graphical interface built on top of an interpreted control language. A Lisp-like command interpreter allowed users to interact with the simulation through various primitive functions. It also provided a versatile command set to facilitate the creation of higher-level utility functions for customization and automation of simulator behaviors during complicated simulation runs. Most normal user interactions with the simulator, however, could be conducted through a button-based graphical interface and a rendered graphics display that acted as a front end to the command interpreter.

Finally, an interface was needed to connect these two components together. The interface consisted of a fixed series of simple commands and queries initiated by the user interface to which the physical model would respond. The commands controlled such behaviors as defining node and bond types, creating new nodes, setting simulation constants, and advancing simulation time. Queries could test aspects of the states of the entire simulations, of individual nodes, or of simulation constants. The interface was implemented through TCP sockets, allowing the two parts of the code to communicate between machines supporting TCP/IP protocols.

An evaluation conducted following the initial construction of the simulator suggested that it had predominantly accomplished its original design goals, although there was room for improvement. The original evaluation concluded that the simulator had generally met the requirements for functionality, providing a more realistic physical model than was created by prior work and a fairly versatile and easy-to-use user interface. Some caveats were needed in that many aspects of the physical model were hard-coded and therefore difficult to change. In addition, the simulator could become hard to use when extensive customization or unanticipated functionality needs required direct access to the control language. Several miscellaneous goals, such as the need for portability while making good use of the hardware available to the project and the need to produce a usable simulator within a pre-specified time frame, were judged to have been met well. The evaluation of processing speed was less favorable, concluding that while the simulator was fast enough for some non-trivial applications, it was nonetheless far slower than would be necessary to apply it to the full range of problems for which it was developed.

1.2 Motivation for the Present Work

This early work with a local rules dynamics assembly simulator left a few important areas incomplete. As the evaluation in the past section showed, the simulator fell short of its original expectations in some important ways. But perhaps even more significant than failures to meet the design goals were the inadequacies that only became apparent with experience. This section examines the weaknesses of the original work and how they have motivated the continuation of the project. It specifically explores difficulties with the earlier user interface design, insufficient versatility of the physical model, the opportunities created by changes in available hardware and software systems, and the initial lack of applications.

The user interface was an important focus of such concerns. It became clear that promptly responding to user commands wherever possible was nearly as important as functionality in meeting the ease-of-use criterion. In particular, difficulties in visualizing simulation state were a major obstacle to efficient use. Furthermore, memory overhead of the user interface, which had been neglected in the original design goals, came to be a serious constraint. Other aspects of ease-of-use have been de-emphasized, however, as it has become apparent that the original goal of creating a powerful and versatile toolkit that can be distributed to biologists with limited computer experience was unrealistic. Making the simulator directly usable by that audience would require the development of tools to assist in design and debugging of simulations as well as more robust numerical methods that do not require as much knowledge on the part of the user for efficient application. Unfortunately, such developments are beyond the scope of what can reasonably be accomplished by a small project in a limited amount of time.

Another area where improvements came to be seen as necessary was in the physical model, which proved to be insufficiently versatile as the range of phenomena explored was broadened. More sophisticated conceptions of such processes as conformational shifting, subunit binding, and Brownian motion have proven important to modeling realistically the full range of phenomena of interest to us.

Changes to the hardware and software environments available to the project created another opportunity to improve on the system. Portability came to be seen as more of a concern than was originally appreciated, as access to the most sophisticated hardware platforms was often limited. Furthermore, the Connection Machines CM-5 that had been the primary platform for the prior work was replaced with two varieties of shared-memory symmetric multi-processor (SMP) machines: a Sun Ultra Enterprise 5000 SMP based on Sun Sparc processors and a DEC Alpha 4100 cluster based on DEC Alpha processors. The presence of true shared memory created a possibility for greatly improving the simulator's parallel algorithms, for which inter-processor communication had previously been a major bottleneck. Related to this hardware change were significant changes in the Cilk multithreading system [9], which had been used to parallelize the simulator. These changes primarily consisted of simplifying Cilk and better adapting it to SMP architectures. Revising the central algorithms and numerical methods of the simulator in order to adapt them to these new environments therefore became an important project. In addition, the appearance of improved computer graphics hardware as a standard component of even low-end personal computers and of freely available software-based rendering support such as the Mesa rendering library [83] created an opportunity to extend the range of portability of the user interface portion of the code without sacrificing a fast, high quality graphics display.

Finally, it was necessary to expand upon the few applications offered by the original work. The original work explored only two simple applications, neither of which used more than a small range of the capabilities the new simulator offered. A central goal of work since the simulator was first created has been applying it to exploring many aspects of the problems for which it was designed. Another important focus of applications is testing the capabilities of the simulator and exploring how it might be applied to projects beyond those envisioned when it was first created.

1.3 Summary of Major Revisions

Although this is a continuation of the work reported by Schwartz [94], it has evolved considerably since then in order to address the points raised in the previous section. This section describes the most significant changes since the original implementation in the simulator and how it has been applied. Areas that have undergone significant changes include the physical model, user interface, numerical methods, and discrete algorithms. In addition, many new applications of local rules dynamics have been developed. Although later sections occasionally discuss how the simulator has evolved, they primarily focus on its most recent state. This section is meant to provide a basis for comparison with the earlier work.

The physical model has been altered in several important ways. Conformational switching now uses combined activation and deactivation energies, rather than a single energy of thermodynamic stability per conformation, in order to provide more user control over the switching process. The Brownian motion model has been entirely replaced with a model meant to provide higher diffusion while still maintaining the stability benefits of moderate damping. In addition, a more thorough theoretical analysis has been conducted of the expected behavior of the major probabilistic aspects of the physical model, including conformational shifting, binding, and Brownian motion.

The user interface has also been significantly changed. A new variable lookup algorithm has been installed to produce faster execution. The garbage collection routines have also been improved to allow more frequent garbage collection and thus reduce memory overhead. In addition, several extensions to the control language have been added, most notably a new construct for explicit iteration. Minor language changes include converting several routines from special forms to primitive procedures and accelerating the dispatching of special forms. The graphical user interface has been almost entirely rewritten. A series of new control language constructs allows much faster updating of graphics, making real time manipulation possible even for large simulations. In addition, a rewritten system of pop-up menus and keyboard and mouse short-cuts has been developed to promote ease of use.

The numerical methods have been augmented primarily by the introduction of a new method that can be used in place of the previous primary numerical method. The new method is based on waveform relaxation [64]. It provides an alternative means of implementing the physical model, which can yield faster execution under some circumstances. In addition, both the old and new numerical methods have been adapted to run on SMP architectures using version 5 of the Cilk runtime system [73]. This adaptation has required substantially altering the parallel code compared to that developed for use with Cilk version 2 [72] on a distributed memory Connection Machines CM-5 computer. In particular, the earlier code had required inefficient routines to mimic message-passing within a multithreading framework by explicitly posting threads with long argument lists in order to transmit data between processors and keep them up-to-date on each other's progress; the hardware-level shared memory of the SMP machines currently in use has eliminated this need. In addition to changes in the code, a new suite of performance tests has been developed and applied to evaluate the effectiveness of both numerical methods.

The major change to the discrete algorithms was the creation of a new version of the collision-detection algorithm for use with the waveform relaxation method. The waveform relaxation method requires creating representations of particle positions continuously over each time step, rather than just at discrete time points. It was therefore necessary to create a new collision-detection algorithm that could efficiently handle collision detection at arbitrary time points. This improvement was accomplished by augmenting the previous bucket-based algorithms with time stamps, allowing nodes to occupy different buckets at different times, with a node's position over all time captured by piece-wise cubic functions. In addition to the development of this new version of the algorithm, new theoretical analyses have been developed for both variants of the collision-detection algorithm and for the binding partner location algorithm.

Probably the most extensive additions to prior work have been in the area of applications. Simulation models have been developed for several viral shell geometries and assembly mechanisms. In addition, several major simulation experiments have been conducted, applying the simulator to explore questions regarding virus shell assembly. Several smaller applications have also been created to demonstrate the applicability of the simulator to different types of systems and to provide a basis for future work in those areas.

1.4 Contributions of this Thesis

This thesis presents the local rules dynamics model as a means of exploring previously inaccessible problems in macromolecular self-assembly. It describes the ideas behind local rules dynamics, discusses how it has been applied, explains the physical model as it is currently implemented, and presents techniques for its efficient implementation. In accomplishing this goal, the thesis makes the following specific contributions:

- *The local rules dynamics simulation model:* This thesis describes local rules dynamics simulation, a method for applying ideas from the local rules theory [8] to accelerate simulations of self-assembly systems in which local binding patterns are well understood. By abstracting away the underlying physical implementation of binding and simulating only its behavior, computationally tractable simulations can be conducted of greater complexity than was possible with previous techniques.

- *Applications:* Several significant applications of local rules dynamics are presented. These applications serve both as demonstrations of the range of capabilities of the local rules dynamics model and a local rules dynamics simulator and as experiments of independent value in providing insight into their underlying physical systems.

- *Implementation:* A computer simulation tool created to implement the local rules dynamics model is described. This tool includes a detailed physical model based on local rules dynamics, a series of algorithms and numerical methods designed for efficient simulation of the model, and a versatile graphical user interface. This implementation demonstrates that a computationally efficient local rules dynamics simulator is feasible and practical. Moreover, the techniques developed may have application to other simulation problems.

Chapter 2

The Local Rules Dynamics Model

A computer simulator necessarily abstracts and simplifies the physical system it models. The crucial challenge in the design of a simulation tool is developing abstractions that are sufficiently simple so as to be computationally tractable but that capture some aspects of the physical system the simulator models well enough for it to have predictive value. This chapter describes an abstraction, which we call the local rules dynamics model, for creating computationally tractable simulations of self-assembly dynamics. It first considers how similar problems have been approached in the literature as a means to explain the motivation behind and the contribution by the local rules dynamics model. It next explains the construction of the local rules dynamics model through a combination of techniques from lower- and higher-level methods and explores how the model can be understood in the context of the other work examined. It then considers some limitations of local rules dynamics. Finally, it outlines the discussion in the remainder of this thesis on the application and implementation of the model. The simulation model described here was developed in collaboration with Bonnie Berger, Peter Prevelige, Jr., and Peter W. Shor and was presented in Schwartz et al. [98].

2.1 Other Methods for Self-Assembly Simulation

To understand the value of local rules dynamics, we need to consider the alternatives. This section considers how various methods have been applied to virus assembly and related self-assembly problems. For the purposes of this analysis, the methods are subdivided by their degrees of abstraction. This section argues for the existence of a gap in the capabilities of the existing methods, as a result of which some important questions cannot be addressed by any of them.

Models applicable to self-assembly and related processes can be roughly divided into high-level and low-level models. High-level models include a variety of abstract techniques for capturing behaviors related to self-assembly. Among these are optimization-based methods, which can rely on continuous optimization of a potential function or combinatorial optimization techniques; Monte Carlo methods for abstract simulation, which use some model of transitions between discrete states, usually in conjunction with Metropolis dynamics [75]; and differential equation models for the interactions of subspecies in an assembly reaction. Static local rules can also be considered a high-level model. Low-level models attempt to create highly realistic representations of the forces underlying self-assembly processes. Low-level models are generally based on molecular dynamics [2] working at the atomic level, although for protein simulations, amino-acid level interactions may also be used.

The key advantage of low-level models is that they generally assume little about how a system should behave beyond well-established basic laws, making it likely they are adequately capturing the behaviors they are meant to model. Low-level molecular dynamics methods have proven useful to a wide variety of biological problems, often with only straightforward application of the basic technique [53]. The primary research effort on such models in biochemistry has been directed at the protein folding problem. Although such low-level models have not been applied to virus assembly, they have been used with the simpler problem of amphiphile self-assembly [117] over short time scales. Amphiphiles are a class of molecules, made of hydrophobic and hydrophilic segments, which tend to assemble on surfaces. Examples of amphiphiles include surfactants and block copolymers. Low-level molecular dynamics has also been applied to the protein docking problem, which involves finding how two proteins are likely to pack against each other [26].

The central obstacle to using molecular dynamics approaches is computational cost. Atomic-detail simulations can handle only a few proteins over a period of at most nanoseconds [29], which is far less than is required for even simple virus-like self-assembly phenomena. For protein simulations in which hydrophobic forces are an important factor in inter-subunit interactions, molecular dynamic simulations require explicit models of large numbers of solvent molecules to get realistic behavior in terms of low-level forces [11, 32], further degrading performance. In addition, even the highly detailed force-field calculations typically used for molecular dynamics simulations may be insufficiently accurate to distinguish true energetic minima from nearly-minimal energy states [78, 41]; using sufficiently realistic potentials that these methods will be reliable could be expected to further exacerbate the computational problems. Low-level methods may thus be able to look at the immediate collision of a single protein with a small, partially-formed shell, but they cannot hope to model in any detail the complicated trail-and-error process that is likely involved in even the transition from one stable shell intermediate to the next. As a result, low-level molecular dynamics methods, despite their importance to macro-molecular simulation in general, have not had much applicability to any but the simplest macromolecular self-assembly problems.

Several simplifications have been attempted to improve run time while retaining the advantages of a low-level model. Stochastic Dynamics [43, 25, 114] attempts to reduce the high cost of simulating the solvent by replacing an explicit solvent simulation with a combination of a modified force field and the introduction of random forces. Minimal off-lattice models, which attempted to deal with the cost of pure molecular dynamics via simplifications of aspects of the model, have provided predictions of time scales and mechanisms behind protein folding [108]. The validity of these models is supported by evidence that, at least in protein folding, low-resolution structural features, rather than atomic-level detail, are most significant in understanding kinetics and energy landscapes [3]. Collective variable models provide another simplification strategy, reducing simulation problems to a small number of degrees of freedom derived from a knowledge of the final structure [58, 62]. Stochastic dynamics [115] and a simplified molecular dynamics model [39] have been applied to amphiphile self-assembly, although not to viral self-assembly. All of the above models make great improvements in computational speed, often with little loss in the reliability of the model. Even with such performance improvements, however, molecular dynamics techniques are unlikely for the foreseeable future to approach the many orders of magnitude speed-up they would need to achieve the same level of detail on simulations of hundreds of proteins on a time scale of seconds as they now achieve with one or two proteins on a time scale of nanoseconds.

High-level models are generally designed for computational tractability, but come at the expense of detail. One strategy for high-level simulation is to treat the problem as that of studying the transitions between members of a set of pre-defined species of assemblies. Horton and Lewis [47] develop a technique for determining likely shell assembly pathways by locating the most energetically stable intermediates of each expected intermediate size. Zlotnick [124] uses a simplified model of virus assembly as the basis of a differential equation model, using energy changes between structures of different sizes to determine transition rates between the allowed species. Zlotnick's model can simulate reaction progress over time from an initial population of individual subunits to a final population containing complete shells.

The use of a fixed number of pre-defined states poses two basic problems. First, it requires assuming many details regarding the final structure and the nature of acceptable intermediates. This requirement may bias final results by making them dependent on the initial assumptions; for example, if assembly can proceed by both monomer addition and addition of larger complexes then assuming only monomer addition may produce invalid results. Furthermore, assuming possible final and intermediate structures limits the approach to problems for which those structures are well understood. This specifically precludes application of these techniques to studying malformations or non-virus self-assembly systems in which final structures have no definite form. Second, keeping track of all defined intermediates can be extremely memory-intensive, as the number of possible intermediates tends to grow exponentially in the number of assembly subunits in the completed structure. The kinds of simplification needed to deal with this memory overhead — considering only a small number of pathways, as done by Horton and Lewis [47], or greatly simplifying the completed structure to ensure a small number of assembly subunits per shell, as done by Zlotnick [124] — may bias the final answers in ways that are difficult to predict.

Marzec and Day [71] and Tarnai *et al.* [107] use variations on a strategy of optimizing a potential function to study shell assembly for a particular virus geometry characteristic of the papovaviruses. Marzec and Day show how the observed geometry corresponds to the minimum point in the potential energy produced by causing assembly subunits to repel each other according to a simple energy function on the surface of a sphere. Tarnai *et al.* similarly examine structures in terms of the combinatorial problem of creating maximally dense packings of assembly subunits on the surface of a sphere. Both approaches provide insight into the final geometries of shells. They are not of much use, however, in understanding the process of assembly.

Because few simulation approaches have been applied directly to virus shell assembly, it is necessary to look also to techniques applied to related problems in other fields. One important source of such techniques is the protein aggregation problem, which concerns a process by which proteins assemble into typically non-functional, irregular aggregates under some conditions. One strategy developed by Witten and Sander [122] uses a simulation of simple particles on a lattice that move about randomly and join other particles to form aggregates. These aggregates can themselves move about and join to others. A more detailed approach developed by Patro and Przybycien uses a model similar to local rules to build up simulated aggregates on a grid; this approach was applied first to irreversible [81] and then to reversible [82] binding models. An even more detailed such approach is that of Gupta *et al.* [38], which models aggregation on a lattice in terms of amino-acid resolution structures, allowing simulated proteins to fold and aggregate on a lattice simultaneously. Istrail *et al.* [49] use a similar approach to study the interactions of pairs of proteins. All of these approaches have the advantage of being fast enough that they can build structures on the order of the size of a virus shell in reasonable times. Moreover, because they are intended to examine irregularly-structured protein aggregates, these approaches can in principle apply to the study of malformed virus particles, at least within a lattice framework.

The problem of amphiphile self-assembly has also been approached by such models. Larson *et al.* [63] develop a Metropolis Monte Carlo method for simulating amphiphiles, allowing examinations of how structure formation depends on initial distributions of components. Another approach uses an Ising model for gathering high-level statistics from a simple model system [121].

These various high-level models are generally limited in that they assume a great deal about how a system behaves and otherwise limit the range of possible behaviors. The models are highly simplified, which is useful in establishing general principles but makes them less applicable to studying special cases. In addition, such simplifications as the use of a lattice may bias the range of structures and interactions that can occur by ensuring that assembly subunits can only affect one another from a small set of discrete neighboring positions; much as with the virus-specific models, these assumptions preclude much of the flexibility of lower-level models. Other common simplifications, such as the use of a two-dimensional model, may also call into question the applicability of the resulting simulations to real systems. Thus, while these models achieve computational tractability, they do so at the cost of a great deal of versatility and possibly reliability.

Some difficulties of both low- and high-level models might be overcome by coupling the two. Reddy *et al.* [89] apply this strategy to virus assembly. They use a low-level force-field calculation with the CHARMM energy-calculation package [10] to determine interaction energies between individual subunits based on atomic interactions derived from solved crystallographic structures. They then apply the search technique of Horton and Lewis [47] to locate the most locally favorable pathways for assembling complete shells from single subunits. This strategy provides some benefits of both high and low-level approaches, giving a detailed model of some important interactions while keeping computations tractable. It also retains some of the problems of pure high-level approaches, however. The approach shares some problems of that of Horton and Lewis [47] in that it is limited to exploring problems that assume correct assembly and in that it may make incorrect assumptions about pathways because of the need to limit the number of intermediates considered. In addition, the improvement given by applying low-level energy calculation is dependent on the assumption that crystallographic structures of completed virus shells provide a correct model of interactions in intermediate structures, an assumption called into question by the substantial conformational changes virus shells may undergo following shell closure [104, 34, 84].

We can better understand the relationship between these different models and their relevance to virus shell assembly by looking at potential virus simulation techniques in terms of hierarchies of abstraction levels. At one extreme of abstraction, we can place detailed quantum mechanical models, capturing the full current understanding of sub-atomic physics. Quantum-level models would produce substantially more realistic simulations than even the molecular dynamics models considered earlier in this section. They are, however, currently inapplicable to studying molecular self-assembly phenomena because of the computational cost of simulating even a few atoms at such a detailed level. At the other extreme, we can place mathematical models of the progress of the infection of many viruses within an entire organism, or even an entire population of organisms. While such an abstract model can capture enormously complex systems, it is useless for studying virus assembly because it abstracts away of all of the details relevant to the behavior of individual viruses. In between these extremes lie the various methods considered earlier in this section, ranging from the low-level molecular dynamics models, which are useful for studying some self-assembly phenomena but are still too computationally costly for studying the overall virus assembly process, to the various high-level models, which can represent shell assembly but abstract away many important details of the process. Figure 2-1 shows how some potential models may be grouped into hierarchies according to levels of abstraction. As the models move from greater realism at the bottom of

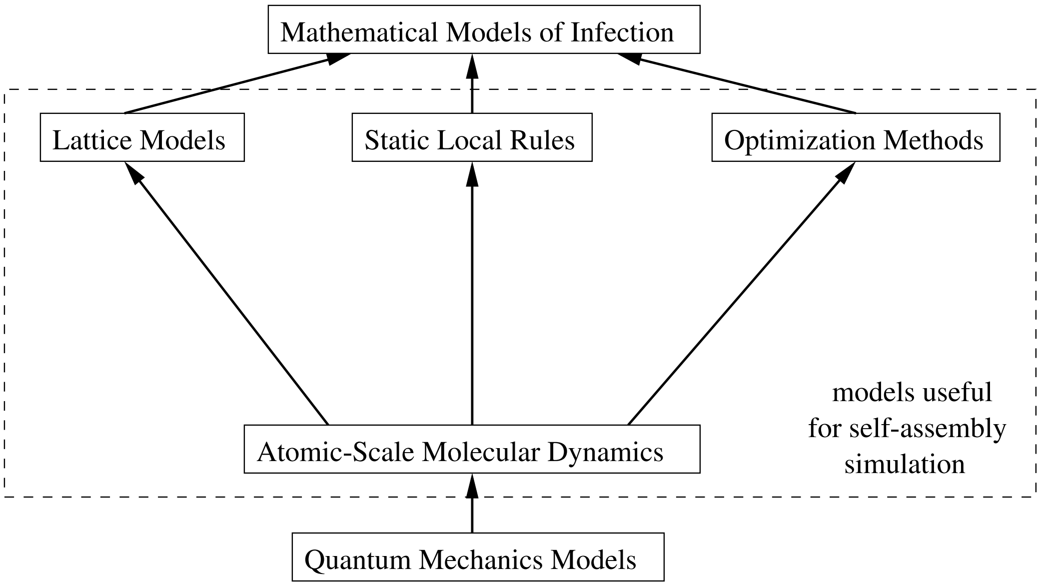

Figure 2-1: Illustration of how some of the simulation methods considered can be related according to hierarchies of abstraction, with more detailed methods at the bottom and more abstract methods at the top. The range of methods currently useful for aspects of self-assembly simulation is also marked.

the figure to greater abstraction at the top, they lose the ability to capture some important phenomena. They simultaneously gain computational efficiency, however, allowing them to examine larger-scale phenomena than the lower-level models can. While the figure shows a general hierarchy of methods from realistic to abstract, it also illustrates another important feature of abstraction: abstraction can proceed along multiple dimensions. Thus, although static local rules, lattice models, and optimization methods can be considered to have comparable levels of abstraction, they choose different details to abstract away, making them suitable for different kinds of problem.

The limits of the varying approaches suggest that there are significant gaps between the kinds of experiments that can be performed with prevailing high-level self-assembly models and those that could ideally be performed with low-level models if computational tractability were not a concern. In particular, there appears to be no approach suitable to tracking the movements and assembly dynamics of a large-scale self-assembly system involving complicated assembly subunits, as would be required to examine many aspects of viral protein shell assembly. All of the available high-level models assume a great deal about aspects of the behavior of self-assembly systems, which makes them unsuitable for questioning those behaviors. As figure 2-1 suggests, a new approach to fill in these gaps might be created by selecting a new point on the hierarchy between the high-level and low-level models. It is not obvious, however, which abstractions to employ relative to the low-level methods or which elements of realism to add back to the high-level methods in order to create a model suitable to the questions we wish to examine.

2.2 The Value of Local Rules Dynamics

This section describes an abstract simulation model intended to fill in some of the gaps between the capabilities of the high- and low-level models discussed in the previous section. This model, which we call local rules dynamics, combines elements of the high-level local rules model and lower-level molecular dynamics to simulate phenomena inaccessible to any prior approach. Local rules dynamics was created to provide a "middle ground" technique between the intractable low-level models and the simplified high-level models to extend the range of questions that can currently be addressed by simulation techniques. Local rules dynamics provides a flexible framework in which a few key abstractions produce computational tractability even though the model retains some useful aspects of low-level models that are lost in the high-level models. Local rules dynamics also provides an important advantage in that the central abstraction used to simplify the model — the use of local rules binding patterns in place of lower-level forces — is based on an understanding of observed binding patterns in a system and not on an arbitrary property of the model, such as the choice of a particular lattice. The remainder of the section explains the abstractions underlying local rules dynamics and discusses how it is meant to be applied.

The central principle behind local rules dynamics is to make a reasonably realistic model of assembly dynamics via a molecular dynamics-like off-lattice framework for particle interactions while using the local rules abstraction to make the model computationally tractable. By assuming that the possible patterns of binding interactions between subunits are known in advance, it becomes possible to abstract away the complex combination of forces — such as electrostatic, hydrophobic, or Van der Waals interactions — that produce inter-subunit binding, thereby greatly simplifying the computational problem of simulating them. It is similarly assumed that the importance of some other aspects of subunits, such as their detailed atomic structures, is adequately captured by these binding patterns for the purposes for which the model is intended. Given these assumptions, local rules dynamics uses a simple abstract model of assembly subunits that can move with full translational and rotational freedom and interact in a three-dimensional off-lattice solution. Interactions between subunits follow local rules binding patterns, but with extensions to allow for the probabilistic association and dissociation of subunits. Local rules dynamics also allows for other extensions, such as the shifting of assembly subunits between distinct conformations, which may vary in binding patterns or other physical properties. The model is meant to allow for qualitative simulations of self-assembly dynamics through which one can examine the behaviors of different assembly systems, explore those systems' robustness to parameter variations, and compare the merits of theoretical models of different aspects of the assembly process.

Figure 2-2 situates local rules dynamics within the abstraction hierarchy of figure 2-1. As figure 2-2 illustrates, local rules dynamics fills a gap in the previously existing methods by lying at an intermediate point between low-level molecular dynamics and high-level static local rules. It accomplishes this by retaining some elements that contribute to the realism of atomic-scale molecular dynamics, such as an off-lattice Newtonian dynamics model and a representation of populations of free subunits, while simultaneously adopting some of the abstractions underlying static local rules models, in particular the local rules notion of binding interactions. The result of this intermediate level of abstraction is a model designed to examine some phenomena that are abstracted away by the higher-level models, but whose simulation would be

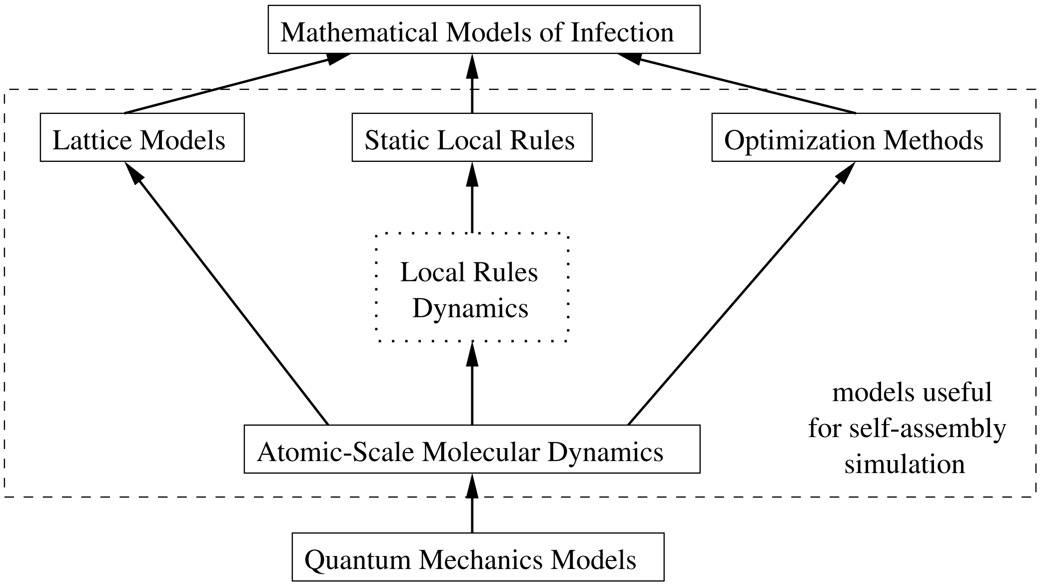

Figure 2-2: Illustration of how local rules dynamics fits into the abstraction hierarchy of figure 2-1.

computationally intractable with lower-level models.

Local rules dynamics also provides a flexible framework that is capable of growing as computational resources and problems of interest dictate. As currently implemented, the model provides significant flexibility to manipulate such parameters as the shapes of assembly subunits, the energies of their interactions, and their structural properties. Later in this thesis, we will see example of how this flexibility can, within limits, adjust the level of abstraction used by local rules dynamics. Section 3.1 shows an example of increased abstraction, in which local rules are applied to multi-protein assembly subunits rather than individual monomers. Section 3.5.2, however, shows an example of reduced abstraction, in which local rules dynamics is used to create an atomic-scale model. Moreover, the general local rules dynamics framework might be reimplemented to provide more realistic behavior in such areas as energy function, thermodynamics, collision dynamics, or structural mechanics, as computational resources allow. Thus, local rules dynamics provides a means of conducting some useful simulations with currently available computational resources, but also of refining those simulations and the range of problems they can address until the model might ultimately converge on the low-level approaches it is meant to supplement. In terms of the abstraction hierarchy of figures 2-1 and 2-2, local rules dynamics can move between levels of abstraction, approaching the low-level molecular dynamics where computational resources allow and approaching static local rules where low-level detail is superfluous. In this way, local rules dynamics can provide a framework for long-term study of self-assembly phenomena similar to the framework lattice Monte Carlo methods have provided to protein folding, in which from simple beginnings, such as the Gō [105, 35] and HP lattice [27] models, increasingly flexible lattices [102] and reasonable energy functions [106, 15] have led to detailed models of protein folding.

There are, however, several caveats to be considered when applying a local rules dynamics model. Because local rules dynamics is still more abstract than the lower-level models and frequently more computationally costly than the higher-level models, it is unsuitable for some classes of problems. The limitations of the local rules dynamics model must therefore be considered when deciding whether it is appropriate for examining any particular question.

Probably the most important limitation of local rules dynamics is that the abstractions in the model generally make it useful only qualitatively. For example, the model does not provide a reliable means to deduce the rate of assembly of a structure given low-level information about the energetics of subunit interactions, although it may be useful in qualitatively comparing how the rate would relate to that produced by a change in parameter values. This limitation suggests that the model is not useful in evaluating quantitative behaviors arising from specific parameter choices without substantial refinement.

In addition, the model is limited to small simulations that can only be repeated a limited number of times due to computational costs. Given current computational resources, the model is limited to the order of one thousand assembly subunits, far less that the tens of thousands that might be found in an actual infected cell. As a result, the model is likely to be more susceptible to random fluctuations than are actual virus assembly systems, making it harder to trust that results will be reproducible.

Another potential objection to the model is its large number of adjustable parameters. Users can manipulate many aspects of the behavior of individual subunits and the overall solution, potentially altering assembly dynamics in a variety of ways. Having many parameters makes it questionable how universal results derived from simulations might be. A highly adjustable model, which lacks the conceptual simplicity of successful models such as Dill's HP lattice protein model[27], will always be susceptible to charges that experimenter bias influences its results. Extra caution is therefore required in drawing conclusions based on simulator behavior.

2.3 Implementing and Applying the Local Rules Dynamics Model

The remainder of this thesis covers applications of the local rules dynamics model and implementation of a simulation tool based on local rules dynamics. Chapter 3 describes a variety of applications of local rules dynamics. These applications include examinations of aspects of viral protein shell assembly that cannot be addressed by other techniques, as well as examples of other systems addressable by local rules dynamics. The applications are presented both to better explain local rules dynamics and to demonstrate its value. The remaining chapters describe the implementation of a local rules dynamics simulator that was used to run the simulation experiments described in chapter 3. The basic elements of the simulator's implementation presented are the detailed physical model it implements, in chapter 4; the major numerical methods used in the implementation, in chapter 5; some key discrete algorithms, in chapter 6; and the user interface, in chapter 7. Chapter 8 draws some brief conclusions about the value of the model, what it teaches us about self-assembly systems, and how it might be improved upon in later work.

Chapter 3

Applications

This chapter describes several applications of local rules dynamics simulations. Foremost among these are various ways our local rules dynamics simulator has been applied to studying virus shell assembly. First, this chapter examines local rules dynamics models for different shell geometries and growth strategies. Second, it covers a group of experiments demonstrating how rate and fidelity of assembly of a model system depend on some assembly parameters. Third, it covers two series of experiments comparing different theoretical models for the source of an assembly behavior called nucleation-limited growth. Fourth, it describes a series of experiments demonstrating how a local rules-based strategy for anti-viral drug design can suppress correct shell assembly in a model system. Fifth, it discusses applications to other self-assembly systems for which local rules dynamics simulations may be useful: the helical polymer actin, borosilicate glasses, protein aggregates, and novel self-assembly systems. These additional applications are primarily meant to serve as a proof of concept for the broader applicability of local rules ideas in general, and the local rules dynamics model in particular, to a variety of self-assembly systems. Chapter 4 provides a detailed description of the physical model used by the simulator and of the specific simulation parameters referred to in this chapter. Some aspects of the physical model implemented by the local rules dynamics simulator have changed during the time that these experiments were being conducted, and it should therefore be noted that specific simulation parameters, such as binding energies, may not be comparable between distinct simulation experiments.

3.1 Shell Geometries

One basic application of the simulator is showing that local rules dynamics can model different shell geometries and different means by which they can form. This section describes local rules models for three different geometries: T=1, T=3, and papovavirus "T=6" geometries. Furthermore, the three models use three different growth strategies. The first grows by addition of individual proteins one at a time to a shell. The second grows by the assembly from individual proteins of larger subunits, which then form into shells. The third grows directly from multi-protein subunits, which are treated as the basic local rules building-blocks of the shell. This section explains the three models and how they were implemented with local rules dynamics. It argues that the experiments support the value and versatility of the local rules dynamics model.

There is considerable variability in geometries and growth patterns among actual viruses. Several quasi-equivalent virus shell geometries have been observed in nature. In addition, there are non-quasi-equivalent special cases, such as the papovavirus shells, in which capsomer (pentamer or hexamer) building blocks are arranged as in a T=7 shell but with pentamers in place of hexamers [88, 4]. Furthermore, even a single geometry may be assembled in multiple ways. For example, the T=7 shell of bacteriophage P22 is built by addition of one coat protein at a time [85] while the coat proteins of another T=7 bacteriophage, HK97, first form pentameric and hexameric capsomers that then join together to form complete shells [123]. Being able to model some of this variability in actual shell formation is important both as a proof of concept for the simulator, showing that it is sufficiently versatile capture a wide range of assembly phenomena, and as a preliminary step to applying it to other problems involving different shell geometries and assembly patterns.

The simplest icosahedral shell geometry is the T=1 shell, for which I have produced a simulation model of assembly by monomer addition. This structure is designed to assemble by addition of individual coat proteins following a nucleation event in which five proteins must come together in a short time span to form a pentameric nucleus. The simulation requires only a single rule, specifying the ideal binding angles for the single T=1 binding environment. Figure 3-1A shows a simulated T=1 shell generated in this fashion.

I have also produced simulated T=3 shells using an alternate strategy. Whereas the T=1 shells use monomeric assembly subunits to model the kinetic behavior of assembly by monomer addition, the T=3 model is designed to model assembly by pentameric and hexameric capsomers, but still using monomeric assembly subunits. Thus, binding energies are biased to favor forming binding interactions within capsomers over those between capsomers, causing most subunits to assemble quickly into capsomers. The capsomers then more slowly come together to form completed shells. Binding enthalpies of association and dissociation are set to $-16,000$ for intra-capsomeric bonds and $-7,000$ for inter-capsomeric bonds, encouraging highly stable binding within each capsomer while making interactions between capsomers unstable unless multiple inter-capsomer interactions are formed. Figure 3-1B shows a simulated T=3 shell generated according to this assembly strategy.

I use an alternate strategy for modeling assembly by capsomers to build the "T=6" papovavirus geometry. The papovavirus geometry is unusual in that it consists entirely of pentamers of protein, rather than the mixture of pentamers and hexamers found in other T>1 virus shells. At least some papovavirus shells can be assembled *in vitro* from purified pentamers alone, suggesting that pentamers and not monomers are the immediate assembly subunit of their shells [92]. Furthermore, the shell contains only two pentamer binding environments, as opposed to six coat protein binding environments, making assembly by pentamers a natural simplification of the local rules. Thus, pentamers, rather than individual coat proteins, are used as the basic assembly subunit for the simulation. Figure 3-1C shows a simulated papovavirus shell generated by this mechanism.

The experiments described in this section support the basic principles of local rules dynamics and illustrate some of its versatility. The three models show the versatility

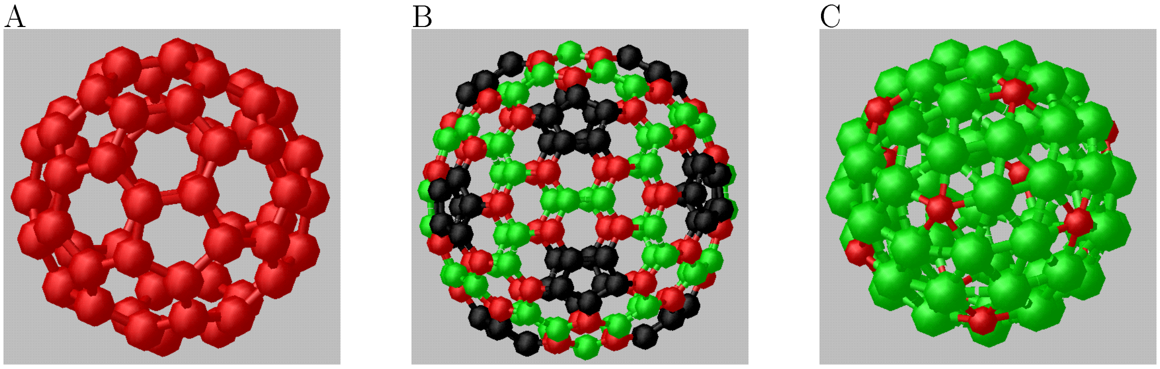

Figure 3-1: Three simulated virus shells. A: T=1 shell generated through monomer additions; B: T=3 shell generated through monomeric assembly subunits biased towards capsomer growth; C: "T=6" papovavirus geometry generated from capsomeric assembly subunits.

of local rules with regards to both shell geometries and assembly strategies. Local rules dynamics therefore extends the basic underlying assumption behind static local rules — that assembly can be directed using only the local information available to individual assembly subunits — by showing how strictly local information can give rise not only to final structures but also to complicated issues of assembly dynamics. The ability of local rules to model the structures therefore suggests it is likely to have broader applicability in modeling self-assembly dynamics. The papovavirus model illustrates the further point that local rules dynamics is a flexible framework capable of working at different levels of abstraction. Although we generally apply local rules dynamics to coat proteins, it can apply to assembly subunits of other sizes. By allowing more or less abstraction, local rules dynamics can trade off low-level detail for computational speed on applications for which the detail is unnecessary.

3.2 Parameter Variations

One important general application of computer modeling is testing the robustness of aspects of a system to parameter variations. For the local rules dynamics model, such an experiment can take the form of testing how observable aspects of shell assembly behavior are affected by changing some of the quantitative subunit parameters. This section describes one such experiment, examining the effects of changes in binding energy, binding tolerance, and concentration on the rate of growth and the fidelity of assembly of a simple shell assembly system. It concludes that model systems exhibit trade-offs between rate and fidelity of assembly as functions of each of the three parameters examined. The work described in this section was performed in collaboration with Bonnie Berger, Peter Prevelige, Jr., and Peter W. Shor and appeared in Schwartz *et al.* [98].

The experiments described in this section test how measures of growth rate and malformation rate vary as functions of the three parameters for a simple model of T=1 assembly. The experiments rely on measuring growth of nucleated clusters of assembly subunits. Nucleation refers to a phase at the beginning of assembly in which some minimal stable structure has formed, allowing the recruitment of more subunits into a growing shell. We use a model of T=1 assembly designed to have a five-particle nucleus; we therefore define any cluster of five or more particles to be nucleated. Based on this definition of "nucleated", we define three measures of growth. "Total growth" measures the total number of assembly subunits involved in nucleated clusters. "Off-pathway growth" measures the number of subunits involved in malformed clusters, those clusters whose binding pattern is not a (possibly proper) subset of a complete shell. "On-pathway growth," which is the difference between total and off-pathway growth, measures the number of subunits involved in clusters that are subsets of correctly formed shells. The experiments consist of measuring these three quantities at fixed intervals for seven simulations: a base simulation and two variants for each parameter value.

The experiments use a two-conformation T=1 assembly system. Binding angles were chosen to be ideal for a T=1 geometry. Each subunit has a mass of 100 kDa. They can shift between two conformations, a 3 nm diameter active conformation capable of binding to other active proteins, and a 1 nm diameter inactive protein that lacks the ability to bind. The different particle sizes were chosen to make the two conformations visually distinct, rather than for any anticipated effect on their behavior. Angle tolerances of 30° and distance tolerances of 10 nm were chosen. The workspace is 125 nm in each dimension. Each test uses 300 subunits. Testing is conducted over a period of 25,000 time steps, with data recorded at intervals of 5,000 time steps. The mapping between time steps and actual time should not be interpreted other than qualitatively. We suspect, however, that the concentration of subunits and binding tolerances are both biased to produce substantially faster growth than would be found for an actual virus shell.

The first parameter examined is binding energy, which controls the likelihood of forming and breaking binding interactions. Binding energies are adjusted relative to the base by -1 kcal/mole and -2 kcal/mole. Lower energies correspond to stronger interactions. Figure 3-2 shows the resulting data. As expected, lower energies yield consistently faster total growth. Moreover, with the exception of the 10,000 time point, lower energies lead to greater amounts of off-pathway growth. At various points in time, each of the three simulations has the most on-pathway growth. The final result is that the -1 kcal/mole simulation ends with the most on-pathway growth, although it has neither the most total growth nor the least off-pathway growth. The base simulation's low off-pathway growth rate is unable to compensate fully for its low total growth rate, giving it an intermediate final level of on-pathway growth. The high total growth of the -2 kcal/mole simulation is more than offset by its higher rate of off-pathway growth, giving it the least final on-pathway growth.

The second set of experiments focuses on the translational tolerance of binding, which controls how close potential binding sites need to be to their ideal relative positions before they can bind. From a chemical viewpoint, binding tolerances correspond to the configurational entropy of binding. Tolerances are adjusted to make them two and four times as lenient as they are in the base simulation. Figure 3-3 shows the resulting data. Total growth increases monotonically with increasing tolerance. As with the binding energy tests, the same relationship holds for off-pathway growth with the exception of time point 10,000. Also as with binding energy, each simulation has the most on-pathway growth at some point in time. Unlike with binding energy, however, the base simulation ends with the highest on-pathway growth despite its low total growth because its rate of off-pathway growth is substantially lower than that of the other two simulations. The quadruple tolerance simulation ends with an

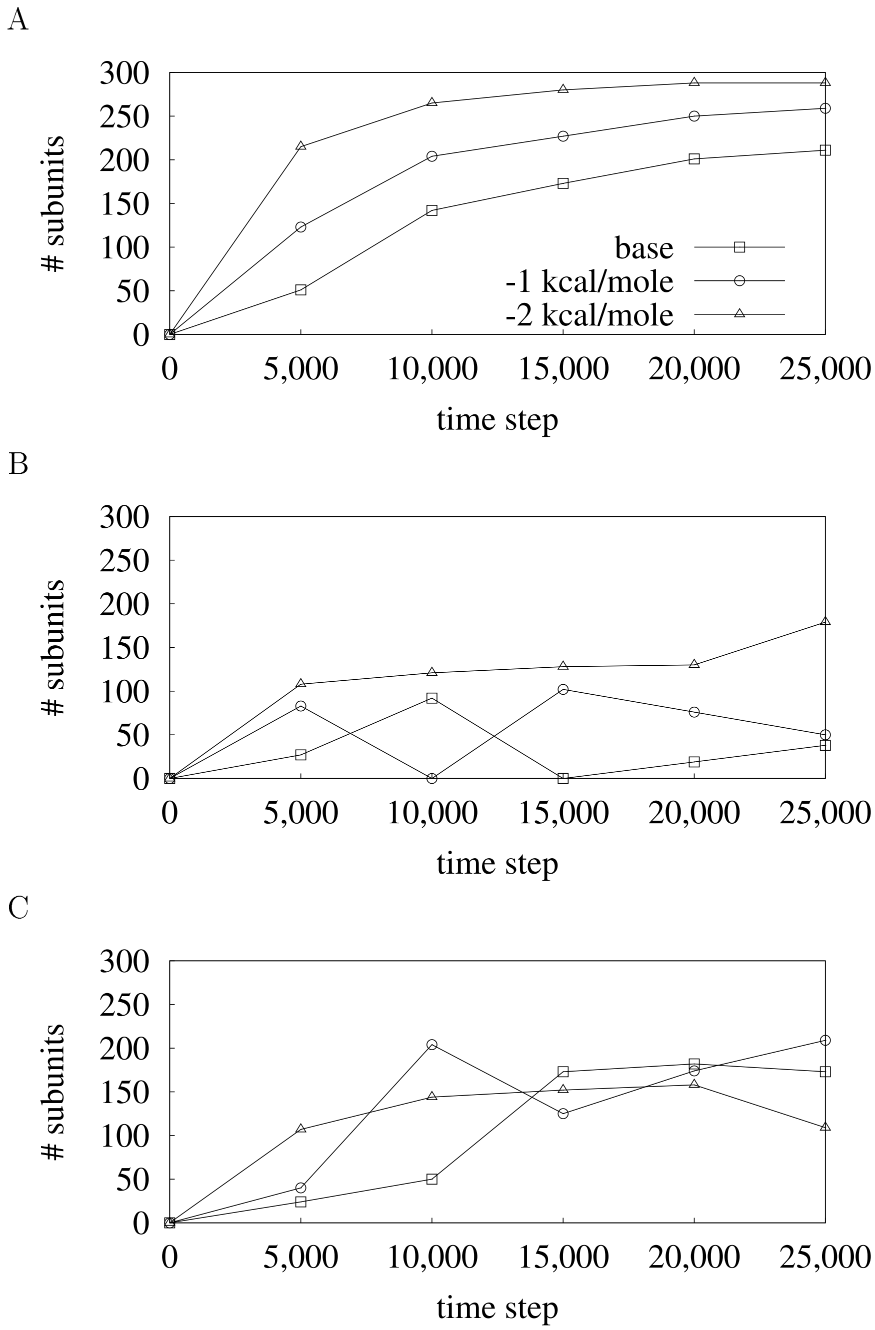

Figure 3-2: Graphs of growth as a function of binding energy for a simple model of T=1 assembly. The graphs show the numbers of subunits involved in nucleated clusters for A: total growth; B: off-pathway growth; and C: on-pathway growth for the base simulation and simulations in which binding energies are adjusted by -1 kcal/mole and -2 kcal/mole relative to the base.

intermediate level of on-pathway growth because its high total growth rate offsets its high off-pathway growth rate. The double tolerance simulation ends with the least on-pathway growth because it has somewhat lower total growth than the quadruple tolerance simulation, but nearly the same level of off-pathway growth.

The final experiments explore the effects of concentration. Concentration is adjusted by decreasing the simulation volume by two and four times while holding the number of assembly subunits constant. Figure 3-4 shows the resulting data. As with the other experiments, these experiments show a monotonic relationship between total growth rate and the parameter examined, with increasing concentration consistently leading to increasing growth rate. The off-pathway growth rate shows a more complicated relationship, with a generally monotonic increase in off-pathway growth as a function of concentration, but exceptions at both time points 5,000 and 10,000. The on-pathway growth shows each simulation leading at some point in time, but as with binding energy, the intermediate simulation ultimately has the most on-pathway growth despite having neither the most total growth nor the least off-pathway growth. The base simulation ends with an intermediate level of on-pathway growth despite having the lowest total growth due to its low rate of off-pathway growth. The quadruple concentration simulation ends with the least on-pathway growth despite having the most total growth because of its greatly elevated rate of off-pathway growth.

These results suggest a trade-off between rate and fidelity of assembly for each of the three parameters examined. In each case, adjusting the parameter in question to produce more total growth also leads to a higher incidence of malformations. For energy and concentration, the result is that intermediate values ultimately produce the most on-pathway growth, avoiding the pitfalls of too little growth or too many malformations. While the base simulation yields maximal on-pathway growth for the tolerance experiments, it also shows the general pattern of higher growth rates coinciding with higher malformation rates. The increase in malformation rate, however, overwhelms the gain in total growth even for the double-concentration simulation. Together these results suggest that optimizing correct growth requires striking a balance between two competing obstacles.

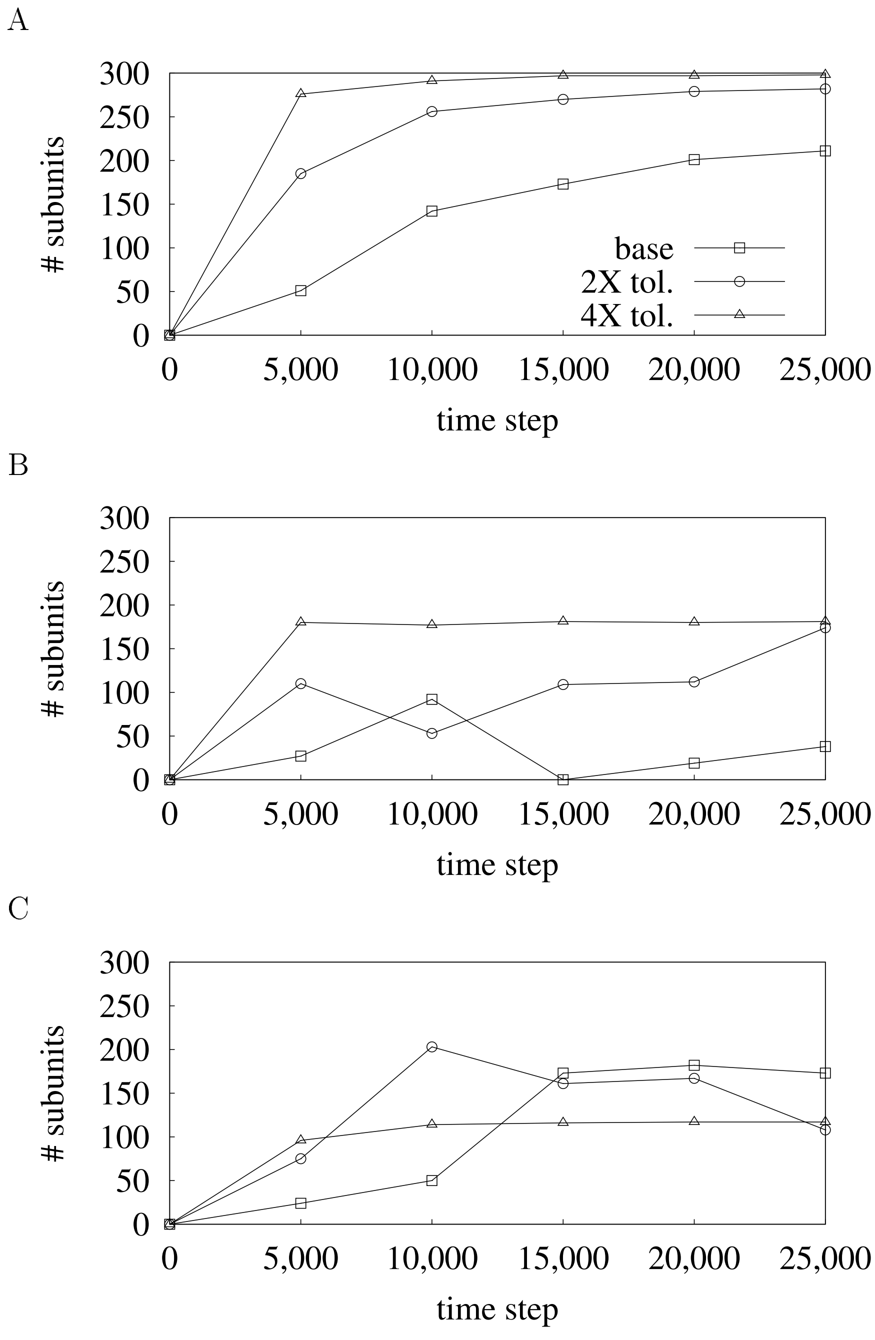

Figure 3-3: Graphs of growth as a function of binding tolerance for a simple model of T=1 assembly. The graphs show the numbers of subunits involved in nucleated clusters for A: total growth; B: off-pathway growth; and C: on-pathway growth for the base simulation and simulations in which translational binding tolerances are adjusted by two times and four times relative to the base.

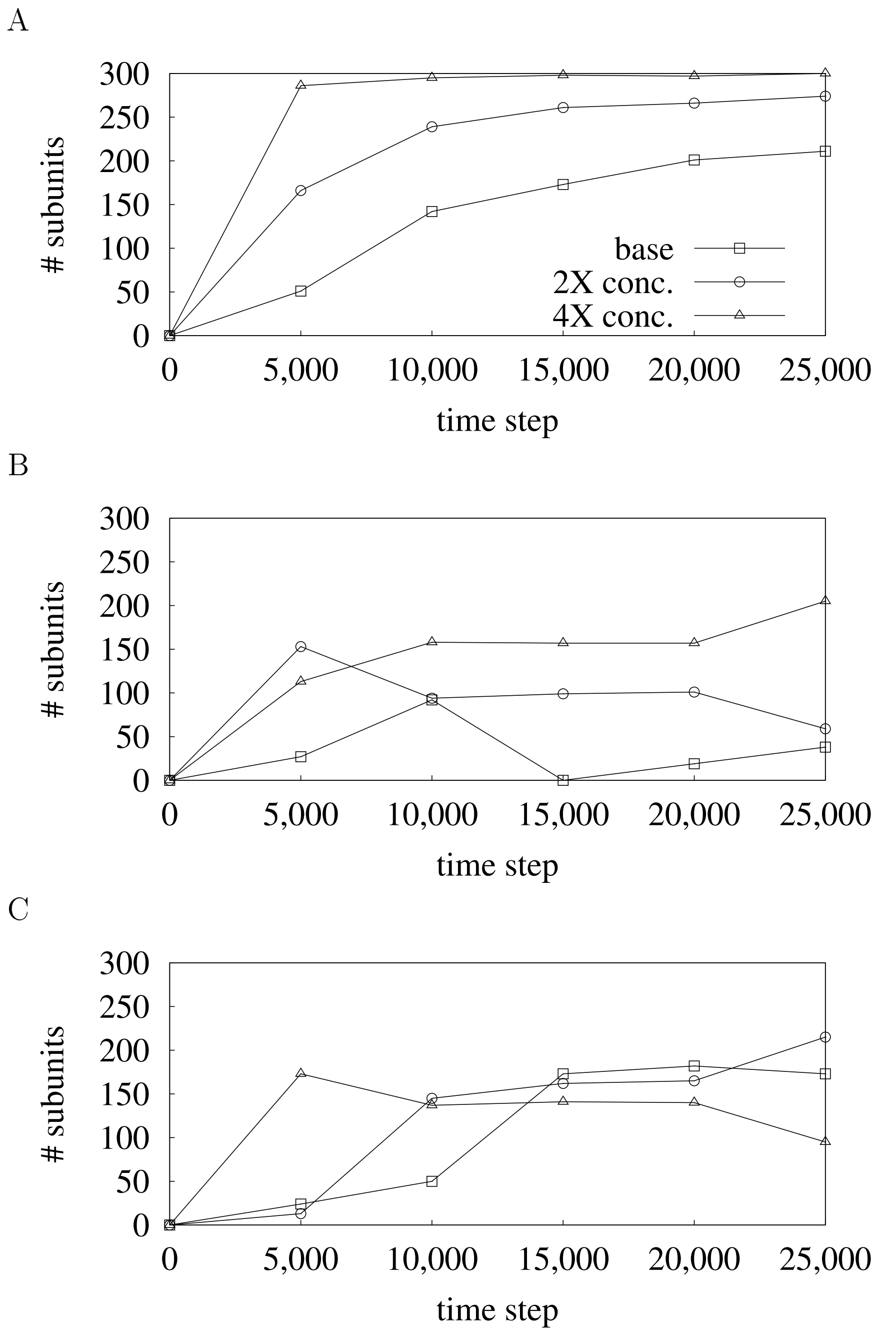

Figure 3-4: Graphs of growth as a function of concentration for a simple model of T=1 assembly. The graphs show the numbers of subunits involved in nucleated clusters for A: total growth; B: off-pathway growth; and C: on-pathway growth for the base simulation and simulations in which concentrations are adjusted by two times and four times relative to the base by decreasing simulation volume while holding the number of assembly subunits constant.

The result suggests some potentially counter-intuitive strategies for disrupting viral growth. If viruses are in fact optimized for a trade-off between the two factors of rate and fidelity of assembly, then introducing changes into the system that tend to push virus shells in either direction would be expected to reduce the yield of correctly formed shells. While it may be obvious that adding an agent that inhibits growth might reduce the yield of correctly formed shells, our results suggest that an agent that promotes growth might also be an effective anti-viral. Thus, we propose that a molecule that catalyzes the nucleation of shells or the addition of new subunits might be an effective anti-viral agent, with the particles it causes to be lost to non-viable shells more than offsetting the higher rate of shell production it induces.

In addition to the main results, the experiments provide some insight into why parameter variations affect off-pathway growth rates within the simulation model. The simulations reveal that parameter variations can affect the incidence of malformed structures by influencing the probabilities of local errors in binding patterns, the rate of collisions between growing clusters, or the probabilities of errors of either kind spontaneously self-correcting after forming. Figure 3-5 shows examples of malformations induced through local errors and collisions of clusters. We expect that increasing binding energy would influence these primarily by stabilizing malformations that do occur and preventing self-correction. Increased tolerance should make local errors more common, as well as making inter-shell interactions during collisions more likely. Increasing concentration should act primarily by increasing the frequency of collisions between incompletely formed clusters. This kind of insight into the specific mechanisms of malformations and how different parameters are likely to influence them might also be useful in designing anti-viral strategies that promote malformations.

These experiments also illustrate how simulation in general can be useful to this kind of study. The particular experiments run rely on the abilities to adjust precisely specific parameters of a model system and to collect detailed data on the state of that system over time. Neither of these capabilities is generally available in actual physical systems. While concentration can be easily and precisely adjusted in practice, neither binding energy nor binding tolerance can similarly be altered with high precision and

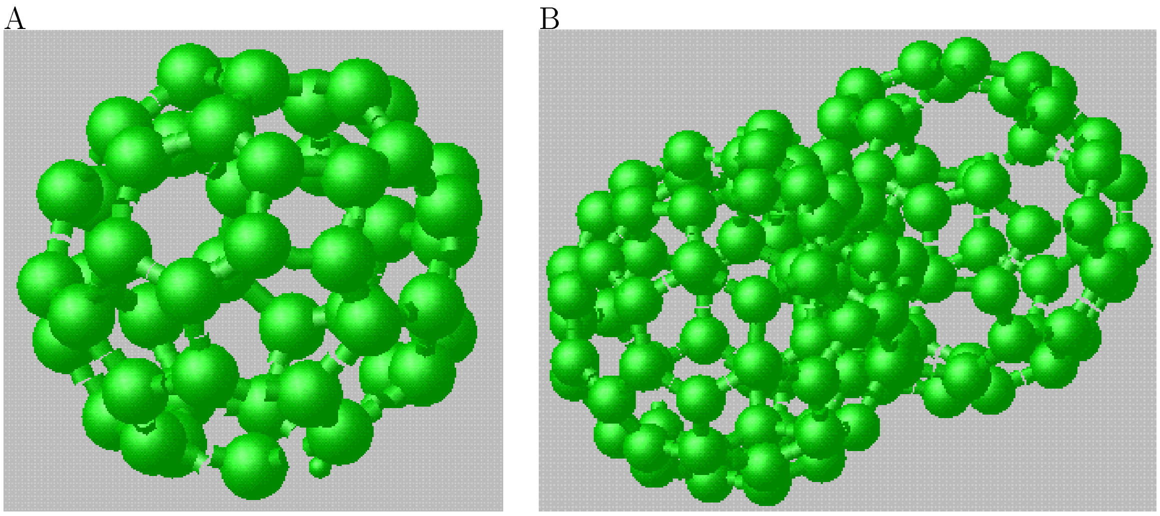

Figure 3-5: Screen snapshots of malformed clusters of particles produced by A: a local error in which particles assumed a tetramer form where they should have formed a pentamer; and B: a collision between two growing clusters that attached to one another.

in isolation from other factors for an actual virus system. Similarly, it is generally impossible to examine the full state of a microscopic system such as a virus self-assembly reaction or to halt it for such an examination at a specific time and restart it later. Halting a simulation, examining its state, and restarting it as if it had never been halted, however, are trivial for a simulated system. This application thus illustrates how simulation can complement laboratory experimentation, providing a means to perform valuable experiments that may be difficult or impossible to perform on an actual physical system.

3.3 Comparing Assembly Models

Another important role for simulation tools is comparing possible models for the underlying cause of an observed behavior. This section describes two sets of experiments comparing the source of a behavior called nucleation-limited growth. The section first explains the concept of nucleation-limited growth and describes three models proposed to explain it. It then describes two sets of experiments developed based on quantitative theoretical models of the different qualitative assembly models.

The first set of experiments supports an assembly model using an inactive conformation, but only ambiguously. The second set of experiments provides firmer support for a particular model called "autostery" [13], which is explained later in the section. Together the experiments illustrate the value of simulation in general and the local rules dynamics model in particular for evaluating theoretical models. Both sets of experiments were performed in collaboration with Bonnie Berger and Peter Prevelige, Jr.

Using simulations to compare competing theoretical models can fill in an important gap when experimental work proves unable to resolve between those models. Simulations can make it possible to test the validity of theoretical predictions that are be based on a significantly greater level of abstraction than that used by the simulation framework. In addition, they provide a means to discover properties of the theoretical models that might not have been obvious from purely theoretical work. These benefits can be significant for self-assembly systems, in which complicated overall behaviors can arise from fairly simple interactions.

One application of the simulator along these lines is exploring the source of nucleation-limited growth of icosahedral virus shells. Nucleation-limited growth refers to growth in which shells assemble by a slow "nucleation" step, in which an initial nucleus is formed from free assembly subunits, followed by a relatively rapid "growth" phase, in which additional subunits add on to the growing shell. By contrast, growth might lack a distinct nucleation phase, with total growing time dominated by the addition of individual subunits to a developing structure. Nucleation-limited growth has been detected experimentally in the icosahedral virus shell of P22 [85]. Experimental approaches, however, have so far proven unable to decide between different possible models for its source.

Some models have been proposed for how nucleation-limited growth can occur. Caspar [13] considers two possible sources for nucleation-limited behavior. One possibility is a variant of a model developed by Oosawa and Kasai [79] to explain nucleation-limited growth in helical self-assembling structures. The basis of Oosawa and Kasai's model is the observation that when a helical structure first initiates but has not yet completed one turn, it has fewer binding interactions per subunit than when it has completed a turn and subunits can interact with those above and below them in the helix. As a result, there is less stabilizing enthalpy per subunit before the first turn is formed and progressively more thereafter. The configurational entropy penalty of fixing a subunit in place is the same, however, regardless of when it adds to the helix. Thus, a growing helix might become stable only after it had passed some threshold number of subunits, creating nucleation-limited behavior. Caspar [13] discusses how the concept might generalize to non-helical systems. We will subsequently refer to this model as the "configurational entropy" model.

A second possibility offered by Caspar is the autostery model [13]. Autostery is named by analogy to the biological process called allostery, in which proteins influence behaviors of other proteins by binding to them and inducing conformational shifts. Autostery proposes that proteins can induce such shifts in other chemically identical proteins as they incorporate into a growing structure. An autostery-based self-assembly system can be built from proteins that occupy active and inactive conformations. If an inactive protein collides with an active protein, the active protein binds to the inactive protein and catalyzes its transition to an active state. The formerly inactive protein is then free to recruit other inactive proteins into a growing structure. Slow nucleation is produced because growth can only initiate if some protein spontaneously transitions to an active state. Because the initial active protein can catalyze the transitions of other proteins, which can then catalyze the transitions of others, and so on, no further spontaneous transitions are required. Thus, if the spontaneous transition is a slow event, the end result is relatively slow nucleation and rapid subsequent growth.

A variant on this model can also explain nucleation-limited growth using separate active and inactive conformations, but without catalysis. In this model, proteins switch freely between the two conformations. Active proteins can bind only to other active proteins, however. In order to give nucleation-limited behavior, this model requires that structures below some certain size are unstable and generally fall apart if not completed quickly. The result is that nucleation requires several proteins colliding while in the active conformation to produce a stable nucleus. If subsequent additions can produce stable structures while adding fewer proteins than are needed in nucleation, then it becomes comparatively more likely that the needed proteins can arrive in the active conformations in time to advance shell growth. Thus, the rate of nucleation will have a higher order dependence than the rate of subsequent growth on the probability of subunits being in the active conformation, promoting nucleation-limited growth. We refer to this model below as the "free switching" model.

It has thus far proven impossible to resolve which, if any, of these models actually accounts for nucleation-limited growth of icosahedral virus shells. Experimental techniques have not been able to provide sufficient evidence to eliminate any of the models for virus shell assembly. Simulations cannot definitively answer this question either. Simulations may, however, provide arguments for favoring some models over others.

We have attempted to examine the relative merits of the models by creating theoretical predictions of their performance and evaluating these predictions through simulations of the models. Our simulations are based on the assumption that a virus that is optimized for a high yield of complete shells will favor a high growth rate relative to nucleation rate. Slow growth relative to nucleation should yield many incomplete shells. In addition, as we note in section 3.2, there is a risk of malformations due to interactions of incomplete shells; shells that complete quickly once begun should have a relatively reduced risk of such malformations. We therefore examine theoretical predictions regarding which models are most effective at promoting high growth relative to nucleation rate and evaluate those theoretical predictions by simulation.

Initial Application

An initial experiment was attempted to compare two different models of nucleation limited growth. These two models corresponded to what is labelled "free switching," and "configurational entropy" above. As a control for the free switching model, a simulation altering concentration is added as well. This experiment is covered in more detail in Schwartz *et al.* [96].

We developed theoretical predictions based on the models. The theory is based on the assumption that pentameric nucleation is followed by subsequent addition of monomers. Four simulation models are considered:

1. a simulation using free switching, with the active conformation occurring a fraction, p, of the time.

2. a simulation using a single conformation with $\frac{1}{p}$ times the volume of model one, yielding a fraction p of the concentration

3. a simulation using configurational penalties with binding tolerances set to reduce binding probability by a factor of p relative to model one

4. a simulation using configurational penalties with binding tolerances set to reduce binding probability by a factor of $\sqrt{p}$ relative to model one.

In order to study how effective each model is at promoting nucleation-limited growth, we examine how the entropy penalties of conformational switching and configurational constraints should act on nucleation and monomer addition. The first model is expected to yield a factor of approximately p^5 reduction in nucleation rate, because nucleation requires bringing together five subunits in the active conformation in a short period of time. Subsequent monomer additions should be slowed by a factor of p, because each requires only that the new subunit adding to a growing shell happens to be active. The third model is expected to slow nucleation by a factor of p^4 and the fourth model by p^2 because each requires four binding interactions to fix all five subunits completely in a pentameric nucleus; the fifth interaction adds enthalpy, but does not impose a configurational entropy penalty. The third and fourth models should slow monomer additions by factors of p and $p^{\frac{1}{2}}$, respectively, because each monomer addition requires paying one configurational entropy penalty. This analysis yields nucleation:growth ratios of p^4, p^3, p^3, and $p^{\frac{3}{2}}$ for the four models. This data is summarized in table 3.1. On the basis of the predictions, we would expect model one to have the highest ratio of growth to nucleation rate of the four models. Model

model	nucleation	growth	nucleation/growth
1	p^5	p	p^4
2	p^5	p^2	p^3
3	p^4	p	p^3
4	p^2	$p^{\frac{1}{2}}$	$p^{\frac{3}{2}}$

Table 3.1: Theoretical nucleation and growth rates and the ratio between them for the four simulation models.

two would be expected to have a similar nucleation rate but slower growth, model three to have a similar growth rate but higher nucleation rate, and model four to have higher rates of both but the lowest ratio between them.

Simulation parameters were determined empirically to give a reasonably high growth rate and a reasonably low malformation rate for model one. T=1 shells are used to simplify the problem and promote more rapid growth. Active subunits occupy a single conformation with binding angles ideal for T=1 assembly. Active subunits are represented by spheres of radius 1 nm and inactive subunits by spheres of radius 0.5 nm; this representation was chosen to make the two conformations visually distinct rather than for any anticipated effect on the results. The probability p from the theoretical analysis is set to $\frac{1}{4}$.

Experiments are conducted using model one as a basis for comparison to the other models. A simulation is first run using model one at increments of 1,000 time steps until it has completed a shell. The number of steps required is used as a baseline for running simulations of the other models. The other models are each run for eight multiples of that base time in order to examine their qualitative behavior.

Model one produces a complete, correctly formed shell at 9,000 time steps. Figure 3-6 shows the simulation at that time. The single completed shell is visible in the lower-left corner of the image. No other growth is apparent.

Model two produces multiple independent nucleations early on, but no complete shells. Figure 3-7 shows the simulation of model two at 9,000 and 72,000 time steps. Figure 3-7A shows that many nucleation events have already occurred as of 9,000 time steps, yielding multiple small clusters of particles but no completed shells. Figure 3-7B shows that these small clusters do eventually coalesce into larger structures, but

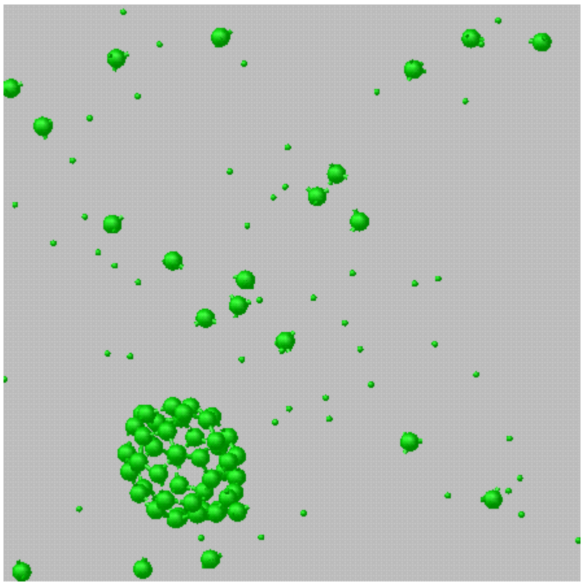

Figure 3-6: A simulation of model one at 9,000 time steps. Smaller spheres represent proteins in the inactive conformation and larger spheres proteins in the active conformation.

none has enough subunits to form a complete shell. In addition, at least one cluster is malformed.

Model three similarly produces multiple nucleations and yields no complete shells, although the effect is less pronounced than with model two. Figure 3-8 shows the simulation of model three at 9,000 and 72,000 time steps. Figure 3-8A shows three distinct partial shells, although two appear to have attached to one another. They separate again later in the simulation. By midway through the simulation a nearly complete shell is visible. As figure 3-8B shows, however, it has still not fully closed by the end of the simulation, apparently due to the fact that free subunits are nearly exhausted even by 9,000 time steps.

Model four produces reasonably good growth, but ultimately fails to produce any correct, complete shells, because most free particles are absorbed into a malformed shell. Figure 3-9 shows the simulation of model three at 9,000 and 72,000 time steps. Figure 3-9A shows that three shells initiate early on. Unlike with model three, some free monomers are still available as of 9,000 time steps. By the end of the simulation,

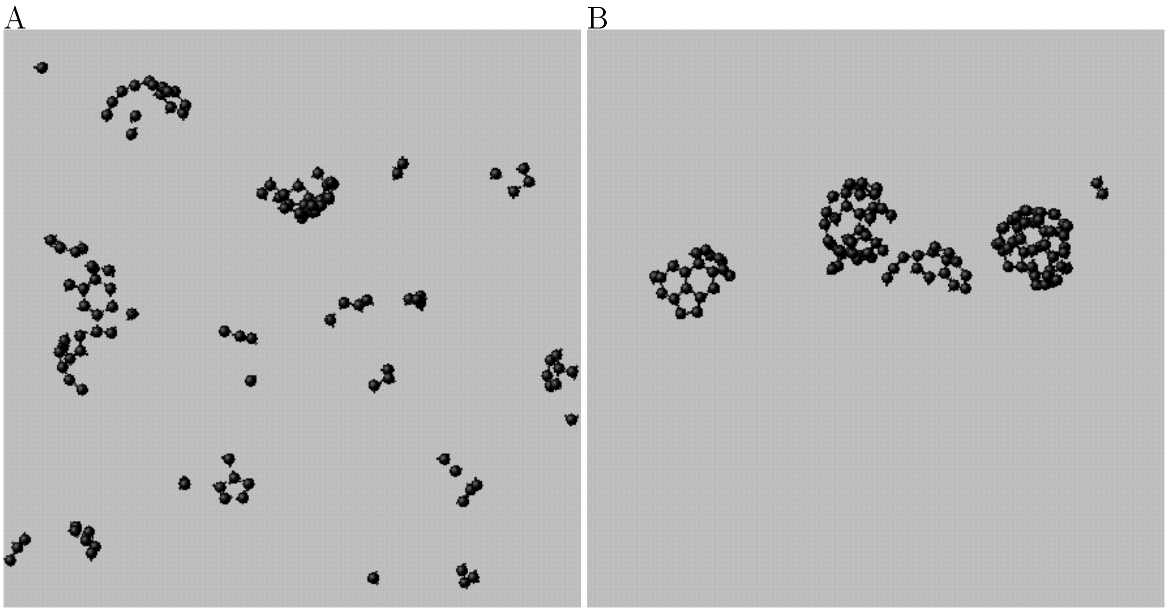

Figure 3-7: A simulation of model two at A: 9,000 and B: 72,000 time steps.

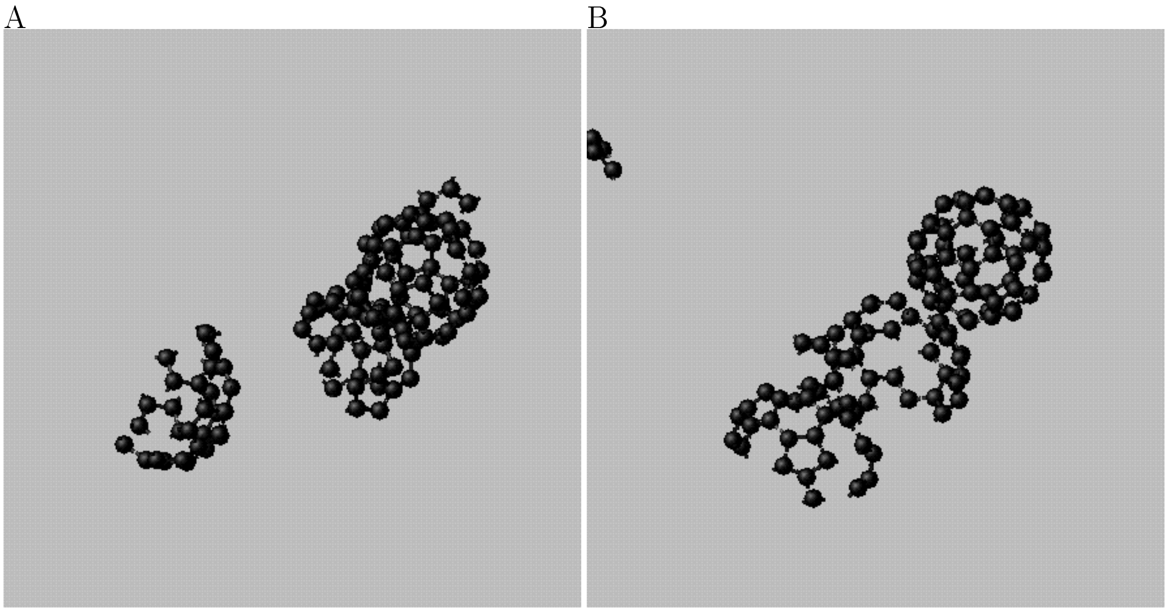

Figure 3-8: A simulation of model three at A: 9,000 and B: 72,000 time steps.

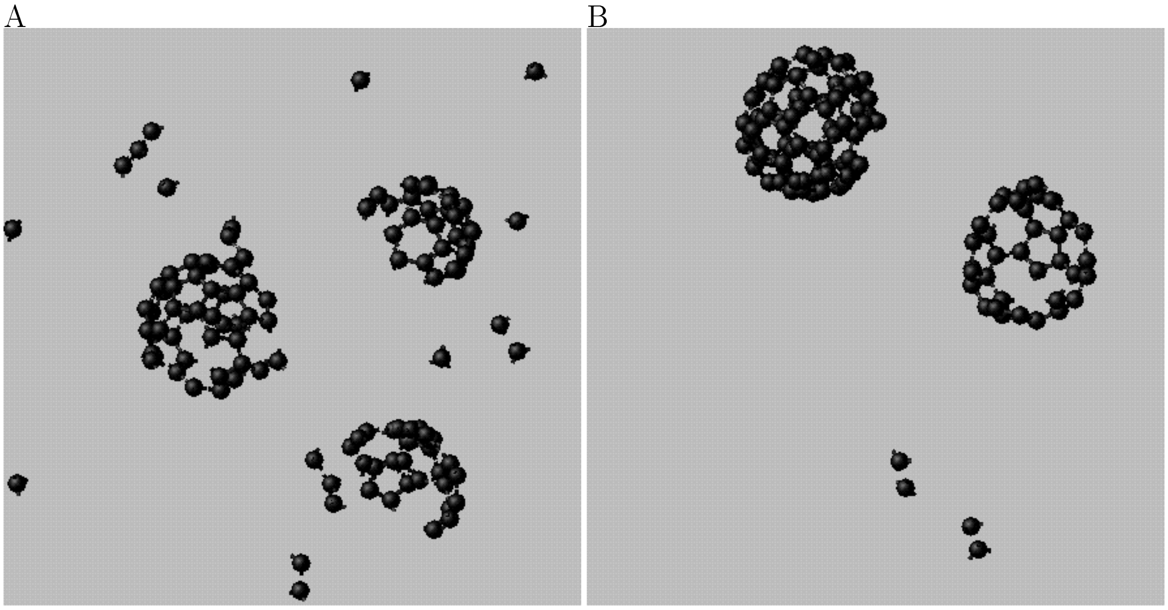

Figure 3-9: A simulation of model four at A: 9,000 and B: 72,000 time steps.

shown in figure 3-9B, however, a malformed cluster of particles visible at the top of the image has absorbed most proteins, essentially blocking further growth of another incomplete structure. The malformed cluster is formed earlier in the simulation when two of the originally three clusters collide and become bound to one another. Unlike with a similar collision in the model three simulation, before the structure can break apart, it becomes trapped in a malformed shape that continues adding subunits.

The results seem consistent with the predictions, although they do not provide conclusive evidence for them. Model two was predicted to have a similar nucleation rate to simulation one but a lower growth rate, which is consistent with the fact that clusters in model two are small early on and grow together only slowly. Furthermore, the multiple nucleations in comparison to the single nucleation for model one can be explained by the hypothesis that model one's higher growth rate produces a rapid drop in free subunit concentration following nucleation, leading to a reduced probability of more nucleations, while models two's slower growth has no such effect. Model three was predicted to have a similar growth rate to model one but a higher nucleation rate, consistent with the observation that model three yields three nucleation events early on, but cannot complete shells because it rapidly exhausts the free subunits. Model four was predicted to have high growth and nucleation rates but a low ratio between them, again consistent with the observation that it produces several partial shells early on but is unable to complete them because free subunits are exhausted. The malformation in model four makes it harder to draw strong conclusions, though, because it is more effective than a correctly-formed shell would be at absorbing free subunits.

While interesting, these early results are unsatisfying for several reasons. The simulations do not include a reasonable representation of autostery. Furthermore, the rapid drop in monomer concentration as shells begin to assemble in each simulation makes it difficult to separate the effects of slow growth due to entropy penalties from that due to the concentration drops. Finally, the strictly qualitative nature of the simulation results makes their interpretation highly subjective.

Revised Application

In order to address concerns about the inconclusive nature of the initial simulations, a new series of simulation experiments was conducted that took better advantage of the full capabilities of our local rules dynamics simulator [97]. These experiments again address a comparison of different assembly models. This time, however, a reasonable model of autostery is implemented. A simulated feedback mechanism maintains concentrations of free monomers by automatically adding more back into the simulation as they become incorporated into growing shells. Finally, the simulation experiment is designed to lend itself to a more rigorous quantitative analysis. In addition, the theory behind the initial predictions is revised in light of more recent experience with local rules dynamics simulations.

The theoretical analysis is revised to take into account the observation that growth following nucleation is not accurately characterized by the monomer addition rate. Earlier simulations reveal that, in order to produce nucleation-limited growth at all, it is necessary to lower binding energies sufficiently that structures smaller than the pentameric nucleus are unstable. Because of this requirement, particles held in place by only a single binding interaction do not bind stably. Growth must proceed via the formation of closed “loops” of particles, each of which is held in place by at least

model	nucleation	monomer addition	overall growth
autostery	p	1	1
free switching	p	$p^{\frac{1}{5}}$	$p^{\frac{4}{5}}$
configurational entropy	p	$p^{\frac{1}{4}}$	p

Table 3.2: Theoretical effects of each of the three models on nucleation rate, monomer addition rate, and overall growth rate for T=1 assembly by monomers, assuming normalized contributions to nucleation rate.

two binding interactions. Under this assumption, the rate-limiting step of growth following nucleation is the formation of the second closed loop of particles after the initial pentamer; this step requires four particles to be added to produce a stable addition. The revised theoretical predictions that result from the assumption that this step controls the growth rate following nucleation appear in table 3.2. Although this prediction is specific to the assumptions of pentameric nucleation and assembly by monomers, we believe a qualitatively similar effect would apply to other reasonable variants on nucleation-limited virus shell assembly. The results suggest that for normalized contributions to nucleation rates, autostery should least impede subsequent growth, free switching should impede it to an intermediate degree, and configurational entropy should most impede subsequent growth.

One major change to the simulation model from earlier work is the addition of a model of autostery. This simulation model relies on two conformations, much like in the previous free switching simulation. Inactive subunits can, however, bind to active subunits, fixing them in place. When an inactive subunit converts to the active conformation while bound, it is correctly positioned to form a second binding interaction with its neighbor, which stabilizes it in the active conformation. The simulations thus mimic the two-state binding process of the theoretical autostery model. One difficulty is finding a set of binding parameters that correctly normalizes the two-step binding of the autostery with the one-step binding of the other models. In particular, it is impossible without some substantially more complicated model to match both the expected time to form a binding interaction and the expected time to break that interaction completely. We decided that both the initial and the stabilizing interactions would have the same energy as interactions in the other models in order to create a similar probability between the models at any given time that an empty binding site is prepared to add an additional subunit. This choice makes interactions in the autostery model considerably more stable than those in other models, however, which may make it more difficult to draw conclusions from its performance relative to the others.

The three candidate mechanisms for this series of simulations are normalized to make approximately equal contributions to the nucleation rate. In each case, an attempt is made to slow nucleation by approximately a factor of 32 relative to what would be expected from the base simulation. For autostery, this means biasing the favorability of the conformations such that free subunits spend a $\frac{1}{32}$ fraction of their time in the active state. For free switching, free subunits are biased to spend one half of their time in the active state. The configurational entropy penalty is induced by reducing the translational tolerance from 10 nm to 6.484 nm. The specific translational tolerance, intended to reduce binding probability by a factor of $2^{(5/4)}$, was derived by interpolation on the results of a series simulations meant determine the dependence of reaction rate on various subunit parameters.

Another significant addition is a model feedback mechanism to hold monomer concentrations approximately constant over the course of the simulation. Every 100 time steps, the simulator calculates the sizes of all clusters of assembly subunits in the simulation. Based on this calculation, the simulator determines how many subunits are not yet involved in nucleated clusters, defined as clusters of at least five connected subunits. If this count of non-nucleated subunits is below a pre-specified constant, set at 90 subunits for the final experiments, then the simulator automatically adds more free subunits to bring it back to the correct number. Thus, the simulator maintains approximately constant free subunit counts over the course of each simulation; this capability is crucial to comparing growth and nucleation rates fairly for the different models.

The specific simulation parameters used were derived empirically from a series of base-line simulations in order to give reasonable rates of growth for all models considered. The resulting values are somewhat different from those used in the earlier series of experiments. All simulations use T=1 subunits in order to simplify the problem and reduce run-time. Active subunits are represented by spheres of radius 1.5 nm with ideal T=1 binding patterns. Where needed, inactive subunits have radii of 1 nm. Both variants have masses of 100 kDa. Inter-subunit binding enthalpies are set to -13 kcal/mol. Base tolerances are 10 nm for distance, 30 degrees for rotation, and 25 degrees for bending. Each simulation is conducted in a cubic workspace of length 110 nm on each side. Data for the final simulations is gathered at multiples of 5,000 time steps for a total of 100,000 time steps. From these data, we compute counts of nucleated clusters and of the total numbers of particles involved in those clusters for the three simulations at each of the 20 time points.

Figure 3-10A shows the numbers of nucleated clusters over time measured for the simulations. The graph shows fairly similar values for the three simulations, as would be predicted from our attempt to normalize nucleation rates, until about midway through the simulations. Later on, the simulations appear to reach "plateaus," with the configurational entropy simulation reaching a higher plateau than the other two.

Figure 3-10B shows the total number of subunits involved in nucleated clusters for the three simulations as a function of time. The graph shows relatively steady growth for all three simulations. Autostery considerably outpaces the other two models. Configurational entropy and free switching have similar values, although configurational entropy slightly outpaces free switching in the latter half of the simulations.

Figure 3-10C shows the average number of subunits per nucleated cluster as a function of time for the three simulations. The data in this graph are the ratio of the data in the second graph to those in the first. The graph shows the highest ratio for autostery, with values generally substantially above those of the other two simulations. Free switching produces generally intermediate values, while configurational entropy produces the least average growth per nucleus. The final gap between autostery and free switching is a factor of 1.81, nearly identical to the 1.80 final ratio between free switching and configurational entropy, contrary to our quantitative predictions of gaps of $32^{(4/5)} = 16$ and $32^{(1/5)} = 2$ based on the theoretical models.

One important qualitative observation from the simulation data is that clusters

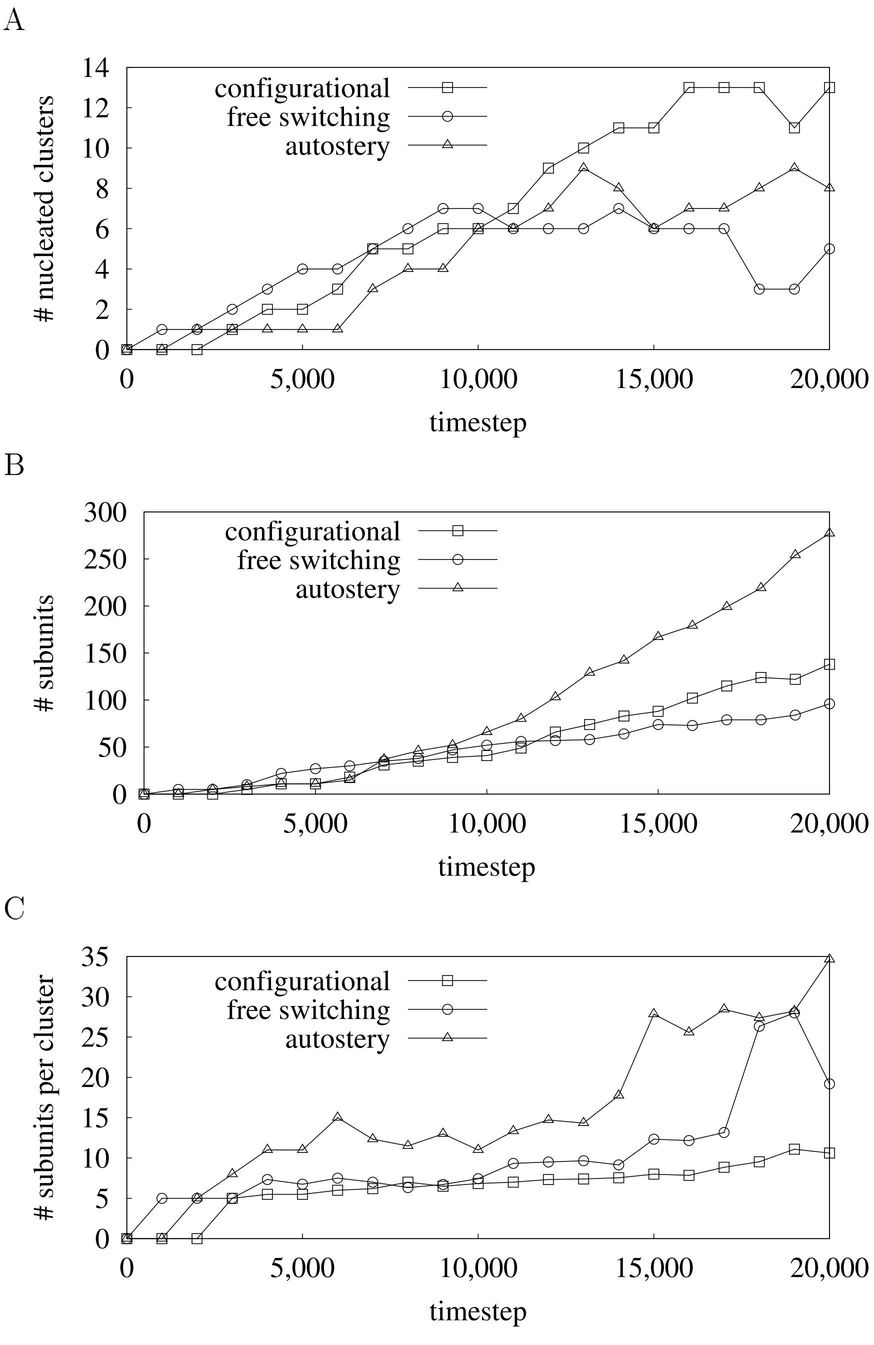

Figure 3-10: Graphs showing nucleation and growth data for the three models of nucleation-limited assembly. A: numbers of nucleated clusters for the three models; B: total numbers of subunits involved in nucleated clusters; C: average number of subunits per cluster, which is equal to the ratio between the data points in part B and the corresponding points in part A.

of particles frequently collide during simulations and bind to one another. This phenomenon appears to be the cause of the plateauing seen in figure 3-10A. These collisions have two important qualitative effects. They seem to promote more nucleation-limited growth, by reducing the number of nucleated clusters on each collision without reducing the number of particles involved in such clusters. They also appear, however, to promote malformations, by occasionally allowing clusters that are not fully compatible to bind and become trapped in aberrant forms.

We conclude that the quantitative data partially support the theoretical predictions. The data match the prediction that autostery should produce the most nucleation-limited growth, followed by free switching, and finally by configurational entropy. The quantitative gaps between the values do not closely match the predictions, however. One reason for this disparity appears to be that correctly formed shells have a maximum size and their growth slows once they get close to it. Our quantitative predictions of growth rate do not adequately capture how this maximum shell size tends to suppress differences between the models. The other major reason appears to be the plateauing effect, which substantially changed overall kinetics. We can hypothesize why this plateauing effect affects the different models to a different degree. With both autostery and free switching, the entropy penalty of the model is paid before a subunit becomes binding competent. Once bound in a shell, all or most subunits are in the binding state. Thus, when two partial shells collide, their subunits do not typically pay any conformational penalty to binding. Configurational entropy penalties are still paid, however, suggesting that they can reduce the likelihood of binding on a collision. Thus, configurational entropy penalties, but not conformational entropy penalties, appear to protect significantly against binding between nucleated clusters. The configurational entropy simulation therefore experiences this effect to a lesser degree and can reach a higher plateau than the other two models. It is not clear whether this effect is an advantage or disadvantage of the configurational entropy model, since collisions have both beneficial effects in promoting nucleation-limited growth and harmful effects in promoting malformed growth. The effects might be reversed for autostery if we assumed that conformational switching, and not the diffusion-limited collision of subunits with a growing shell, is the rate-limiting step in subunit addition.

In addition to what this work reveals about the biological problem, it also raises some important issues about our simulation model and simulation in general. The simulations demonstrate the value of a versatile, programmable simulator by allowing such features as a programmable feedback mechanism for maintaining concentration. More generally, the application demonstrates the value of simulation for exploring hypothetical models and examining their effects. In this capacity, the simulations not only support some predicted effects based on theory but also expand upon them by showing an unanticipated effect — that of the models on the rates of interactions between partially formed shells — that might not have been discovered if the models had not been observed in simulations. The application therefore demonstrates the value of simulation as a complement to pure theory, allowing an insight not otherwise available into how theoretical predictions might succeed or fail in practice.

3.4 Disrupting Growth

This section describes an application of local rules dynamics to the development of anti-viral drug strategies. It first describes a model for disrupting shell assembly using "poisoned subunits" with incorrect local rules. It then discusses an experiment to evaluate the effectiveness of the strategy in a model system with varying concentrations of poisoned subunits. It concludes that the strategy is effective at disrupting growth even when only low concentrations of poisoned subunits are present.

Designing and testing strategies for disrupting the assembly of virus shells is one of the key goals behind the development of local rules dynamics. It is hoped that such strategies could help guide drug development by suggesting general ideas for anti-viral drug design. Local rules dynamics simulations can help by determining whether a proposed abstract particle has certain properties that would be necessary to an effective anti-viral. One particular prerequisite for an anti-viral drug is that it must be able to reduce significantly the yield of correctly formed, viable shells even if the potential drug can be introduced only at low concentrations into an infected cell.

One possible strategy that arose from earlier static local rules work is the use of a "poisoned subunit" that attaches to a shell but does not follow its local rules correctly. In the earlier simulations, it was found that even a single such subunit could potentially disrupt the local geometry sufficiently to produce global malformations of a sort that would be expected to render an actual shell non-viable. Local rules dynamics provides a means to further explore the design of a poisoned subunit. Due to the abstract nature of simulations and our subunit model, we can only consider how the local rules of such a particle might be altered. Applying a poisoned subunit strategy in an actual experimental system would require determining how a real subunit obeying a particular distorted rule set might be created.

In order to test this strategy, I refine a model system of T=1 assembly and compare its normal behavior to that achieved in the presence of low concentrations of a poisoned subunit. The basic system uses a two-conformation "free switching" model. Each simulation uses a fixed quantity of 300 initially unbound subunits in a cubic workspace 120 nm on each side. Normal subunits each have two conformations of mass 100 kDa, with radius 1.0 nm in the inactive conformation and 1.5 nm in the active conformation. Inter-subunit binding energies are all -15 kcal/mole with distance tolerances of 7.5 nm and rotational and bending tolerances of 30 degrees. Conformational energies are set so that the subunits spend one quarter of their time in the active conformation when free in solution. A poisoned subunit is created by using a single conformation identical to the active conformation of the normal subunit except that one bond angle is shifted by 90 degrees outward from the inner surface of the subunit. Simulations are run with 0, 5, 10, 15, 20, 25, and 30 poisoned subunits for 250,000 time steps per simulation.

Figure 3-11 shows the quantitative results. The graph shows the final number of subunits involved in on-pathway and off-pathway nucleated clusters as a function of the number of poisoned subunits. Clusters of connected subunits are defined as nucleated if they contain at least five subunits. A cluster is considered on-pathway if its connectivity pattern is consistent with some subset of a completed shell, even if

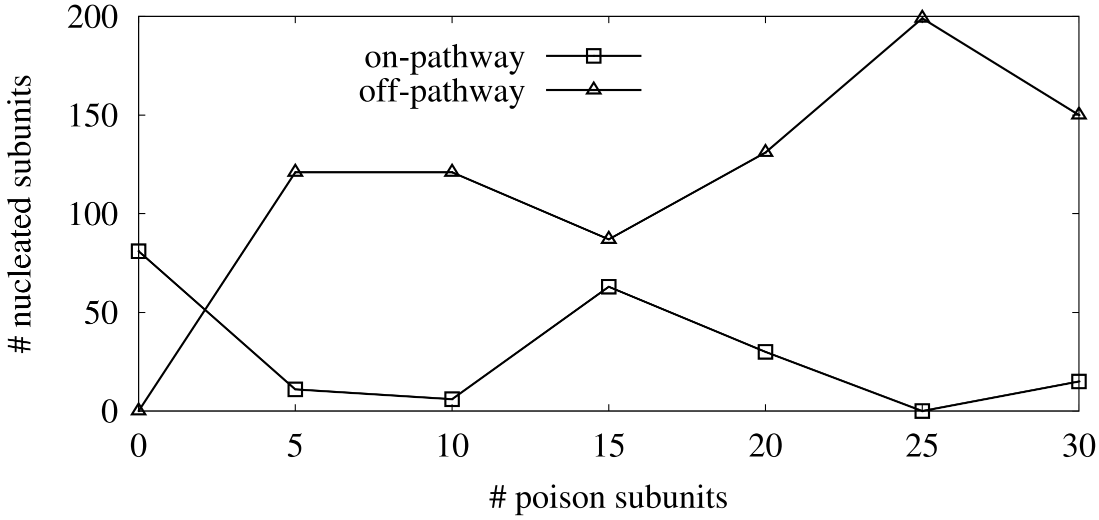

Figure 3-11: Graph of on-pathway and off-pathway growth as a function of the number of poisoned subunits present during the simulation.

the cluster contains one or more poisoned subunits. A shell whose connectivity pattern is inconsistent with any subset of a completed shell is defined as off-pathway. As the graph shows, the simulation with no poisoned subunits produces no off-pathway growth. All simulations in the presence of poisoned subunits produce less on-pathway growth than the simulation free of poisoned subunits and produce at least some off-pathway growth. There is no clear relationship between the number of poisoned subunits introduced and the degree to which growth is disrupted, however. The results nonetheless suggest that the poisoned subunits successfully disrupt shell growth and lead to reduced yields of on-pathway growth even when introduced at low concentrations.

A qualitative analysis confirms this finding. Figure 3-12 compares the final states of the simulation conducted in the absence of poisoned subunits to that of the simulation conducted in the presence of just five poisoned subunits. While figure 3-12A shows the production of a correct shell and progress toward others, figure 3-12B shows only a single large malformed cluster produced through distorted poisoned subunit bonds and a small still-correct cluster that has nonetheless incorporated a poisoned subunit. Of the seven simulations, only that conducted in the absence of poisoned

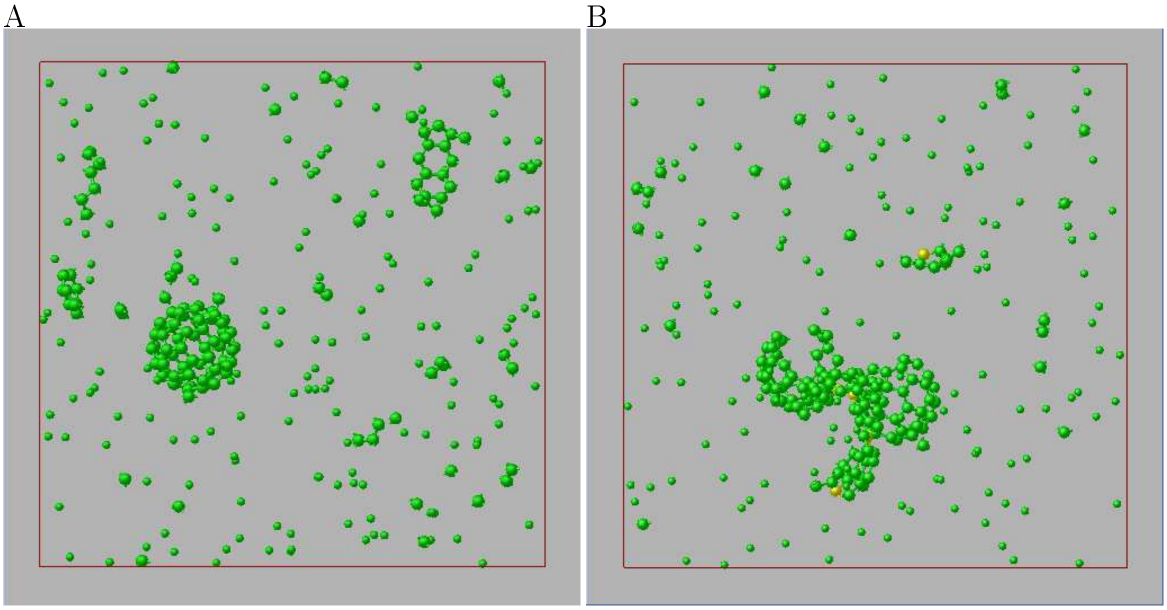

Figure 3-12: Screen snapshots of the end results of simulations illustrating the effects of growing with and without poisoned subunits. A: a simulation containing no poisoned subunits; B: a simulation containing five poisoned subunits.

subunits produces any correct, fully-formed shells in the course of the simulations. Furthermore, almost all geometrically correct assemblies observed in the other simulations have incorporated at least one poisoned subunit, suggesting that they would be unlikely ever to be able to complete correctly, even if allowed more time to assemble.

Although the work in this section includes only a simple experiment, it does suggest at least the theoretical possibility of shell assembly-targeted anti-viral agents being constructed using the poisoned subunit idea. The results demonstrate that even a small number of poisoned subunits can disrupt shell growth within a simulation. By binding to growing shells and producing malformations, the poisoned subunits can cause many otherwise viable subunits to be wasted in non-viable shells. The result, as the quantitative data shows, is a substantial reduction in the yield of correctly forming shells.

3.5 Non-Virus Systems

Although the local rules dynamics model was originally designed to study the specific problem of virus shell assembly, it can be applied to other systems that exhibit similar properties. This section explores four different types of systems that may be amenable to the simulation techniques we have developed: actin, borosilicate glasses, protein aggregation, and a novel self-assembling system. Each subsection presents a sample application showing how the simulator can model aspects of the system in question and thus potentially provide new insights into its behavior. Together, the applications illustrate the versatility of the local rules dynamics model.

3.5.1 Actin

This section describes a simple model system based on the helical polymer actin. The section explains some aspects of actin behavior and describes a simple simulation model based on them. It then provides evidence that the model produces actin-like structures and exhibits equilibrium kinetics qualitatively similar to those of actin. The section concludes that local rules dynamics can capture some aspects of a well-characterized assembly system, although it is still far from being able to produce a detailed, realistic model.

Although virus protein shell assembly is currently poorly characterized with regards to low-level issues of assembly dynamics, much more is known about some other self-assembly systems. A well-understood system presents special challenges in that a simulation model should be able to incorporate those known aspects of the system that influence the behaviors being modeled. Exploring how local rules dynamics models can be adapted and refined in response to actual data on low-level assembly dynamics is crucial to applying it to systems better understood than virus shell assembly and to keeping it up to date as knowledge about viruses and other self-assembly systems increases. Actin is an important example of a well-characterized biological self-assembly system, which makes it a good choice for exploring the capabilities and limitations of local rules dynamics in modeling such systems.

Actin is a key structural protein in eukaryotes. It is found in nearly all eukaryotic cells and is also a central component of muscles. In its functional state, actin forms helical polymers, which have the appearance of two right-handed helices twisting around one another [52]. These polymers are made of many copies of the actin protein. Actin is capable of spontaneous self-assembly from free proteins (called G-actin) to the helical form (called F-actin). Each F-actin protein also binds one copy of the energy-storing molecules ATP or ADP. In its normal assembly process, actin undergoes a conformational switch while bound in the helix, triggered by hydrolysis of ATP to ADP and subsequent release of a phosphate ion. The conformational switch alters the binding properties of actin, reducing affinity for other actin proteins in the helix. The result is that at normal physiological conditions, actin undergoes a process called "treadmilling" in which ATP-actin molecules continually bind to one end of the helix (called the barbed end) while ADP-actin molecules tend to dissociate from the other end of the helix (called the pointed end), causing length to increase continually at the barbed end and decrease at the pointed end [99]. The conformational switch also produces a transient state binding ADP and inorganic phosphate between the ATP- and ADP-binding actin monomers [59]. Actin can also grow and shrink by breaking within the helix or annealing separate helices together [57]. Because association rates are concentration-dependent, however, altering the concentration of the ADP- or ATP-binding forms of actin can cause the helix to polymerize or depolymerize at either end.

I have created a simplified model of some aspects of actin assembly dynamics in order to examine how local rules dynamics can apply to a well-characterized system. I examine only ATP-actin, using only a single kind of binding interaction to specify the bonds between two ATP-actin subunits. My goal is to capture in a qualitative sense the competition between on- and off-rates for this single type of binding interaction. Specifically, I want to create a model for actin assembly that accurately captures the fact that at a specific concentration, actin has a fixed and known ratio of on-rate to off-rate at a given binding site. This application is meant as an example of how to approximate this kind of comparative quantitative behavior within a local rules dynamics model without sacrificing the computational speed that the local rules abstractions bring.

The first step in this process is arbitrarily normalizing concentration. Although I could have created an actin model at a realistic physiological concentration, the dynamics would have been too slow for a reasonable simulation. I therefore select a particular mapping of simulation concentration to real concentration and simulation time to real time. I use 60 particles in a workspace 100 nm on each side to represent a concentration of 1 μM. At a real concentration of 1 μM, actin has on approximate on-rate of 5 sec^{-1} and an approximate off-rate of 1 sec^{-1}[100]. I empirically choose bond parameters that give reasonable run times, settling on a distance tolerance of 7.5 nm and angle tolerances of 30 degrees each. I then run 1,000 trials of a simulation testing the time for a single particle to form a bond with any other particle in a pool of 75 particles in a workspace 50 nm on each side (giving a simulation concentration of 10 μM). In order to test only the diffusion-limited time to form a single binding interaction, I use only one binding site per particle and set its association and dissociation energies to -100 kcal/mole, making binding highly stable. All other subunit and binding interaction parameters are identical to those of the model actin subunit. The average time to form a binding interaction in these trials is 1,878 steps. This measured time yields a mapping of 93,900 time steps per second of simulated time.

I then choose bond energies directly to give a desired off-rate that is one fifth the empirically calculated on-rate at the concentration used. For simplicity, I use identical activation and deactivation energies. Because actin polymerizes cooperatively, with each new addition to a growing helix stabilized by two bonds, the off-rate in (time steps)$^{-1}$ is given in terms of the energy by the formula $(1+e^{-E/k_BT})^{-4}$. Inverting this formula gives an energy of -5.01 kcal/mole. Combining the mathematically chosen energies with the other empirically chosen parameters gives a complete model of an ATP-actin subunit with a known ratio of on-rate to off-rate at a given concentration. This strategy therefore provides a simple demonstration of how such dependencies as a specific ratio of one assembly parameter to another can be satisfied within a local rules framework by a combination of empirical parameter determination and direct calculation.

Simulations using my model for ATP-actin particles exhibit qualitative features of actin assembly dynamics. Figure 3-13A shows the status of a simulation of two hundred initially unbound actin particles in a cubic workspace 150 nm on each side at 250,000 time steps. The simulation shows the spontaneous polymerization of small actin fibers within a pool of free subunits. Figure 3-13B shows a close-up view of a single actin fiber within the simulation. The fiber shows the characteristic two-helix turn of actin. In addition, the binding sites at the ends of the actin are visible, showing how cooperative binding works in the model to stabilize those particles that bind to a nucleated fiber with two binding interactions each when they first join the fiber. Cooperative binding provides greater stability to the attachment of a subunit to a growing fiber than to the attachment of two previously free subunits. Figure 3-14 shows a plot of the number of binding interactions formed at multiples of 1,000 time steps over the course of the simulation. The graph shows that the model has reached an equilibrium state by about 50,000 time steps after which the total number of binding interactions remains roughly constant over time, although interactions are continually being formed and broken. The graph therefore displays the basic kinetic character of a polymer such as actin that establishes an equilibrium between its free and its polymerized states at physiological concentrations. Once the equilibrium is established, individual particles rapidly exchange between the ends of filaments and the population of free particles in solution. The overall number of free particles, however, remains close to the molecule's critical concentration, the concentration below which there is no net polymerization of individual fibers.

This application still provides a crude model of the underlying physical system, demonstrating some of the current limitations of local rules dynamics even while it shows some of its potential. I have modeled only ATP-actin because the time scale of conformational shifting is longer than that of subunit addition. Modeling both time scales would require using short time steps to capture the shorter time scale phenomena but a long time period, and thus many steps, to capture the longer time scale phenomena. The generally prohibitive computational cost of simulating many

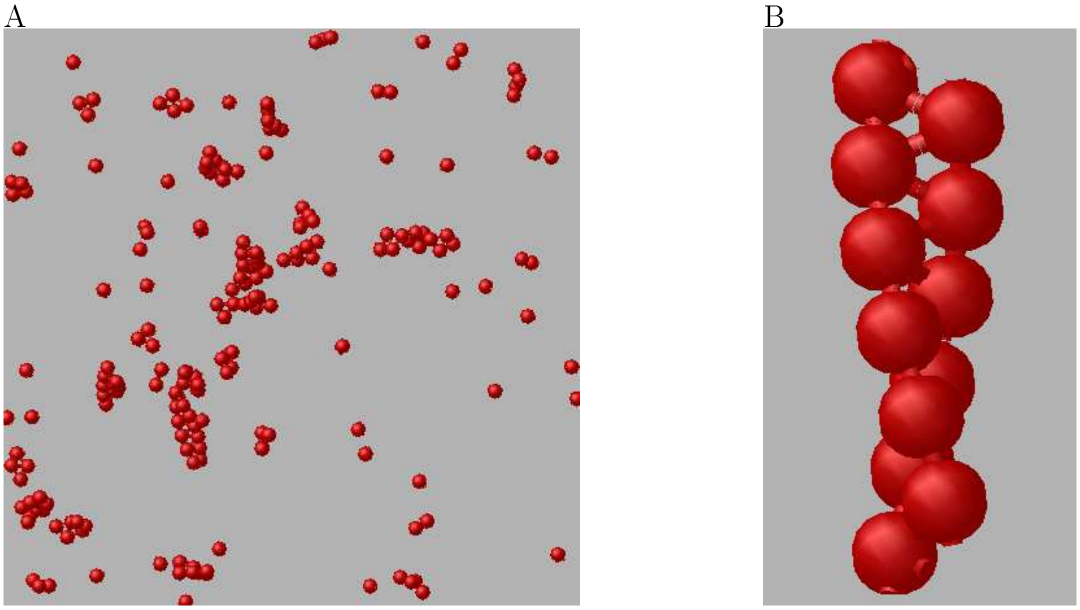

Figure 3-13: Screen snapshots of an actin assembly simulation. A: the full simulation; B: a close-up view of a small actin fiber.

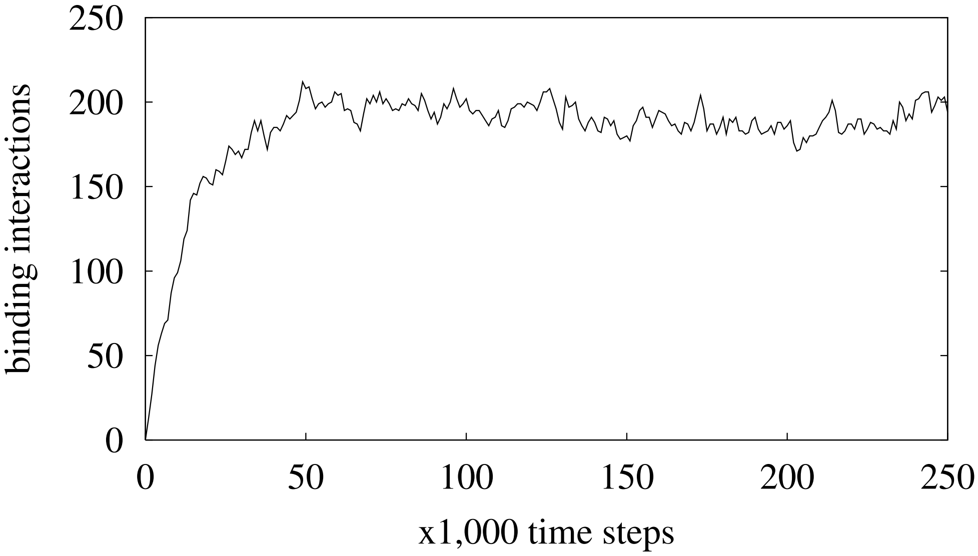

Figure 3-14: Graph of the number of binding interactions formed as a function of time for the actin simulation illustrated in figure 3-13.

time steps makes it difficult to capture processes that occur on dramatically different time scales within a local rules dynamics framework. In addition, there is much more to actin behavior than single-strand polymerization. Actin strands may interact with other molecules and may join into higher-level structures with other actin strands, behaviors that are also amenable to local rules-based modeling. Capturing some of these more complicated phenomena would be essential to making a local rules-based actin model that is useful for current research into actin, but would also require much more computational power. Nonetheless, the ability of our simple model to incorporate approximately some experimentally determined aspects of actin assembly dynamics — such as a known ratio of two kinetic rates — does demonstrate how a local rules dynamics model can simulate some aspects of quantitative experimental data from a system it models in order to create an increasingly detailed and versatile representation as the system becomes better characterized experimentally.

3.5.2 Borosilicate Glasses

This section describes an application of local rules dynamics to modeling the response of borosilicate glasses to radiation damage. The section first describes the system modeled and the motivation behind modeling it. It then describes an atomic-scale local rules dynamics model of borosilicate glasses. The section concludes that the local rules dynamics model exhibits important properties of actual borosilicate glasses, supporting the view that local rules dynamics can extend the range of phenomena that can be modeled relevant to such systems.

The structure and behavior of borosilicate glasses is another important problem to which local rules work can be applied. Borosilicate glasses have attracted considerable interest in recent years because of their possible use as a storage material for nuclear waste. They are considered good candidates for this task because they have been demonstrated in actual waste disposal and because it is believed that they will be easy to process, adaptable to different waste compositions, and resistant to radiation effects [118]. Their usefulness in this capacity is, however, considered uncertain because it is not yet fully known how the glasses will respond at the atomic level to long-term radiation exposure, or how changes at the molecular level will affect their overall structural properties [118].

Because it is not possible to conduct experiments on the time scales that would be required for nuclear waste storage, computer simulation is an important tool in the study of silica glasses and their responses to radiation. Considerable work has already been done using molecular dynamics methods to explore how borosilicate and other glass structures rearrange in response to irradiation [28, 18, 19, 22, 20, 21, 23, 23, 24]. These approaches are, however, limited to looking at small local changes over short time scales and therefore cannot directly answer questions about changes in macromolecular properties over periods of tens to hundreds of thousands of years, as the application requires.

Local rules have also already been successfully applied to studying several related systems. A local rules-based simulator adapted from Berger and Muir's earlier virus assembly simulator [77] has been used to study properties of several crystalline and glass structures [45, 50, 51]. One promising approach based on local rules uses a maximal matching algorithm to optimize bond configurations. Under this approach, the set of available bonding sites is treated as a set of vertices in a graph. The graph contains an edge between any pair of vertices whose corresponding bonding sites are capable of bonding to one another (meaning that they belong to different carbon-silicon tetramers and are within allowed angle and distance tolerances of one another). Weights are assigned to edges based how close the corresponding bond would be to the optimal bond length. A maximal matching algorithm then selects the global bond configuration maximizing the sum of the individual bond weights. This basic routine is incorporated into an overall program in which bonds are broken in a local region after which the program repeatedly forms a maximal bond matching, creates the bonds, minimizes stresses in the structure, re-breaks the bonds, and repeats. The result is a process that appears empirically to converge on a low-stress bonding arrangement without the under-bonding problems of earlier approaches [44].

Some preliminary work has been conducted on applying the local rules dynamics model to this system. Local rules dynamics provides an alternative means of explor-

```
(set_constant "temperature" current_temp)

(define one_cycle (procedure (x) (sequence
 (do_one_update_new true)
 (set current_temp (+ min_temp (* (- current_temp min_temp) temp_scale)))
 (print (strcat "Set current temp to " (num2str current_temp) "K\n"))
 (set_constant "temperature" current_temp))))

(define runsim (procedure ()
 (iterate one_cycle 1 num_temp_steps)))
```

Figure 3-15: Code for implementing a cooling schedule for a borosilicate simulation. The code reduces temperature from an initial value of `current_temp` by geometric decay to approach a value of `min_temp`, by multiplying the amount by which the current temperature exceeds `min_temp` by `temp_scale` for each of `num_temp_steps` steps. The code is written in the simulator control language, which is documented in appendix A. The simulation is activated by calling the procedure `runsim`.

ing the process of bond rearrangement, allowing particles to form and break bonds and rearrange themselves during simulations of local regions of glasses responding to radiation damage. This capability may yield a method closer to the actual physics of the process than other prevailing approaches and may therefore provide more reliable data.

A local rules model was developed for a particular model system: a borosilicate glass consisting of boron, silicon, and oxygen atoms. The system is meant to model how a small region of a borosilicate glass reconstitutes itself if its atomic bonds are disrupted by radiation damage. The system was developed using realistic values for atomic radii, atomic masses, bond lengths, and bond energies in order to provide a more reasonable model of the competing effects of different kinds of interactions. In addition, the simulation rules file incorporates code for implementing a "cooling schedule" by which the simulation is cooled from a temperature of 5,000 K to a temperature of 298 K by a series of small discrete steps approximating a realistic exponential temperature decay. The code appears in figure 3-15.

A simulation of 600 oxygen, 150 boron, and 150 silicon atoms contained in a workspace 50 angstroms in each dimension is conducted to verify that the model can

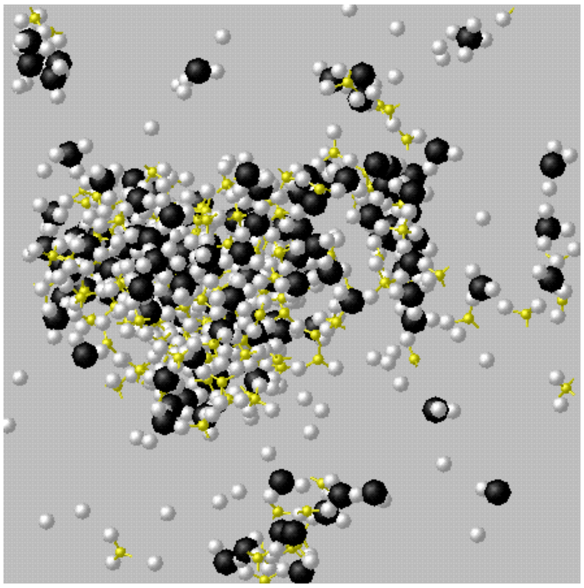

Figure 3-16: A screen snapshot of a 900 atom borosilicate glass produced through local rules simulation. The large spheres represent silicon, the medium-sized spheres oxygen, and the smallest spheres boron.

produce reasonable structures. The cooling schedule uses an initial temperature of 5,000 K, an asymptotic limit of 298 K, and a scaling factor of 0.95 on each cooling step, for a total of one hundred cooling steps. Prior to each cooling, the simulation is allowed to run for one hundred time steps. The resulting structure appears in figure 3-16. The picture shows that the atoms condense considerably from their initial distribution to a single large clump, some smaller clumps, and several free atoms or small clusters. Based on smaller simulations, I expect that running the simulation longer would lead to further condensing of the structures present.

This particular application is still in an early stage and is presented primarily as an illustration of how local rules dynamics simulations can apply to such systems; considerable additional work will be required before it is likely to be useful in understanding such systems. Because the actual atomic level structure of this form of amorphous borosilicate glass is not known, it is impossible to evaluate directly how well the simulated structure models the actual system. Various statistical properties of the formed structures might, however, eventually be matched to experimental observation of actual systems in order to evaluate how well the simulation models actual borosilicate structure and dynamics.

3.5.3 Protein Aggregation

This section describes a local rules-dynamics model for protein aggregation. The section first explains the nature and importance of the protein aggregation problem. It then describes the construction of a local rules-based model for protein folding and aggregation, which it illustrates through a simple model system. Finally, it describes an experiment showing that the model exhibits an important property of actual aggregates called kinetic irreversibility.

Protein aggregation is a phenomena in which proteins in solution come together, typically while folding, to form generally irregular clusters of multiple proteins [61]. Aggregation is important to the bio-technology industry because it is a common outcome of cloned gene expression [69, 16] and thus a significant obstacle to the mass production of purified proteins. In addition, aggregation has been implicated in several human diseases, including Alzheimer's disease and light-chain amyloidosis [119]. Aggregation is similar to other protein self-assembly in that it involves many, typically chemically identical proteins coming together to form a larger structure. Unlike with virus assembly, though, the structures formed by aggregation generally lack any discernible symmetry or regularity. This makes aggregation particularly difficult to analyze experimentally, as experimental techniques for determining atomic structures generally depend on finding repeating or symmetric units. Several simulation models for aggregation phenomena have been developed, including statistical models [17], highly abstract lattice Monte-Carlo simulation models [122, 81, 82], and lower-level sequence-based methods [38, 49]. These approaches, however, have been limited in their abilities to study the dynamic nature of the process.

Local rules simulation can provide a mechanism for exploring the dynamic process of formation of protein aggregates. Local rules dynamics may be a particularly useful technique for exploring protein aggregation precisely because protein aggregates are so irregularly structured; unlike some higher-level methods, local rules dynamics does not require structures it builds to exhibit any overall symmetry, provided local interactions between structural elements follow local rules-based binding patterns. Moreover, local rules dynamics makes no assumptions about specific pathways to formation, which may be important given evidence that aggregates can build by combinations of multimers of any size, and not merely by addition of free monomers to established growing sites [103].

In order to adapt protein aggregation to the local rules dynamics framework, it is necessary first to create a local rules-based model of the folding of individual proteins. The model I use is similar to both the diffusion-collision model [54, 55], which proposes that proteins fold by forming "microdomains" that then must slowly diffuse and collide with other microdomains in order to coalesce into a final structure, and the building-block folding model [116, 113, 112], which proposes that folding occurs via building-blocks that may shift between different conformations during assembly. Tsai *et al.* [112] discuss how the building-block model may specifically apply to aggregation. We can recast the assumption that a protein can be treated as a small number of microdomains or building-blocks that must find an optimal packing during folding into a local rules dynamics model by viewing a protein as a chain of domains, each of which contains some set of binding sites, which model the energetics created by packing against other domains in certain ways. The use of conformational shifting in the local rules dynamics model makes it possible for a domain to shift its binding patterns dynamically over time. For instance, a domain might shift between an alpha helix and a beta strand form (with different preferences for other domains), only one of which is found in the final folded protein. The conformational switching capability, however, is not used in the model system we discuss here.

From this model of single-protein folding, a local rules dynamics model of multi-protein aggregation follows directly. Specifically, the same forces that hold domains of a protein together as it folds can hold together distinct proteins that have collided with one another. Provided proteins have the ability to form interactions with multiple neighbors, this model allows for large aggregates to form, held together by the same interaction forces that drive folding of a single protein. The conformational switching ability also allows for proteins to exhibit interaction types not found in the single-protein native structure. Thus, given an understanding of the local interactions holding an aggregate together, regardless of whether or not they match those in the native structure, local rules dynamics provides a model for exploring both the irregular aggregate structure and the process by which it forms.

In order to demonstrate the potential of the local rules dynamics model for protein folding and aggregation, I have created a simple model system. The system consists of a simulation of model proteins, each made of three domains meant to represent alpha helices joined together by two linker regions meant to represent turns. Each of these five segments is represented by a distinct node in the simulation. The overall protein is created by adding all five nodes in the correct relative positions and joining them by highly stable bonds, with association and dissociation energies of -100 kcal/mole. In addition to these preset bonds, the three helix portions each have two binding sites capable of joining to the other two helices with binding energies of -13 kcal/mole. While the intra-chain bonds have tolerances of 1 nm for translation and 5 degrees for rotation and bending, the binding sites joining separate helices have tolerances of 10 nm for translation and 30 degrees for rotation and bending. The correctly folded native form implied by this binding pattern models a three-helix bundle in which each of the three helices connects to the other two. Figure 3-17 shows a fully folded bundle.

Creating a simulation of many such model proteins allows us to examine the competition between folding and aggregation. Figure 3-18A shows a small aggregate produced by the collision of three proteins during the folding reaction. The aggregate shows how separate proteins can become tightly bound through "domain swapping" in which each forms a three-helix bundle in which one of its own helices is replaced by a helix from another protein. Figure 3-18B shows a more complicated aggregate of seven proteins.

I conduct simulations to explore the behavior of this model system. I create a simulation of 100 model proteins over the course of 25,000 time steps in a cubic workspace 100 nm on each side. Proteins are initially added with random positions and orien-

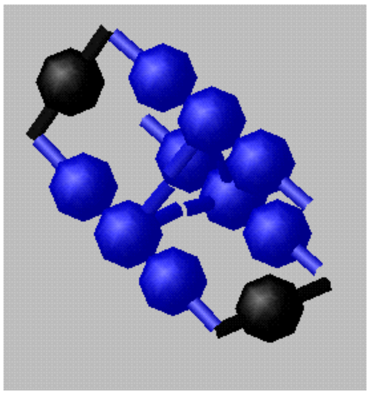

Figure 3-17: A model of a three helix bundle. The helix regions are represented by three spheres each. Turn regions linking helices within the chain are represented by single spheres.

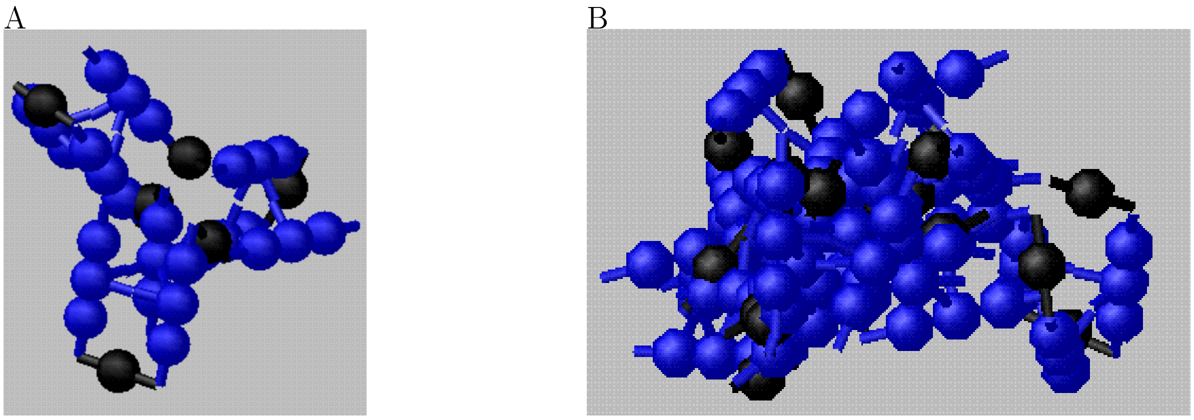

Figure 3-18: Screen snapshots of protein aggregates produced by folding multiple model proteins simultaneously in a simulated solution. A: an aggregate of three proteins; B: an aggregate of seven proteins.

tations and are in fully unfolded, linear forms. Figure 3-19A shows the results of the simulation. The figure shows a single large aggregate of 60 proteins, several smaller aggregates, and only seven correctly folded single proteins. I subsequently increase the workspace size to 200 nm in each dimension and run the simulation for an additional 25,000 time steps. Figure 3-19B shows the results. The distribution of aggregates is essentially unchanged, except that a two-protein aggregate has separated from the 60-protein aggregate; the number of correctly folded proteins is still seven. By contrast, a simulation run under identical conditions to the original 25,000 time step simulation, but with the workspace size set to 200 nm in each dimension from the beginning, produces the results of figure 3-19C. This simulation produces 62 correctly folded monomers from the original 100 proteins. It also yields no aggregates of more than five proteins. The contrast between figures 3-19B and 3-19C demonstrates that the simulation model exhibits an essential feature of protein aggregation: kinetic irreversibility. Aggregated proteins in solution are not generally at a thermodynamic equilibrium, but rather are kinetically trapped in an aggregated state and do not easily detach and refold correctly when their concentration is reduced [69]. By showing that the local rules dynamics model spontaneously exhibits this observed behavior of actual aggregates, the results indicate that this simple model system can be a useful representation of some aspects of the competition between protein folding and aggregation.

3.5.4 Novel Self-Assembly Systems

This section describes an application of local rules dynamics to the development of a novel self-assembly system. It illustrates the applicability of local rules dynamics to such systems by describing the design and refinement of a simulation system meant to model simple self-assembling circuits. It then describes a pair of experiments showing that the assembly system can build different circuits from identical sets of components depending on which of two scaffolding subunits is introduced into the solution.

One hope of the present work is that it can teach us principles that can be applied to the design of artificial self-assembly systems. The idea of using self-assembly as a

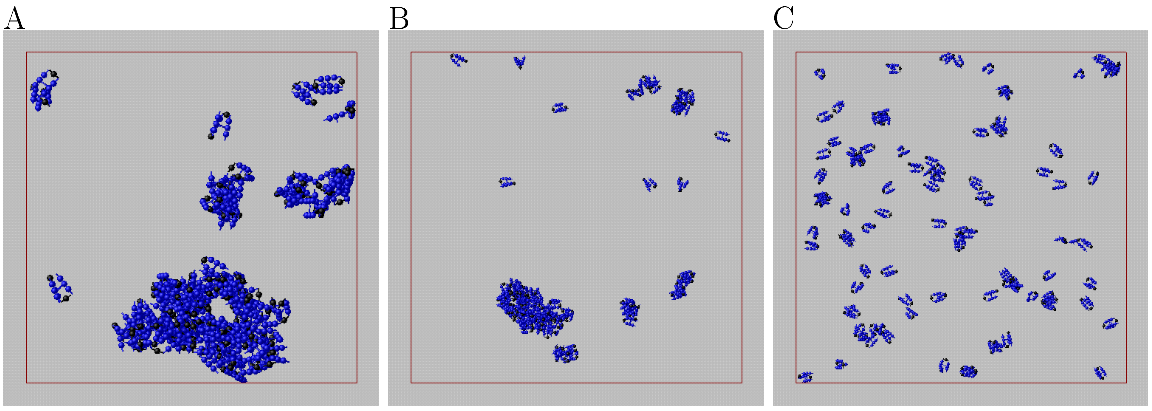

Figure 3-19: Simulations of the results of allowing one hundred model proteins to fold in the same solution. A: a simulation conducted in a cube 100 nm on each side over 25,000 time steps; B: the simulation of A after volume is increased eight times and the simulation run for an additional 25,000 time steps; C: a simulation conducted by beginning with a workspace 200 nm in each dimension and running for 25,000 time steps.

micro-fabrication tool was proposed by Whitesides [120]. It has since been applied to a variety of materials and structures. Experience with virus assembly can help us by suggesting techniques that can be used in building such systems and by motivating the development of tools, such as the local rules dynamics simulator described in this thesis, that have broader applications. In order to explore such issues, I create a simple model system meant to illustrate both how local rules dynamics simulation might help in developing novel self-assembly systems and how an understanding of natural self-assembly systems might suggest ideas useful to the design of artificial systems.

My intention was to create a model system of self-assembling circuit elements. I use two basic element types: "gates" meant to represent elements that each have one or more inputs and produce some output and a "wire" element that is meant to transmit a single input to its output. The binding sites of these elements are designed to bond strongly to one another but to have strict tolerances; as a result, there is a strong kinetic barrier to the assembly of structures, produced by a configurational entropy penalty. In order to produce structures quickly and reliably, the elements must be brought together via a "scaffolding" subunit that guides assembly and removes ambiguities in the structural form, but is ultimately removed from the assembled structure. The scaffolding subunit thus acts like a catalyst to promote inter-element bonds. Each element type therefore has an extra binding site for attachment to the scaffold. These less specific scaffold bonds act over longer distances to bring circuit elements together in specific patterns. The key advantage of this system is that by transferring most of the "intelligence" of the self-assembly system to the scaffold, we allow simple, versatile elements that would likely be easy to construct to build the final structures, while reusable scaffolds control behavior specific to a single circuit design. Thus, the same cheap circuit elements could be used in many different kinds of self-assembling circuits while comparatively few expensive scaffolds would be needed for the production of specific circuits. This idea was inspired primarily by the P2-P4 virus system [70, 101, 1], in which a single coat protein produces different shell geometries depending on which scaffolding proteins are transiently associated with it during assembly.

Although this model is highly abstract, we can speculate on how such a system might be physically constructed. The strong short-range bonds that hold elements together in a completed structure might be covalent bonds, which are strong but typically require that specific atoms be brought into close proximity with one another. The longer-range, weaker bonds that bring elements to the scaffold might then be ionic bonds or hydrophobic interactions, which are typically much weaker than covalent bonds but can also be far less specific with regards to the configurational constraints of binding. Different binding specificities might be imposed on the hydrophobic interactions or ionic bonds via different spatial patterns of hydrophobic and hydrophilic patches or of positive and negative charges. Washing a scaffold free of a completed circuit without disrupting the bonds between circuit elements might be accomplished by adding a salt to the solution in the case of ionic bonds or adding a hydrophobic solvent in the case of hydrophobic interactions. Either would weaken the scaffold bonds without affecting covalent inter-element bonds. Separation of completed circuits from freed scaffold and unused circuit elements could then be accomplished with standard mass-based separation techniques, allowing the completed circuits to be removed for use in larger structures while scaffold and unused circuit elements are added back to

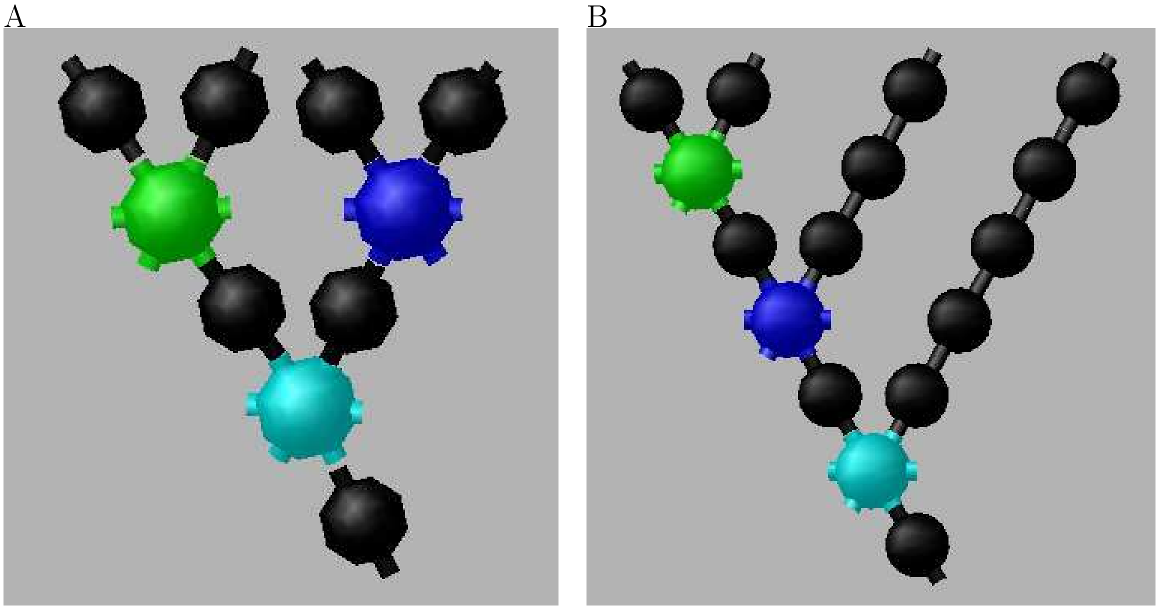

Figure 3-20: Two circuits designed to be produced by the assembly of gate and wire elements on different scaffolds.

the solution for another round of fabrication.

I designed a simple experiment to test this principle of local rules-based scaffold-directed nano-assembly. I first designed a set of three gate types and one wire type. Each gate has six binding sites arranged in a plane (although only three are used for this experiment) and one scaffold binding site perpendicular to the plane. The wire element has two binding sites pointing in opposite directions and one scaffold binding site perpendicular to them. I created two scaffold subunits designed to build two different circuits from the same element types. Figure 3-20 shows the target circuits. If we imagine the three gates implementing binary functions $f(x,y)$, $g(x,y)$, and $h(x,y)$, then the first circuit will implement the function $h(f(x_1,x_2),g(x_3,x_4))$ and the second will implement the function $h(g(f(x_1,x_2),x_3),x_4)$. I then set out to refine the system to produce either desired circuit from the same set of circuit elements depending on which scaffold subunit is present.

I ultimately arrive at a system capable of producing completed circuits of either type from a fixed pool of components depending on the type of scaffolding subunit used. Simulations of both assembly reactions run in the presence of a single scaffold subunit and an excess of all other components: five each of the three gate elements and 50 wire elements. Inter-element bonds all have enthalpies of -100 kcal/mole and tolerances of 0.1 nm for translation and 5 degrees for rotation and bending. Scaffold bonds prior to washing have enthalpies of -20 kcal/mole and distance tolerances of 10 nm for translation and 30 degrees each for rotation and bending. Wires and gates each have a mass of 100 kDa while the scaffolds have masses of 100 kDa per element they are capable of incorporating. Wires are represented by individual spheres of radius 1.5 nm and gates by spheres of radius 1.75 nm. Simulations are run at temperatures of 298 K for 500,000 time steps for each of the assembly and washing phases. Test runs using the final parameter sets for each of the two scaffolds each produce a single complete circuit of the desired type.

The simulations are not yet perfect, however. Figure 3-21A shows a scaffold-circuit element complex produced by a simulation of the system with the second scaffold subunit. The complex most likely appears because a partially assembled circuit partially detaches from the scaffold prior to the washing phase, allowing a second circuit to begin assembling in the place of the detached portion of the first. Although a complete circuit is assembled, partially formed circuit pieces can also be seen in figure 3-21B, which shows the final state of this simulation. The nature of the error here suggests that we might achieve higher fidelity by increasing the strength of scaffold bonds, thus preventing the kind of partial detaching observed in this application. The error and our analysis of it thus illustrate how local rules dynamics simulations can be useful in spotting problems and developing strategies to resolve them.

Future work can improve and expand upon such systems. For example, simple assembled circuits of the kind presented here can become the basic subunits of further local rules-based assembly reactions that result in more complicated molecular machines. Thus, a series of reactions, each with a scaffold-attachment and a washing phase, could produce much more complicated self-assembling machines than would be feasible from a single reaction.

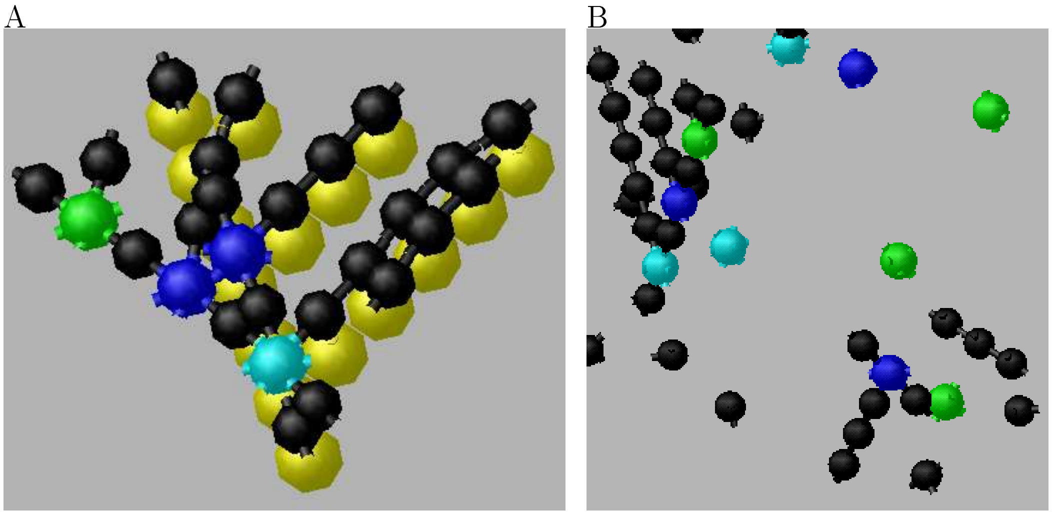

Figure 3-21: Screen snapshots showing an error during use of the second scaffold subunit. A: a complex produced when an assembling circuit partially detaches from the scaffold during assembly, allowing additional polymerization on the scaffold; B: the final simulation state, showing both the correctly formed complete circuit and partial circuits produced by the additional polymerization.

Chapter 4

Physical Model

The first task in implementing a simulator based on the local rules dynamics model is filling in the details of the physical model implemented. It is necessary to create a model that has sufficient versatility and a sufficiently detailed conception of some key processes underlying self-assembly dynamics that it is a useful simulation model. The model, however, also has to be constructed with computational efficiency in mind. The local rules dynamics model places constraints on the overall simulation model by implying a basic model of binding interactions and solution dynamics. Other factors that are included in the simulation model for the present purposes include some extensions to local rules-based binding interactions to allow for more realistic binding kinetics, a notion of conformational shifting needed to capture the role of multi-conformation proteins in viral self-assembly, and a model of Brownian motion intended to maintain realistic thermodynamic behavior over time.

The remainder of this chapter describes the physical model in detail. It first explains the implementation of individual assembly subunits, typically representing proteins. Next, it describes their means of interacting with one another: elastic collisions or binding. Finally, it discusses how the simulator combines multiple subunits into a thermodynamically reasonable model of a solution of interacting particles by implementing a model of Brownian motion.

4.1 Protein Model

This section describes the basic assembly subunits of the physical model, which generally represent viral coat proteins. The section explains a design by which subunits can be built with different shapes and binding configurations. In addition, it presents a model of conformational switching allowing all or part of a subunit to shift its shape or binding configuration over time.

The basic unit of simulations is a model protein, referred to henceforth as a node, particle, or subunit. The structure of a node is defined hierarchically in terms of three node types: single, variable, and complex. Single nodes are the basic unit of node construction; each consists of a single sphere with a characteristic mass and radius and a specified binding pattern. Variable nodes implement conformational switching. A variable node is defined by two or more sub-node types with characteristic energies. It shifts between the types according to the energies in a manner described below. Complex nodes allow for the creation of different shapes of nodes, built through unions of spheres. They are defined by two or more sub-nodes with given positions and orientations, which are treated as if they are rigidly connected to one another. Both variable and complex node types can have single, variable, or complex subtypes, allowing for possibilities such as complex nodes with both stable and variable domains or for variable nodes with complicated shapes. The different node types are illustrated in figure 4-1.

Conformational switching of subunits is controlled by a model of energetics. Each conformation of a variable node is assigned an activation and a deactivation energy. The activation energy determines the likelihood of entering that conformation on a specific time step provided the subunit has just left another conformation, while the deactivation energy gives the likelihood of leaving the conformation on a given time step provided the particle is in it at the beginning of the time step. Separate activation and deactivation energies allow the user to control both thermodynamic equilibriums and kinetic rates of conformational switching. Given a particle with conformations $C_1, \ldots, C_N$, with associated activation energies $E_{a1}, \ldots, E_{aN}$ and deactivation ener-

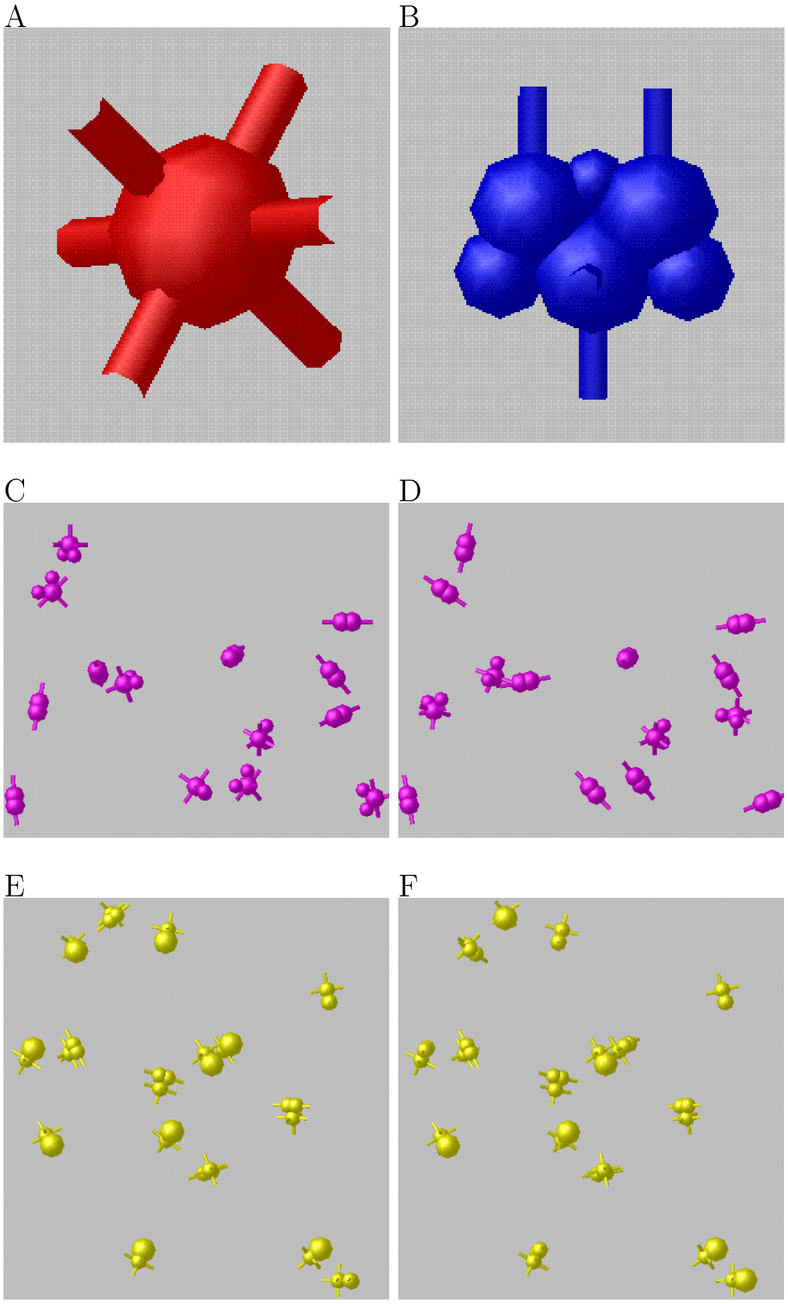

Figure 4-1: An illustration of the different allowed node types of the simulator. A: a single node consisting of a sphere with a specified binding pattern; B: a complex node consisting of five single nodes, some of which have their own binding patterns; C,D: variable nodes at two consecutive time steps, each conformation of which is itself a complex node; E,F: complex nodes each of which has a variable and a non-variable region, shown at two consecutive time steps.

gies $E_{d1}, \ldots, E_{dN}$, we can analyze that particle's kinetic and thermodynamic behavior by representing it as a Markov chain in which the transition probability from any state i to another state j is given by

$$\begin{aligned} p_{ij} &= (e^{(-E_{di}/k_BT)}+1)^{-1}\frac{e^{(-E_a i/k_BT)}}{\sum_j e^{(-E_a j/k_BT)}}, i \neq j \ , \\ p_{ii} &= 1-\sum_{j\neq i} p_{ij} \ . \end{aligned}$$

On the basis of this Markov model, we can derive some kinetic and thermodynamic results. The expected number of time steps spent in a state C_i after initially assuming it, which provides a measure of the kinetic stability of a conformation, is

$$s_i = (1-p_{ii})^{-1} \ ,$$

which expands to

$$s_i = \left(e^{(-E_{di}/k_BT)}+1\right)\frac{\sum_j e^{(-E_a j/k_BT)}}{\sum_{j\neq i} e^{(-E_a j/k_BT)}} \ .$$

The expected proportion of time spent in C_i, which is a measure of its thermodynamic favorability, is

$$k_i = \frac{\left(\sum_{j\neq i} p_{ji}\right) / \left(\sum_{j\neq i} p_{ij}\right)}{\sum_k \left(\sum_{j\neq k} p_{jk}\right) / \left(\sum_{j\neq k} p_{kj}\right)} \ .$$

In this model, the switching rate between conformations is a function, in part, of the time step. Energies cannot therefore be considered to be equivalent to thermodynamic equilibrium energies, but rather must be calculated for a given step size to give desired kinetic and thermodynamic behavior.

4.2 Binding Interactions

This section describes a model of local rules-based binding interactions. It first explains the model, which combines local rule specificities with a notion of binding energies. It further analyzes the model and derives several kinetic and thermodynamic properties of the binding interactions from the energy model. Finally, the section explains the mechanical forces, produced through a mass-spring model, by which binding interactions affect subunit behavior.

Subunits assemble into larger structures by forming binding interactions with other subunits using a model of local rules-based bonds. Each simulated subunit has some pattern of binding sites, also called edges. Each edge has a specified length, a direction relative to the subunit's other edges, and an edge type. The edge type specifies the edge's mechanical strength; its tolerance to errors in positions and angles of binding; its energies, which control its probability of forming and breaking; and a set of other edge types to which edges of the given type can bond. When an edge comes within its distance and angle tolerances of another edge of a compatible type, the two have a probability of binding controlled by their energies, allowing them to form a spring-like attachment between their two subunits, which might break apart at some later time. The details of these interactions are described below.

The model uses a probability system based on a transition state model of reactions. The model assumes that an interaction has some initial and final energies, with possibly some intermediate energy representing a transient state before the full association. The model establishes activation energies for the association and dissociation reactions much like those used for controlling conformational shifting. Each edge type has some characteristic association energy E_a and dissociation energy E_d. While only some pairs of E_a and E_d values are physically reasonable, the model allows for arbitrary choices of values. In addition, formation of binding interactions is controlled by three kinds of tolerances: translational, rotational, and bending. If an interaction is already formed, then the participating edges have a probability of breaking the interaction given by

$$p_d = \left(e^{(-E_d/k_B T)} + 1\right)^{-1} \ .$$

If an edge is not involved in a binding interaction, then it first determines which nearby edges are within the allowed tolerances. Being within the allowed tolerances means that the distance between the endpoints of the edges are within the translational tolerance of each other; that the "up" vectors of the two edges, which are vectors perpendicular to the edges that are used to measure rotation, are within the rotational tolerance of each other; and that the angular difference between the two edges and between each of them and a vector between the centers of their corresponding single nodes are each within the bending tolerance. The full set of edges satisfying all three tolerance conditions for a given edge becomes a set of candidate binding partners. Assuming any such candidates exist, the probability of forming a binding interaction with any particular candidate i from a set E of candidates is given by

$$p_{ai} = \frac{e^{(-E_a/k_B T)}}{|E|e^{(-E_a/k_B T)} + 1} \ .$$

Thus, the probability no interaction is formed is

$$p_{a0} = \left(|E|e^{(-E_a/k_B T)} + 1\right)^{-1} \ .$$

Because binding interactions will change with the movements of particles, it is impossible to predict kinetic or thermodynamic behavior of binding interaction formation without considering other issues such as the tolerances of the bonds and the concentration of all the various subunit types in solution. The binding energies capture only the enthalpy and not the entropy of interactions. We can, however, determine such values in the special case that we have two edges with compatible types held by other interactions in the correct positions for binding, thus implying no entropy penalty to binding. This situation might hold for a single edge pair in a complete portion of a growing virus shell, in which each subunit is also bound to other particles in that shell. In this case, the expected number of time steps required to break a formed bond is given by the inverse of the probability of breaking the interaction and not immediately reforming it:

$$t_d = \left(e^{(-E_d/k_B T)} + 1\right)\left(e^{(-E_a/k_B T)} + 1\right) \ .$$

The equilibrium probability of that bond being formed at any given time is

$$k_i = \frac{\left(e^{(-E_a/k_BT)}+1\right)^2\left(e^{(-E_d/k_BT)}+1\right)^2}{\left(e^{(E_a/k_BT)}+1\right)^2+\left(e^{(-E_a/k_BT)}+1\right)^2\left(e^{(-E_d/k_BT)}+1\right)^2}\ .$$

This model makes it possible to design proteins to meet known kinetic constraints by adjusting a small number of parameters. A reaction is typically characterized by such properties as its entropy and enthalpy, which give rise to on- and off-rates of the binding reaction. The simulator's binding energy values can be considered reaction enthalpies. Adjusting the entropy of the binding interaction is more complicated, as it requires tuning binding tolerances to give desired kinetic constants. It is not *prima facie* clear what tolerance values should correspond a desired entropy, and trial-and-error remains the best method for matching values to particular behaviors. Selecting specific on- and off-rates can be accomplished by choosing an E_a and E_d pair to give the desired dissociation rate for a given step size, then selecting tolerances empirically to give the correct on-rate for those values. In applications, the problem is commonly simplified by setting $E_a = E_d$, which represents the case of no transition state between the unformed and fully-formed bond.

I note that this physical model is only an approximation of a fully realistic model of binding probabilities, even given the constraints on formation of binding interactions. Ideally, the full ensemble of possible binding interactions among all edges in a simulation would simultaneously sample from a Boltzmann distribution of possible states on each time step. The given model will behave correctly only in the case where no edge has more than one candidate binding partner on any given time step. In other cases, we approximate the correct value by choosing the order in which edges are processed at random, but this procedure does not give the same probability distribution as solving the full problem simultaneously. The model described in this section is chosen primarily because of the computational difficulty of solving the more realistic problem efficiently.

When a bond is formed, a mass-spring model controls the forces the binding interactions exert on their associated nodes. The model involves three springs, imposing a translational force and rotational and bending torques on the bound nodes. The translational force is intended to pull the endpoints of two bound edges together. For two single nodes, node 1 and node 2, connected by a binding interaction, the translational force on node 1 is given by

$$\vec{T}_t = k_t(\vec{e}_2 - \vec{e}_1) \ ,$$

where k_t is the translational spring constant, and $\vec{e}_1$ and $\vec{e}_2$ are the end points of the bound edges of nodes 1 and 2, respectively. In addition, if the single nodes are not at the centers of mass of their corresponding nodes, then the force also results in a torque being applied on the nodes given by the cross product of $\vec{T}_t$ and the vector from the center of mass of the node to the center of the bound single node.

The rotational forces act to twist the nodes around bonds in order to align "up" vectors specifying a preferred rotational orientation. For rotational spring constant k_r; unit up vectors $\hat{u}_1$ and $\hat{u}_2$ for the edges of nodes 1 and 2, respectively; and unit vector $\hat{d}_1$ specifying the direction of the edge of node 1, the rotational torque acting on node 1 is given by

$$\vec{O}_r = k_r[(\hat{u}_1 \times \hat{u}_2) \cdot \hat{d}_1]\hat{d}_1 \ .$$

The final springs create a bending torque intended to straighten out bound edges so they will tend to point anti-parallel to one another. If the edges of nodes 1 and 2, respectively, have directions $\hat{d}_1$ and $\hat{d}_2$ and straightening spring constant k_s, then the straightening torque exerted on node 1 is given by

$$\vec{O}_s = k_s\sqrt{0.5 + 0.5(\hat{d}_1 \cdot \hat{d}_2)}\frac{\hat{d}_2 \times \hat{d}_1}{||\hat{d}_2 \times \hat{d}_2||} \ .$$

The forces and torques acting on node 2 can be computed symmetrically to those acting on node 1.

Because the various units of simulations are relative, the units of the spring constants cannot be unambiguously specified. They can, however, be given for abstract distance unit u_d, mass unit u_m, and time unit u_t. For these, k_t has units of $u_m/{u_t}^2$

and k_r and k_s have units of $u_m \cdot {u_d}^2 \cdot radians/{u_t}^2$.

4.3 Collision Interactions

The physical model also supports two kinds of collision forces that can affect subunits. The first of these forces push apart any two subunits that collide, modeling elastic collisions. The second collision forces occur between subunits that collide with an artificial boundary that is created around the simulated solution. These boundary collisions keep subunits within a small user-specified workspace; in the absence of some such constraint, subunits would quickly diffuse, making any kind of assembly simulation impossible. An alternate solution might have been the use of periodic boundaries, but we rejected this alternative primarily due to it greater complexity of implementation. This section provides a detailed explanation of the two kinds of collision forces.

The exact forces used for these interactions were chosen empirically to give reasonable physical behavior while being quickly processed by the numerical methods. To some degree, these are contradictory goals, as a reasonable hard-sphere model necessarily has discontinuities in low-order derivatives, interfering with the convergence of prevailing numerical methods for such problems. One possible solution might have been to use long-distance forces, such as the d^{-2} forces used for collisions in the original local rules simulator of Berger and Muir [77] or the d^{-12} forces commonly used in true molecular dynamics simulations. Either of these would, however, be computationally costly and less realistic for large proteins while still presenting considerable problems for the numerical methods. We therefore decided to have partially elastic collision forces that initiate only when two sphere radii touch or when a sphere touches the edge of the workspace and grow continuously with increasing overlap from zero at the point of first contact.

The force between two spheres on a collision is given in terms of the radii r_1 and r_2 of the two nodes, the vector $\vec{d}$ between their centers, and an empirically selected dimensionless collision constant C that is currently set to 200, by

$$\vec{F}_c = C \min\left\{\left(1 - \frac{d - r_1 - r_2}{r_1 + r_2}\right)\left(\frac{d - r_1 - r_2}{r_1 + r_2}\right)^{-4}, 192\right\}\frac{\vec{d}}{|d|} .$$

The force exerted by the workspace boundary on a sphere that crosses it has a direction inward from the boundary and a magnitude given in terms of the distance d the sphere extends past the boundary, the sphere's mass m, its radius r, and the collision constant C from above, by

$$F_w = \frac{Cm}{r}\min\left\{\frac{r(r-d)}{d^2}, 12\right\} .$$

Although both kinds of forces are purely translational, they can cause a torque on a node if applied to a single node that is not at its parent node's center of mass.

4.4 Thermodynamic Constraints

An additional set of physical constraints are addressed by a model of Brownian motion. This section describes the nature of this model. It first explains the use of a damping force to suppress kinetic energy. It then describes two candidate models for a random perturbation force that complements the damping force by increasing kinetic energy. The section argues for the use of a method intended to minimize randomness in particle paths, thereby accelerating diffusion, over an alternate method that is more realistic and theoretically satisfying.

Brownian motion is a by-product of collisions of molecules or other small particles in solutions that causes them to move around in seemingly random walks. The simulation adds a model of Brownian motion that relies on a combination of damping and random perturbations to simulate the effects of collisions with small molecules, typically water molecules, that are not explicitly represented in the simulation. This model of Brownian motion enforces a thermodynamic constraint relating kinetic energy (defined as one-half the mass times the velocity squared) to temperature. Specifically, the average kinetic energy per degree of freedom of the simulation (including three translational and three rotational degrees of freedom) should be $\frac{1}{2}k_BT$, where k_B is Boltzmann's constant and T is the absolute temperature. In the absence of numerical errors, the use of strictly conservative forces would preserve the kinetic energy of the simulation. In a simulation, however, round-off errors and other errors in numerical approximations will eventually cause such energies to drift in the absence of a corrective method. The simulation model prevents drift by using damping forces to reduce kinetic energy and randomized perturbations to increase kinetic energy back to approximately the correct level.

Damping forces are chosen to represent a continual drag on particles whose force is proportional to particle velocity. The exact force is given by

$$\vec{F}_d = -dm^{(2/3)}\vec{v} \ ,$$

where d is a user-supplied damping constant, m is the mass of the particle, and $\vec{v}$ is its velocity vector. The use of the two-thirds power of m as a coefficient is meant to approximate the contribution of surface area. The approximation is used rather than an explicit calculation of surface area because of the computational difficulty of efficiently determining the exact exposed surface area in the direction of velocity. In an exact continuous model, this force on its own would produce the differential equation

$$m\frac{d\vec{v}}{dt} = -dm^{(2/3)}\vec{v} \ ,$$

whose solution is

$$\vec{v}(t) = \vec{v}(0)e^{-dtm^{(-1/3)}} \ .$$

Thus, from an initial velocity v_0 in any dimension, over a time step Δt, the velocity in that dimension would be expected to decay to $v_0e^{-d\Delta tm^{(-1/3)}}$ in the absence of any other forces.

A similar formula specifies a drag on rotational velocity. This drag is imposed by a torque given by

$$\vec{O}_d = -dI^{(2/3)}\vec{a} \ ,$$

where I is the moment of inertia, $\vec{a}$ is the angular velocity vector, and d is the user-supplied damping constant defined in the preceding paragraph. Once again, an approximation to the actual effect of different particle shapes is used, in the form of the two-thirds power of the moment of inertia, due to the difficulty of calculating the exact value that would be created by rotation of a complex shape about an axis. The rotational damping would be expected to cause an initial angular velocity a_0 over a time step Δt to decay to $a_0 e^{-d\Delta t I^{(-1/3)}}$ in the absence of any other torques.

The correct form of random perturbations to give realistic results currently remains an open problem. I have considered different possibilities and have found that each presents some difficulties. Experience has shown three central issues that must be considered when deciding on a correct model for Brownian motion. The first is whether the model is physically realistic. The second is whether it solves the numerical problem of drifting kinetic energy. The third is whether it allows for reasonable performance of the simulator. So far, I have found no technique that gives completely acceptable results by all three criteria. Although most of this thesis covers only the current state of the simulator, in this chapter we look at both the method in use and another currently rejected method. We examine both methods because I believe the rejected method to be more theoretically satisfying and expect that given other computational improvements, it will become the preferred method for the problem.

The damping force and the expected average kinetic energies place some constraints on the form of the random perturbations. Specifically, in a reasonable model, the perturbations should be selected such that, in the absence of other factors, they on average exactly cancel out the influence of damping on kinetic energy when average kinetic energy is at the expected value. Furthermore, in order for the Brownian motion model to fulfill its purpose of preventing drift of average kinetic energy, the combination of the two must tend to have the property of reducing kinetic energy when it is above the desired average value and increasing it when it is below that value.

In other words, the desired average kinetic energy must be a stable equilibrium point in the absence of other factors. The system is, however, still under-constrained at this point. In particular, we have specified only the average of the kinetic energy, not any of its higher moments. Completely specifying the desired ensemble of a randomized space, such as the space of kinetic energies of the simulation, requires providing additional model characteristics which would fix these other values, possibly in terms of other simulation parameters.

One way to specify the nature of the perturbations is to add the additional assumption that the random perturbations of particle velocity are uniformly distributed over a range based solely on particle mass, moment of inertia, and the specified damping constant. These constraints completely specify the nature of the perturbations, although it is unclear that they necessarily do so in an ideal manner. The literature of statistical physics has not approached the problem of the ensemble of particle states from the specific perspective needed for this kind of simulation model, and so the correct answer for creating the ideal complete set of probabilistic moments is an open question. The use of a uniformly distributed force was initially chosen to provide a conceptually and computationally simple answer.

Given this decision, the only parameter we must determine is the maximum value allowed for the random perturbation. We can calculate this value by noting that the simulator will first apply the perturbation then apply damping, and that these two influences together should produce no expected change in kinetic energy when the initial kinetic energy for a given degree of freedom is the desired average value of $\frac{1}{2}k_BT$. Suppose we begin with an initial translational velocity v_0 along a particular degree of freedom. Then the initial kinetic energy given subunit mass m is

$$K_0 = \frac{1}{2}m{v_0}^2 \ .$$

If we allow a uniformly distributed random perturbation over the range $[-p_{max}, p_{max}]$, then the expected average kinetic energy after application of it will be

$$K_1 = \frac{1}{2p_{max}} \int_{-p_{max}}^{p_{max}} \frac{1}{2} m(v_0 + p)^2 dp \ ,$$

which reduces to

$$K_1 = \frac{1}{2} m({v_0}^2 + \frac{1}{3}{p_{max}}^2) \ .$$

The damping will then multiply the velocity by $e^{-d\Delta t m^{(-1/3)}}$, meaning that the kinetic energy after damping is applied will be multiplied by the square of that, giving an expected kinetic energy after both the random perturbation and the damping have been applied of

$$K_2 = \frac{1}{2} m({v_0}^2 + \frac{1}{3}{p_{max}}^2) e^{-2d\Delta t m^{(-1/3)}} \ .$$

We then must chose p_{max} to meet our constraint that at the desired average initial velocity, $K_2 = K_0$. We first note that when $\frac{1}{2} m {v_0}^2 = \frac{1}{2} k_B T$, $v_0 = \sqrt{\frac{k_B T}{m}}$. Therefore, we want to satisfy the following equation:

$$\begin{aligned}
\frac{1}{2} m(\frac{k_B T}{m} + \frac{1}{3}{p_{max}}^2) e^{-2d\Delta t m^{(-1/3)}} &= \frac{1}{2} k_B T \\
\frac{1}{6} m {p_{max}}^2 e^{-2d\Delta t m^{(-1/3)}} &= \frac{1}{2} k_B T (1 - e^{-2d\Delta t m^{(-1/3)}}) \\
p_{max}^2 &= \frac{k_B T}{m} 3 (e^{2d\Delta t m^{(-1/3)}} - 1) \\
p_{max} &= \sqrt{\frac{k_B T}{m}} \sqrt{3 e^{2d\Delta t m^{(-1/3)}} - 3} \ .
\end{aligned}$$

Thus, we can specify the range over which the random perturbations should vary to give the correct average kinetic energy per degree of freedom.

We can also verify that the expected average kinetic energy is a stable equilibrium point when other forces are not acting on the particle under consideration. We can express K_1 in terms of K_0 as

$$K_1 = K_0 + \frac{1}{6} m {p_{max}}^2 \ .$$

This formulation indicates that the expected increase in kinetic energy is independent of the initial kinetic energy. The decrease produced by damping, on the other hand, increases with increasing initial kinetic energy, as we can verify by taking the derivative of K_2 with respect to K_1. Since $K_2 = K_1 e^{-2d\Delta t m^{(-1/3)}}$, the derivative is merely $e^{-2d\Delta t m^{(-1/3)}}$, which is clearly positive for all finite Δt. Therefore, if initial kinetic energy is above the desired average kinetic energy, then expected kinetic energy after one round of perturbation and damping will be lower than initial kinetic energy. On the other hand, if initial kinetic energy is below the desired average kinetic energy, then the expected kinetic energy after perturbation and damping will be higher than the initial kinetic energy. This model of Brownian motion therefore meets our requirement that it tend to move kinetic energies towards the desired average value. This demonstration that the desired average value is a stable equilibrium point of the model applies similarly for rotational velocity, except that we must compute separate moments of inertia for each rotational degree of freedom, because moment of inertia, unlike mass, depends on the axis of rotation.

The preceding method for modeling Brownian motion, however, has proven largely inadequate in practice because of the necessity of a relatively high damping constant to keep bond oscillations down to a realistic magnitude and to suppress instabilities in the numerical methods. A high damping constant requires large random perturbations to preserve the average kinetic energy of simulations. The combination of high damping and high uniform random perturbations results in subunits pursuing a nearly random walk over time. Since a subunit following a random walk will tend to drift a distance proportional to the square root of the number of time steps that have elapsed, the model results in a rate of diffusion that is low and highly dependent on the length of time steps; these properties are both unrealistic and serious impediments to producing simulations in a reasonable run time, given the diffusion-dependence of many reaction rates. As a result, this method has proven largely inadequate in practice despite its conceptual elegance. While it might be preferable if a more stable numerical method were used that allowed for significantly lower damping, given the current method, a different conception of Brownian motion appears to be necessary.

An alternative method now in use relies on the goal of attempting to randomize particle paths as little as possible, using the Brownian motion model primarily as a way to suppress numerical errors and produce nearly pure Newtonian dynamics. This alternative method chooses random perturbations such that they will only increase velocity along a given degree of freedom, with the range of the magnitude of increase chosen such that the combination of perturbations and damping will produce an expected final velocity closer to the thermodynamically predicted average velocity for the particle than it was at the beginning of the time step. The method works on similar principles to the earlier method, choosing a probability uniformly over a range based on the desired velocity and other particle parameters such that it will be expected to cancel damping exactly when the particle has the thermodynamically-predicted average velocity of $v_{opt} = \sqrt{k_B T/m}$, for absolute temperature T and mass m. The expected multiplicative amount of decay in velocity over the course of a time step of size Δt, given damping constant d and mass m, is $\left(1 - e^{-d\Delta t m^{-1/3}}\right)$, due to the presence of a damping force $-dm^{2/3}v$ for velocity v. Therefore, a velocity perturbation should be expected to produce an average change of $v_{opt}\left(1 - e^{-d\Delta t m^{-1/3}}\right)$. As a result, the magnitude of random perturbations is chosen to be uniform on the range $\left[0, 2v_{opt}\left(1 - e^{-d\Delta t m^{-1/3}}\right)\right]$.

This newer method is used primarily to give reasonable global behavior rather than to model realistically the low-level mechanism of Brownian motion. The central disadvantage of this method is that it is unrealistic, relying on an *ad hoc* means of selecting random perturbations in order to give correct behavior. This method does, however, solve the central problem of low diffusion rates and thus is considerably more practical for producing realistic global simulation behavior. Because this method tends to produce random perturbations that exactly counter damping when velocities are close to the expected value, we can qualitatively assert that particle velocities and directions are close to constant until collisions occur. The result is that the distance a particle is expected to diffuse varies linearly in the number of time steps it is diffusing, rather than roughly as the one-half power as with the previous method.

One final modification of the Brownian motion model which imposes some trade-offs is a decision to apply random perturbations uniformly to clusters of connected particles rather than to individual particles. The main advantage of this modification is that it tends to damp bond oscillations, which created a practical problem in earlier simulations. Stresses formed within structures due to random perturbations driving connected particles in different directions led in practice to significant distortions of bound structures in ways that are physically unrealistic for virus shell assembly. While these distortions might have been controlled by a combination of stronger spring constants and lower error tolerances with the numerical methods, that strategy would have had a significant cost in run time. Applying Brownian motion to connected clusters of particles rather than individual particles provides a solution that allows structures to approximate minimal-stress states. The cost of this solution is that natural bond oscillations and structural flexibility, which may play important roles in some processes, are unrealistically minimized. This minimization may limit the range of phenomena that the simulator can adequately model.

Chapter 5

Numerical Methods

The most crucial numerical task of the code is advancing the simulation time by implementing the Newtonian physics model described in chapter 4. This task consumes a large majority of the run time for typical simulations, and hence the numerical methods must run efficiently to achieve good overall performance. This chapter examines two numerical methods that have been considered for this problem, describes their implementations, and draws some conclusions about their relative merits from the results of a series of performance tests.

Several constraints influenced our choice of numerical methods. Force computations derived from the physical model of chapter 4 can be complicated and time-consuming, as they can include damping forces on all particles, collision forces between different particles and between particles and the simulation boundary, and three spring forces per binding interaction formed. The costly force computations suggest that forces should be computed sparingly. Furthermore, the numerical routines must be robust, because simulation runs often have unpredictable requirements due to variations in simulation characteristics over time. Significant amounts of work could be lost if the numerical methods suddenly became unable to proceed long into a simulation. For similar reasons, the methods must be adaptable in order to perform efficiently on a variety of problems. Design of the numerical methods is also complicated by the fact that discrete operations handling such tasks as forming bonds and updating conformations must be interleaved with the continuous numerical methods.

A final constraint is the availability of parallel hardware for this project, suggesting that the numerical methods should make efficient use of such hardware.

These constraints limit the sorts of numerical methods that are appropriate for the problem. Fixed time step methods are unlikely to provide the level of adaptiveness required by the problem. As a result, multi-step methods are not likely to be usable. Implicit methods are unlikely to be useful, because the complicated force functions make inverting forces efficiently difficult. Limits on time step lengths imposed by the discrete tasks make it inadvisable to rely exclusively on using long time steps combined with high-order methods. Low-order accuracy methods, however, are likely to perform poorly for cases in which considerable optimization is required.

Two methods have been designed to meet the requirements of this problem. The first combines Richardson extrapolation [90] with a simple first-order forward Euler method to achieve efficient common case performance with easy adaptability to situations requiring higher accuracy. The second adapts the waveform relaxation method [64] to work with particle simulations in order to achieve similarly good adaptability and common case performance while taking better advantage of parallel hardware.

Both methods have been implemented to run in parallel with the Cilk multithreading system [9]. Cilk's interface to parallelism is based around a "spawn" directive that invokes a procedure as a distinct thread of execution that can run in parallel with the parent thread that spawned it. The methods currently use Cilk 5.2 (Beta 1) [73], a version of Cilk designed for shared-memory SMP machines. Cilk provides facilities for measuring the parallel performance of a program in terms of its serial work, T_1, and its critical path, T_∞. These measures allow us to derive the available parallelism, T_1/T_∞. The work, critical path, and available parallelism allow estimates of the expected actual performance on different numbers of processors. Cilk's scheduler guarantees that the run time on P processors satisfies

$$T_P = T_1/P + O(T_\infty) \ .$$

Furthermore, the Cilk scheduler has been found empirically to achieve run times of

$$T_P \approx T_1/P + T_\infty$$

when $P \ll T_1/T_\infty$ [86].

The remainder of this chapter examines the two methods, presenting first the extrapolation method and then the waveform relaxation method. Each method is described and analyzed theoretically. In addition, the following sections describe the parallelization of each method using Cilk and provide an analysis in terms of work, critical path, and available parallelism. The final section presents the results of a series of performance tests designed to analyze the parallel performance of each method in practice. While both methods are shown theoretically to be well-suited to the needs of the problem, the performance tests indicate that the extrapolation method performs better in work and critical path for most problem classes considered, and should therefore generally be the preferred method. The waveform relaxation method, however, usually makes more efficient use of parallelism and should be comparatively more favorable for use with other hardware or software systems than those available to this project.

5.1 Extrapolation-Based Method

The first attempt at a numerical method for evaluating the time evolution of the system was based on the Euler method with Richardson extrapolation [90]. This method was chosen because of the expectation that it would allow high adaptability, providing a means to update particle information quickly when possible, but also allowing efficient computation when greater optimization is required. This section first examines the serial method, explaining the major techniques it uses — Euler iteration, Richardson extrapolation, and an adaptive step size — as well as the details of its implementation. The section then describes the parallelization of the method with Cilk and shows that it has asymptotic $O(N)$ work and $O(N/\log N)$ available parallelism in the number of nodes simulated.

Serial Method

The Euler method is the simplest finite-difference method for numerically integrating systems of differential equations. The forward Euler method uses the following assumption to approximate a value x at different time points:

$$x(t + \Delta t) \approx x(t) + \Delta t \frac{dx(t)}{dt} \ .$$

The related backward Euler method uses the approximation

$$x(t + \Delta t) \approx x(t) + \Delta t \frac{dx(t + \Delta t)}{dt} \ .$$

These assumptions give the following two discrete iteration formulas:

$$\begin{aligned} x_{n+1} &\leftarrow x_n + \Delta t f(x_n) \ , \\ x_{n+1} &\leftarrow x_n + \Delta t f(x_{n+1}) \ , \end{aligned}$$

where f is a function computing the derivative of x at a given time point. Applying the second formula requires inverting the function $x - \Delta t f(x)$, which can be done either analytically or numerically. Both versions produce errors that are first order in the step size.

The simulation model adapts the basic Euler method to the problem at hand using a two-part method for updating position and velocity data. The method updates simulation state by first calculating new velocity data based on the current positional data via forward Euler, then updating positional data based on the new velocity information via backward Euler. The method is described by the following two equations:

$$\vec{v}_{n+1} = \vec{v}_n + \Delta t f(\vec{u_n}, \vec{v_n}) \ ,$$

$$\vec{u}_{n+1} = \vec{u}_n + \Delta t \vec{v}_{n+1} \ ,$$

where $\vec{v}$ represents both translational and rotational velocities, $\vec{u}$ represents both translational and rotational positions, and f computes the derivatives of the velocities in terms of the current positions and velocities based on the forces and torques acting on the simulated particles. While this combined forward-backward Euler is still only first-order accurate, it yields slightly better constants than purely forward or purely backward Euler.

Although applying these formulas is straightforward for translational velocity, it is more complicated for rotational forces. The derivative of translational velocity is determined by applying Newton's second law of motion, $\vec{F} = m\vec{a}$, to the forces specified by the physical model, dividing forces by particle masses to determine acceleration. An analogous law, $\vec{T} = I\vec{r}$, gives the derivative of rotational velocity in terms the torque $\vec{T}$ acting on a particle and its moment of inertia I about its axis of rotation. Rotational velocity, however, cannot be added to current rotational position in a straightforward manner to update the position. In order to address this problem, we apply a mathematical technique used in computer graphics applications for updating current rotation by some fixed amount of additional rotation. The technique is based on a representation of a rotation as a quaternion. For a rotation of θ radians about an axis (x, y, z), the quaternion representing that rotation is given by $a + b\hat{i} + c\hat{j} + d\hat{k}$, where

$$\begin{aligned} a &= \sin(\theta) \ , \\ b &= \frac{x}{\sqrt{x^2+y^2+z^2}} \cos(\theta) \ , \\ c &= \frac{y}{\sqrt{x^2+y^2+z^2}} \cos(\theta) \ , \\ d &= \frac{z}{\sqrt{x^2+y^2+z^2}} \cos(\theta) \ . \end{aligned}$$

Using this representation, composition of rotations is accomplished simply by multiplying their corresponding quaternions [40]. This technique provides a simple method to convert rotational velocity into the derivative of rotation, as needed by the Euler method. In practice, I have found the heuristic of composing the current rotation with the product of the rotational velocity and the time step size produces faster convergence than using the derivative of rotation, as the strict Euler method requires. We therefore use the heuristic in practice despite the fact that it is not as well-supported theoretically as the pure Euler method.

Richardson extrapolation is a technique for combining low-order accuracy approximations at different step sizes to yield higher-order accuracy approximations. Richardson extrapolation requires that the approximation have an error of the form

$$e(h) = e_1 h + e_2 h^2 + e_3 h^3 + \cdots \ ,$$

where h is some parameter — the step size in this case — that can be adjusted. Richardson extrapolation uses linear combinations of approximations at different values of h to eliminate low-order error terms, yielding higher-order accuracy approximations. For example, given approximations $x(\Delta t)$ and $x(\Delta t/2)$ to an ideal value x^* at two step sizes, an improved approximation, $\hat{x}(\Delta t)$, can be derived as follows:

$$\begin{aligned}
\hat{x}(\Delta t) &= 2x(\Delta t/2) - x(\Delta t) \\
&= 2x^* + 2e_1\frac{\Delta t}{2} + 2e_2\frac{\Delta t}{2}^2 + 2e_3\frac{\Delta t}{2}^3 + \cdots - x^* - e_1\Delta t - e_2\Delta t^2 - e_3\Delta t^3 - \cdots \\
&= (2x^* - x^*) + (e_1 - e_1)\Delta t + \frac{1}{2}(e_2 - 2e_2)\Delta t^2 + \frac{1}{4}(e_3 - 4e_3)\Delta t^3 + \cdots \\
&= x^* - \frac{1}{2}e_2\Delta t^2 - \frac{3}{4}e_3\Delta t^3 - \cdots
\end{aligned}$$

The result is a second order approximation in Δt. Second order approximations at different step sizes can then be combined to give third order approximations and so on, yielding an order of accuracy as high as is necessary for the problem [90]. Richardson extrapolation is applied during the updating of subunit positions and velocities on each time step to yield higher-order accurate approximations.

The combination of Euler and Richardson extrapolation lends itself naturally to an adaptive step size method. On each time step, the simulator first calculates an approximation using the base step size as the step size for the Euler method. The simulator then halves that step size and repeats the Euler calculation using two of these half-size sub-steps. It then performs Richardson extrapolation on the two approximations to yield a second order approximation. Convergence is tested by measuring the distances between the original first order positional and rotational data and the newly calculated second order data and seeing if they fall below user-specified translational and angular tolerances in each dimension. In addition, the old and new first order values are also compared. Lower-order values are calculated and saved in addition to the highest-order values allowed by the extrapolation for each step because the behavior of collisions and stiff springs means that higher-order accuracy will not necessarily produce higher absolute accuracy and so lower order-accuracy solutions may converge to the final answers more quickly. If convergence fails for any subunit for each comparison, then the method continues by halving the step size again on those subunits that failed to converge while holding it constant on those that did converge, performing another round of the Euler iteration, and using Richardson extrapolation to produce a higher-order accurate solution. Convergence is again tested between new order n and old order $n-1$ solutions for each value n so far calculated, as well as between the two most recent first-order values. This procedure continues until all subunits have converged on the same step (possibly using different sub-step sizes for different subunits) or some subunit has reached a maximum number of subdivisions, currently set at 256. If the simulation fails to converge, then the simulator cuts the base step size in half and attempts to run on the two halves of the interval with the error tolerances halved in each. This process of subdivision may be repeated to produce further successive halving if necessary. The successive halving heuristic is meant to increase robustness of the method, providing a means to apply extrapolation to smaller sub-intervals if the extrapolation fails to work well on the full time step. The overall adaptive method allows the amount of work to vary considerably from particle to particle and over time as the needs of the problem dictate, promoting quick convergence where possible and a high order of accuracy where necessary.

Parallelization

The primary parallelization used for the extrapolation method is extremely simple. The simulator spawns separate threads of execution for each subunit being evaluated on each sub-step. The simulator begins each sub-step by performing a successive binary division of nodes, spawning separate threads for each half of the available nodes until either a single thread is handling each node or a maximum number of subdivisions, based on the number of processors, has been achieved. The maximum number of subdivisions was chosen empirically to be $\log_2(4P)$, where P is the number of processors. The use of a maximum number of subdivisions is a heuristic meant to reduce total work when running on small numbers of processors without significantly impeding the ability of Cilk's work-stealing runtime system to distribute work evenly among available processors. Each thread is responsible for calculating all forces acting on the subunits it handles and updating position and velocity data for those subunits on a single Euler sub-step. Once all threads have completed this task, they synchronize. They can then either repeat the process for the next Euler sub-step, or, if all sub-steps for the current base step are complete, perform extrapolation and test for convergence. Although extrapolation and convergence testing are in principle highly parallelizable, their serial cost is sufficiently small compared to force calculations that we process them serially.

For the purposes of analyzing the parallelization, we will ignore the use of a pre-set maximum number of threads. We will also assume some fixed per-node cost T on a given sub-step. Given these assumptions, the work of the parallel method is given by the cost of subdividing the particles and the cost of evaluating them in separate threads. Both costs are linear in the number N of nodes present, giving $T_1 = O(N)$. The critical path is given by the cost of subdividing along a single thread of execution followed by the cost of evaluating a single node after subdivision. The number of subdivisions is $O(\log N)$ and the cost of evaluating a single node is the constant T. Thus, $T_\infty = O(\log N)$. From this critical path value, we can conclude that the available parallelism satisfies $T_1/T_\infty = O(N/\log N)$. In practice, the amount of work may vary from node to node contrary to our assumption of fixed work per node, for instance if some nodes are bound to others or involved in collisions while others are free in solution, which could reduce the available parallelism proportionally to the ratio between the maximum and the average work per node.

5.2 Waveform Relaxation-Based Method

A central problem with the previous method is the large number of "spawn" calls and synchronizations it needs. A new numerical method was therefore implemented to reduce these costs and thus improve parallelism. Although Cilk is efficient at implementing spawn and synchronization operations, those operations nonetheless can begin to consume considerable run time as the number of threads increases, limiting the amount of parallelism that can be used efficiently. I therefore implemented a waveform relaxation method [64], a type of numerical method designed to give high parallel performance on numerical integration problems of many variables.

Waveform relaxation treats a system of coupled equations as a collection of independent "waveforms." The technique involves iteratively recalculating each waveform independently, given the present values of the other waveforms, using some standard serial numerical integration technique. These iterations are repeated until all waveforms converge to the correct solutions for the global problem. The waveform relaxation technique was originally developed for circuit simulations, a problem that typically involves integrating large systems of coupled differential equations. The major advantage of waveform relaxation methods on shared-memory multiprocessors is that they require only infrequent synchronization, since the solutions of distinct waveforms on a given round can be performed in parallel, with synchronization required only between rounds of the global iteration. Waveform relaxation therefore seemed to be a good choice for the shared-memory symmetric multi-processor machines available during the later development of the local rules dynamics simulator.

This section describes an adaptation of the waveform relaxation method for use with the simulator. It first covers the serial implementation of the method, in which an embedded Runge-Kutta method is combined with the general waveform relaxation framework. The section also discusses some complications in dealing with rotational motion. It then describes the method's parallel implementation with Cilk and concludes that it, like the extrapolation method, has asymptotic $O(N)$ work and $O(N/\log N)$ available parallelism in the number of nodes simulated.

Serial Method

The serial method adapts the general waveform relaxation principle to a particle simulation framework. One modification of the general scheme convenient to this problem is that all parameters for a given subunit are treated as a single waveform, as opposed to treating each variable as a distinct waveform. Thus, twelve variables — three translational coordinates, three rotational coordinates, three components of translational velocity, and three components of rotational velocity — are treated as a unit and evaluated together by a standard serial scheme. The waveform relaxation algorithm combines waveforms for all particles into a unified solution to the overall simulation's evolution.

Individual particle waveforms are evaluated by an embedded Runge-Kutta method. Embedded Runge-Kutta methods [74] are a form of adaptive numerical method in which lower-order accuracy and higher-order accuracy solutions to a numerical integration problem are computed simultaneously for less computational cost than producing the two solutions independently would require. The two solutions provide a built-in accuracy check on the final results. They can also save some work by allowing the accuracy of a solution to be improved, up to some fixed order, without requiring computation of each improved solution from scratch. For reasons related to the collision-detection algorithm described in section 6.2, it is convenient to limit the method to third order accuracy, with a fourth order accuracy calculation used for error checking. The specific numerical method used is adapted from that developed by Zonneveld [125] and is described by the following equations:

$$k_1 = f\left(t_0, x_0\right) \ ,$$

$$x_1 = x_0 + hk1 \ ,$$

$$k_2 = f\left(t_0 + \frac{1}{2}h, x_0 + \frac{1}{2}hk_1\right) \ ,$$

$$x_2 = x_0 + hk_2 \ ,$$

$$k_3 = f\left(t_0 + \frac{1}{3}h, x_0 + \frac{1}{2}hk_2\right) \ ,$$

$$k_4 = f\left(t_0 + h, x_0 + hk_3\right) \ ,$$

$$x_3 = x_0 + h\left(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4\right) \ ,$$

$$k_5 = f\left(t_0 + \frac{3}{4}h, x_0 + h\left(\frac{5}{32}k_1 + \frac{7}{32}h_2 + \frac{13}{32}k_3 - \frac{1}{32}k_4\right)\right) \ ,$$

$$x_4 = x_0 + h\left(-\frac{1}{2}k_1 + \frac{7}{3}k_2 + \frac{7}{3}k_3 + \frac{13}{6}k_4 - \frac{16}{3}k_5\right) \ ,$$

where t_0 is the initial time, x_0 is the simulation state at t_0, h is the time step, f is a function for computing the derivative of the simulation state with respect to time from the forces acting on the particle, and x_i is an ith order approximation to the simulation state at time $t_0 + h$. On each sub-step, the simulator computes k_1, x_1, k_2, and x_2 before testing for convergence between x_1 and x_2. Convergence is tested by evaluating whether distance and rotation values for the two approximations are within user-specified distance tolerances of one another in each degree of freedom when scaled by the fraction of the base step that is contained in the current sub-step. If the convergence test fails, the simulator then computes k_3, k_4, and x_3 and applies a second convergence test between x_2 and x_3. If that test also fails, it then computes k_5 and x_4 and tests for convergence between x_3 and x_4. If even this attempt was unsuccessful, the simulator readjusts the time step and tries again. Otherwise, it accepts the solution and moves on to the next sub-step.

The adaptive time step scheme attempts to partition the base user-supplied step size into smaller sub-steps in order to achieve the desired accuracy with a small number of sub-steps. Initially, the simulator assumes only a single sub-step covering the full interval. If that fails to achieve convergence even at fourth order, then the simulator calculates a new step size based on the estimated error at the previous step size. The new step size is chosen to be

$$\min\left\{\tau_{rem}, \left(\frac{\frac{1}{2}t_t h_s}{h e_t}\right)^{\frac{1}{3}}, \left(\frac{\frac{1}{2}t_r h_s}{h e_r}\right)^{\frac{1}{3}}\right\} ,$$

where τ_{rem} is the time remaining in the base step, t_t and t_r are allowed maximum translational and rotational error tolerances, e_t and e_r are the estimated errors from the previous round, h is the base step size, and h_s is the sub-step size from the previous round. The three factors are an estimate of how much time, up to the total amount left in the current time step, the simulator can cover while remaining within the error bounds. If the simulator again fails to achieve convergence with the reduced sub-step size, then it reduces the sub-step size again according to the same formula. If the method fails to converge or converges at the fourth order then the simulator recomputes the sub-step size based on the estimated error using the same formula as above. If it converges with a lower-order approximation, the simulator increases the sub-step size to covers the full time remaining for the base step. This basic scheme is modified with a heuristic to prevent the sub-step size from increasing or decreasing too rapidly. We allow a maximum step size change of 2 times following convergence at second order and 1.5 times following convergence at third or fourth order. Step size is decreased by at most 8 times following a failure to converge. While these modifications are *ad hoc*, they were found empirically to improve performance in a common case in which the basic formula for sub-step size estimation causes sub-step sizes to “see-saw” between extremely small sub-steps at which the method converges and much larger sub-steps at which it does not, leading to negligible progress.

Running the adaptive embedded Runge-Kutta method for a single particle produces a series of data points describing particle position, rotation, velocity, and rotational velocity at each sub-step. This information suffices to derive a piece-wise cubic approximation to each of these parameters over the course of the entire time step. Because the scheme never achieves higher than third order accuracy (with the fourth order approximation providing an accuracy check on the third order scheme), a piece-wise cubic approximation has adequate accuracy to approximate the position of a given particle when another particle needs to know it for its own evaluation.

There is a complication in the evaluation of rotational motion, in that rotation varies non-linearly with rotational velocity. As was mentioned in discussion of the extrapolation method, rotational velocity is not the derivative of rotational position. In order to find the instantaneous derivative in rotational position needed by the numerical methods, we can apply the mathematical transformation described in section 5.1 for converting composition of rotations into multiplication of quaternions. Given a rotational velocity $\vec{u}$, we can determine the instantaneous derivative of the current rotation $\vec{r}$ by applying the function f that converts rotations into quaternions. Specifically, the derivative is given by

$$\frac{d\vec{r}}{dt} = \frac{d}{dt} f^{-1}\left(f(\vec{r}) \cdot f(\vec{u}t)\right) \ .$$

For $\vec{r} = (r_x, r_y, r_z)$ and $\vec{u} = (u_x, u_y, y_z)$, the resulting derivative $\frac{d\vec{r}}{dt} = \left(\frac{dr_x}{dt}, \frac{dr_y}{dt}, \frac{dr_z}{dt}\right)$, after some simplification, reduces to

$$\frac{dr_x}{dt} = \frac{(r_y r_y + r_z r_z)u_x - r_x(r_y u_y + r_z u_z)}{2|\vec{r}|\tan(|\vec{r}|/2)} + \frac{\vec{r}\cdot\vec{u}}{\vec{r}\cdot\vec{r}} r_x + \frac{1}{2}(r_z u_y - r_y u_z) \ ,$$

$$\frac{dr_y}{dt} = \frac{(r_x r_x + r_z r_z)u_y - r_y(r_x u_x + r_z u_z)}{2|\vec{r}|\tan(|\vec{r}|/2)} + \frac{\vec{r}\cdot\vec{u}}{\vec{r}\cdot\vec{r}} r_y + \frac{1}{2}(r_x u_z - r_z u_x) \ ,$$

$$\frac{dr_z}{dt} = \frac{(r_x r_x + r_y r_y)u_z - r_z(r_x u_x + r_y u_y)}{2|\vec{r}|\tan(|\vec{r}|/2)} + \frac{\vec{r}\cdot\vec{u}}{\vec{r}\cdot\vec{r}} r_z + \frac{1}{2}(r_y u_x - r_x u_y) \ .$$

These values are calculated from the current rotation and the current rotational velocity when the derivative of current rotation is needed by the numerical methods.

The solutions of individual waveforms by the embedded Runge-Kutta method described earlier in this section are combined into the global waveform relaxation algorithm. A single round of the method, which advances particle states by one time step, begins by establishing an initial waveform for each particle. The initial waveforms assume that each particle starts with the same state as it had at the end of the previous time step and that its velocity and rotational velocity are unchanged over the course of that time step. Once initial waveforms have been established, the simulator repeatedly runs the Runge-Kutta scheme individually on each waveform, using the last completed values of all other waveforms to calculate the positions of other particles in the simulation. Convergence is tested each time all waveforms have been recomputed by comparing the final state of each waveform with the final state derived from the previous approximation and accepting only if all differences fall within the user-defined angle and distance tolerances. The process continues until either all waveforms converge on a given round or a hard-coded maximum number of iterations has been reached. A maximum of ten iterations was empirically determined to provide good performance. In the event that the maximum is reached, the simulator breaks the time step into two sub-steps and evaluates each in turn, using half the error tolerances that it used on the full time step. This binary subdivision is repeated as necessary until convergence is achieved.

Figure 5-1 shows pseudo-code for the overall waveform relaxation method. The algorithm first creates an "initial guess" waveform for each node, as described in the last paragraph. It then repeats a central loop in which new waveforms are calculated based on those from the previous round, convergence is tested, and, if convergence fails, the old waveforms are replaced by the new waveforms to begin the loop again. If convergence is not achieved within MAX_ROUNDS iterations of the **while** loop, the procedure breaks and signals failure, indicating smaller sub-steps must be used. Otherwise it signals success, indicating that the simulator can proceed to the next sub-step.

Parallelization

The parallelization of the waveform relaxation-based method follows straightforwardly from the serial method. We can parallelize the method by spawning a separate thread for each element in each **for** loop in the pseudo-code of figure 5-1. The convergence test, though, is not actually parallelized in practice due to its small serial cost for any reasonable simulation size; actual run time is dominated by the **for** loop of lines 7–10. A theoretical analysis of that **for** loop's parallelization on N nodes can proceed identically to that of the parallelization of force field calculations with the

```
for each node i
   create an initial waveform w_{0,n}
end for
converged ← false
rounds ← 1
while (not converged) and (rounds < MAX_ROUNDS)
   for each node i
         calculate a new waveform w_{1,i} using the embedded Runge-Kutta
         procedure, based on position data in w_{0,j} for all j ≠ i
   end for
   converged ← true
   for each node i
         if convergence failed
            converged ← false
         end if
   end for
   if (not converged)
         for each node i
            w_{0,i} ← w_{1,i}
         end for
   end if
end while
return (converged)
```

Figure 5-1: Pseudo-code for the waveform relaxation method applied by the simulator. The pseudo-code describes the process of updating node position and velocity data for a single sub-step.

extrapolation-based method. Assuming a constant maximum cost for an execution of the Runge-Kutta method, the **for** loop has work $T_1 = O(N)$ and critical path $T_\infty = O(\log N)$, again giving available parallelism $T_1/T_\infty = O(N/\log N)$.

5.3 Comparing the Methods

This section compares the performance of the two methods on a series of simulations. The tests measure both serial and parallel performance. The section concludes that the extrapolation method is superior in most cases for both serial and parallel operation. The waveform relaxation method, though, does make more efficient use of parallelism.

In order to test the practical performance of the two methods, I constructed a series of tests designed to stress different aspects of the simulator code. Each test involves running the simulator on a specific kind of problem for a fixed length of simulation time and a range of problem sizes. In each case, problem sizes consist of numbers of particles of all powers of 2 from 1 to 1024, with workspace volume scaled proportionally. For each test, I characterize processing speed and parallel performance of the numerical methods according to three measures: total work, critical path length, and available parallelism. All tests are carried out on an 8-processor Sun Ultra HPC 5000 SMP running SunOS 5.7. The code is compiled using Cilk's `solaris-threads` architecture option. For the purposes of the tests, a compiler constant that limits the number of threads spawned in certain regions of the extrapolation method is set to 65,536, effectively removing the maximum thread restriction for the problem sizes examined. Both versions of the simulator code are invoked with the Cilk options `-nproc 8 -heap 256`, specifying that all eight processors will be used and that a maximum heap size of 256 megabytes will be available. Work and critical path are measured with Cilk's internal work and critical path timers, with measurements taken only around the subroutines performing the actual updating of particle state, including updating of binding patterns but specifically excluding time spent waiting for input or in communication with the user interface. The five tests examine the effects of the

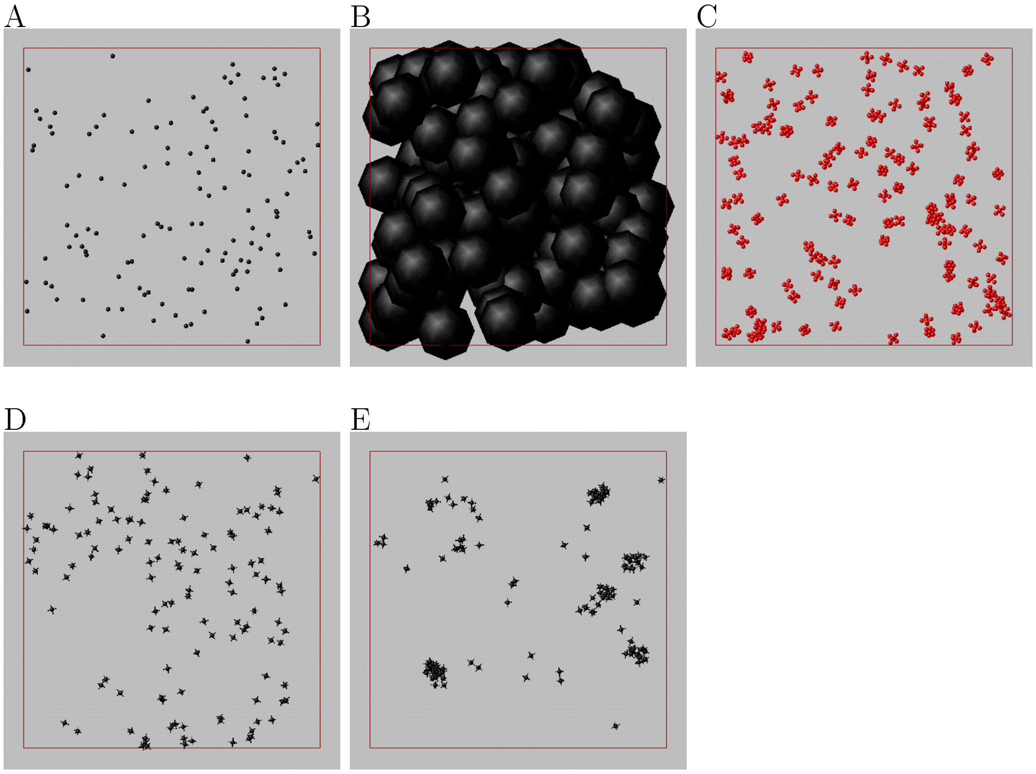

Figure 5-2: Screen snapshots of the five tests. Each snapshot shows a test following a 128 node run. A: test one; B: test two; C: test three; D: test four; E: test five.

following cases: simple nodes at low density; simple nodes at higher density; nodes with complex shapes; rapid forming and breaking of binding interactions; and stable, high strength binding interactions. Figure 5-2 illustrates the problems with screen snapshots of 128 node simulations of each of the five tests.

Test one is meant to test the basic updating functions. It uses simple spherical particles with no binding sites placed in a large enough volume that collisions should be infrequent. Particles have masses of 100 kDa, radii of 1.5 nm, and no binding sites. Simulation parameters give a distance error tolerance of 0.1 nm, angle tolerance of 0.1 radians, temperature of 293 K, and damping of 2. For a simulation of N particles, the simulation workspace is a cube with volume $15,625N$.

Figure 5-3 shows the resulting data. The graph reveals that the extrapolation method has consistently lower total work. Although extrapolation initially also has a lower critical path, it soon exceeds that of the waveform relaxation method. Available parallelism reveals that both methods yield increasing parallelism with increasing problem size, but that the rate of increase is noticably higher for waveform relaxation than for extrapolation.

Test two is meant to examine how the simulations handle extensive numbers of collisions. This test is expected to stress both the algorithms for collision dynamics themselves and the numerical methods, which require more work to handle the more complicated particle trajectories that result. Simulation conditions are identical to those of test one except that the simulation volume for an N particle test is reduced to $8N$.

Figure 5-4 shows the resulting data. In this case, the extrapolation method performs significantly better than the waveform relaxation method by both work and critical path measures for all problem sizes. Waveform relaxation does lead to considerably more efficient use of parallelism, but its larger critical path length means that it will nonetheless be expected to perform more poorly than extrapolation on this test regardless of numbers of processors available. Further examination reveals that the central cause of the work and critical path discrepency is a blow-up of kinetic energy in the waveform relaxation simulations, requiring much greater work at optimization. In other tests not shown here, I find that using one tenth the time-step to promote smaller truncation errors significantly mitigates the problem, as does reducing error tolerances, which implicitly reduces the time step, and increasing damping, which suppresses velocity increases for both methods.

Test three is meant to examine the influence of complicated shapes on nodes. Test conditions are identical to those of test one except that each node consists of seven spheres of mass 100 kDa and radius 1 nm, arranged as a central sphere with another sphere at an offset of ± 2 nm along each of the X, Y, and Z axes. This node design is intended to examine both the cost of running code for dealing with multiple spheres per particle and the indirect effects of more complicated shapes, such as the need to account for torsional forces on collisions.

Figure 5-5 shows the data for test three. Test three is interesting in that the waveform relaxation method yields consistently lower work and critical path than

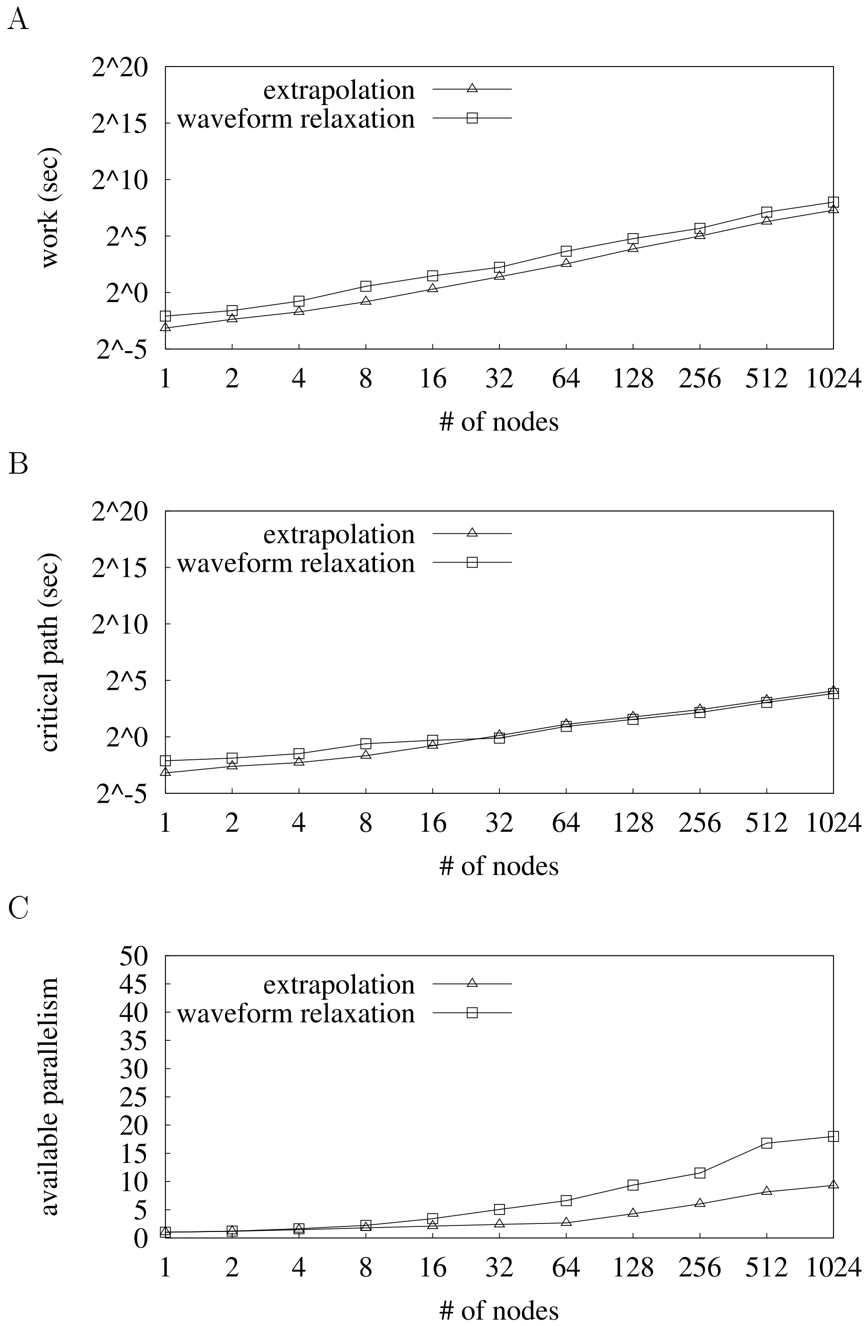

Figure 5-3: Performance results for a test of simple nodes at low concentration. A: log-log plot of work; B: log-log plot of critical path; C: semi-log plot of available parallelism.

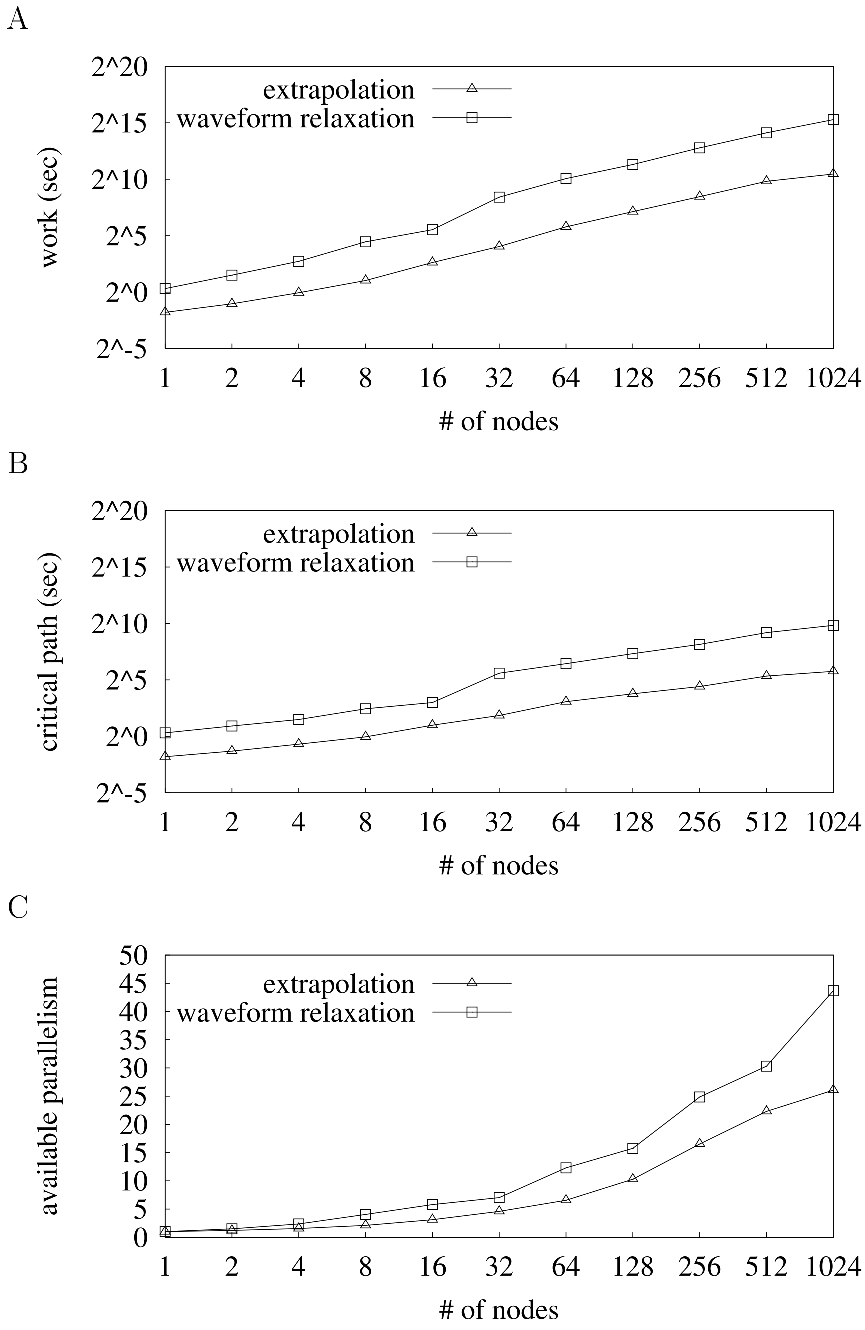

Figure 5-4: Performance results for a test of simple nodes at high concentration. A: log-log plot of work; B: log-log plot of critical path; C: semi-log plot of available parallelism.

extrapolation. It is also the only one of the five tests for which extrapolation yields consistently higher available parallelism than waveform relaxation.

Test four is designed to stress a purely serial part of the code: the forming and breaking of bonds. It uses simulation conditions identical to those of test one, except that each node has six binding sites, one projecting from each of the positive and negative X, Y, and Z axes. These edges each have length 2 nm. They are set to have association and dissociation energies of 0, implying that each edge has a 50 percent chance per time step of forming a binding interaction if it is within its allowed tolerances of another edge and a 25 percent chance per time step of breaking a bond if it is already involved in one. Tolerances are set to 10 nm for translation and 30 degrees each for rotation and bending. All spring constants are set to 0. Thus, although the simulation still performs the mathematical operations of calculating spring forces, the springs have no effect on particle trajectories and therefore should not influence the numerical methods.

Figure 5-6 shows the performance data for test four. The data shows similar performance for the two methods, with extrapolation having somewhat lower work and generally slightly smaller critical path length than waveform relaxation. Waveform relaxation has a somewhat higher available parallelism, although for both methods the available parallelism is low. The low parallelism is unsurprising, since the test is designed to stress a serial portion of the code.

Test five is meant to examine the influence of strong bond forces on the two numerical methods. It uses a simulation setup identical to that of simulation four except that bond parameters have been changed. Bond energies are set to 30,000 kcal/mole for formation and $-30,000$ kcal/mole for breaking, which at the temperatures examined means that bonds have a probability of $\sim 1.5 \times 10^{-7}$ of forming if unbound edges are correctly positioned or breaking if a bond is already formed. For the short simulation time the test examines, this choice of energies effectively means that the bond state can be assumed static. In addition, bond tolerances are set to 1 nm for translation and 5 degrees each for rotation and bending, in order to ensure that few bonds will ever be close enough to require testing for bond formation. The

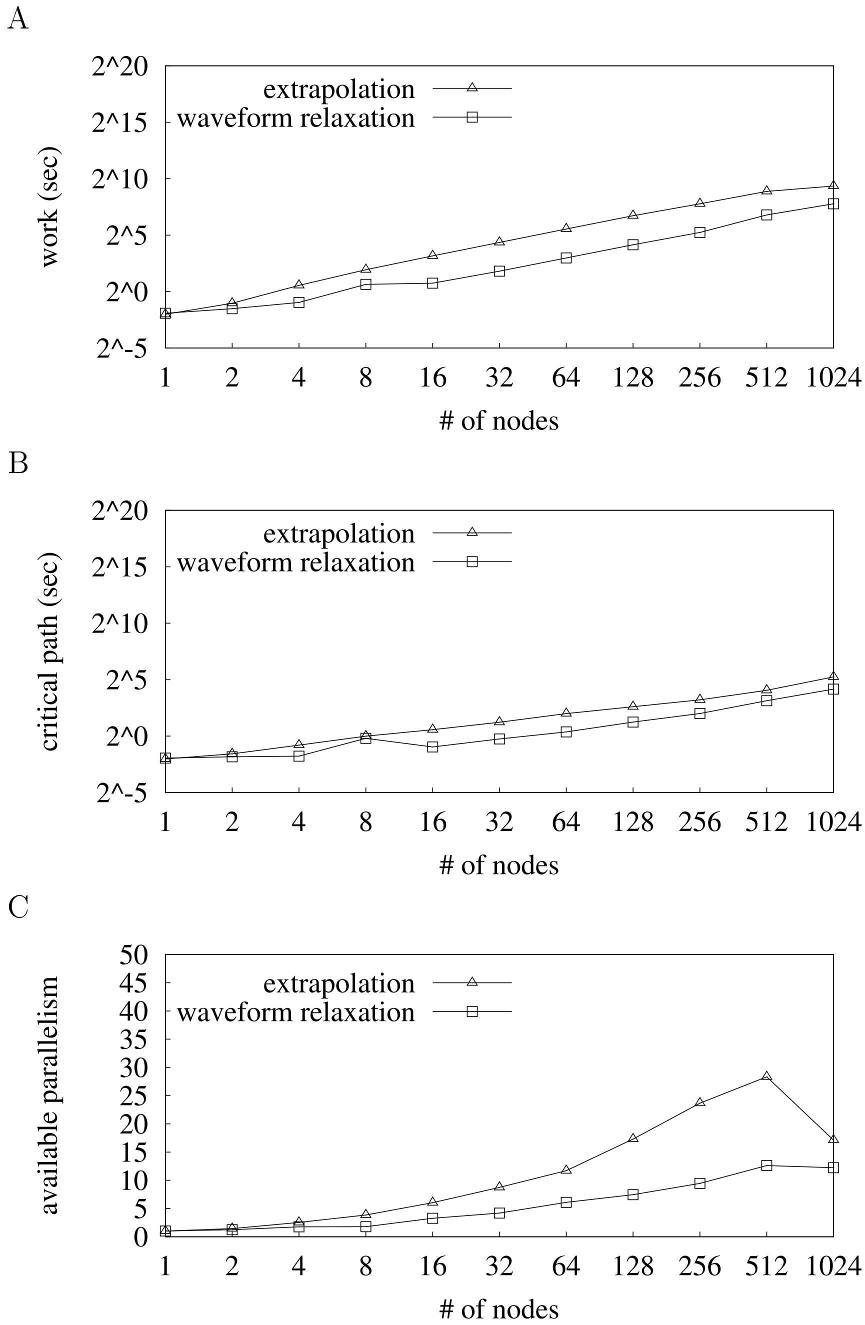

Figure 5-5: Performance results for a test of nodes of complex shape. A: log-log plot of work; B: log-log plot of critical path; C: semi-log plot of available parallelism.

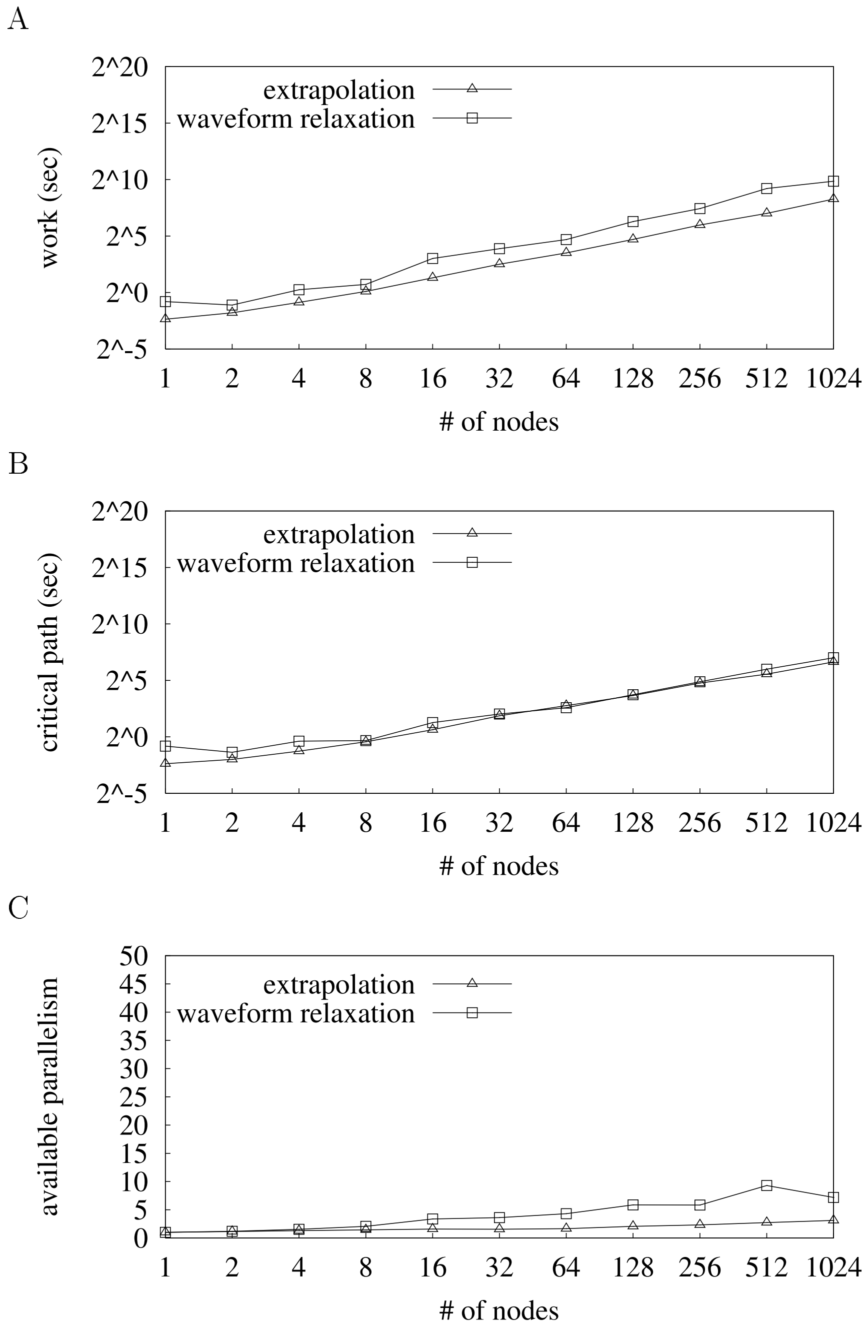

Figure 5-6: Performance results for a test of nodes designed to form and break bonds rapidly, but to exert no binding forces. A: log-log plot of work; B: log-log plot of critical path; C: semi-log plot of available parallelism.

bonds have strengths of 10,000 units in each dimension. These spring constants are relatively strong, providing a test for how spring forces influence simulation behavior.

Figure 5-7 shows the results for test five. The results are similar to those of test two, with extrapolation having significantly shorter work and critical path, and waveform relaxation having a noticeably higher available parallelism. Further examination of the simulation progress reveals that the cause of the large difference between the two methods is a blow-up of particle velocities similar to that seen with test two, as the waveform relaxation method yields rotational kinetic energy growing faster than it can be damped for the parameter range considered.

One important caveat in interpreting these results comes in weighing the relative importance of the specific tests in judging how the methods are likely to perform in practice. Each test exaggerates some aspects of normal simulation behavior, and so none by itself reflects realistic simulation behavior. Test one is probably the most accurate reflection of a practical simulation early in its run time, before subunits have had a chance to assemble into larger structures. Test five may better capture a more advanced stage of a typical self-assembly simulation. Test three does not accurately reflect most applications to date, as we have so far made little use of node shape in controlling simulation behavior. It does, however, represent a direction we would like to pursue towards more realistic nodes. Tests two and four, although included to test the behavior of certain aspects of the code that would not normally be stressed, are not realistic representations of normal simulation practice; while studying parallel performance in those cases may be of value in fine-tuning code, performance on those cases should not be considered particularly important to judging the effectiveness of either method in practice.

On the basis of the results, we can draw some general conclusions about the effectiveness of the two methods. With the exception of test three, extrapolation leads to consistently lower total work, suggesting that it should generally be the preferred method for serial execution or when only a small number of processors is available. With the exception of test three, the extrapolation method also produces a smaller critical path than waveform relaxation, suggesting that even for arbitrarily large num-

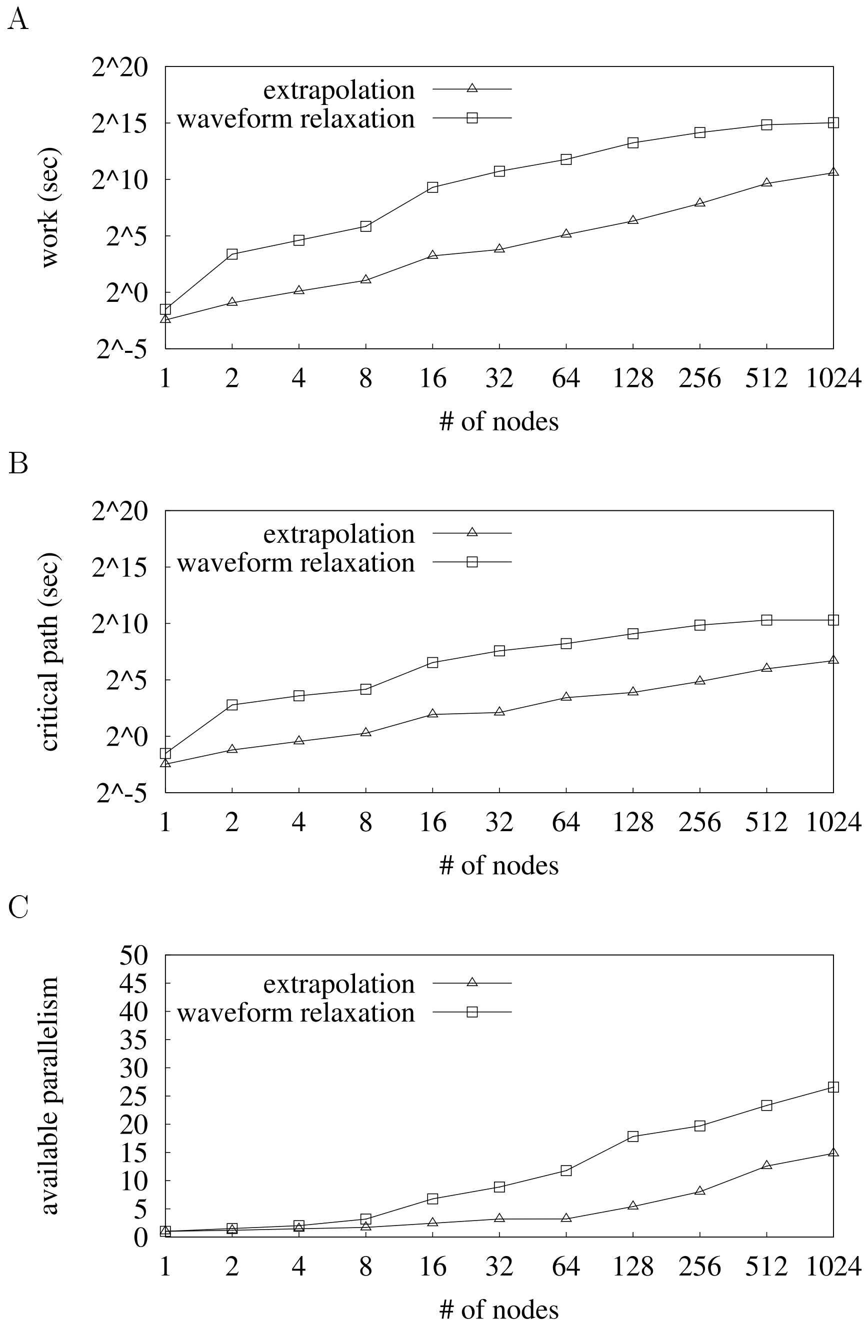

Figure 5-7: Performance results for a test of nodes with strong, highly stable binding interactions. A: log-log plot of work; B: log-log plot of critical path; C: semi-log plot of available parallelism.

bers of processors, the waveform relaxation method is unlikely to produce shorter run times than the extrapolation method. The waveform relaxation method, however, generally makes significantly more efficient use of parallelism, suggesting that as the number of processors available increases, waveform relaxation should perform comparatively better, at least for large problem sizes. The one unusual exception is when complex nodes are involved, in which case the above results appear to be reversed, possibly indicating that the waveform relaxation method is more effective at handling the kinds of rotational motion produced by collisions of complex particles. We can also conjecture that changes in architecture or programming environment might change the relative performance of the two methods. In particular, the extrapolation method depends to a greater degree than the waveform relaxation method on the efficient fine-grained parallelism allowed by the low spawning and synchronization costs of the SMP implementations of Cilk. Implementing these methods on an architecture with higher communication costs or with a parallel programming environment that was not as effective as Cilk at using fine-grained parallelism would likely make the extrapolation method less favorable relative to the waveform relaxation method.

One interesting note is that the strengths of the waveform relaxation method we have detected are largely inherent to the waveform relaxation scheme in general while the weaknesses seem to result mainly from the particular single-node serial method chosen. As the tests reveal, waveform relaxation tends to give much better available parallelism than the extrapolation method. This result is as we would expect, since waveform relaxation shifts more work out of serial portions of the code and increases the amount of work done between thread synchonizations. The two key problems with the waveform relaxation method as we apply it are its relatively high work compared to the extrapolation method and its reduced effectiveness at supressing increases in particle velocities over time. The high work is largely a reflection of the fact that the adaptive Runge-Kutta method often requires tiny sub-steps even at its maximum order of accuracy. This need for small sub-steps may argue for using a higher order method than the third order with fourth order error checking we use in practice, despite the complications a higher-order method would cause.

In addition, the method wastes considerable work determining reasonable sub-step sizes, often causing it to fail to converge to a solution or to take somewhat smaller steps than necessary. More advanced methods of step size selection might be able to reduce these problems substantially. The blow-up of kinetic energies, which results from accumulated truncation errors of the method, is also a problem of the Runge-Kutta method rather than the general waveform-relaxation framework, and might be mitigated or eliminated by a better choice of serial method. The problems with the waveform relaxation method may therefore be primarily a reflection of the fact that we have more experience with the extrapolation method and therefore have accomplished more towards optimizing it for our problem. It thus appears possible in principle to get the high parallelism of waveform relaxation without the high work and unstable velocities that we have found in its current incarnation. Further research is needed to see if this improved application of waveform relaxation is feasible.

Chapter 6

Algorithms

This chapter discusses the major discrete algorithms used by the simulator. The most significant discrete algorithms are those responsible for locating compatible binding sites in order to form binding interactions and for detecting collisions between particles in simulations. Collision detection requires two variants, one for use with extrapolation and one for use with waveform relaxation. The algorithms all use bucket-based methods to achieve linear time and space performance in the number of nodes (for collision detection) or the number of binding sites (for updating binding interactions). For each algorithm, the chapter presents the problem to be solved, describes the chosen approach, and analyzes its theoretical performance.

6.1 Formation of Binding Interactions

One problem concerns how to locate potential binding partners for free bonds in a simulation. For each unpaired binding site found in a simulation, it is necessary to locate all other binding sites that have compatible types and are within the allowed translational, rotational, and bending tolerances. If any such compatible sites exist, the simulator can then decide which, if any, will bind to the site under consideration. This section describes an algorithm for the problem combining hash tables and a bucket-based system. It explains the components of the algorithm and proves that it requires linear time and space in the number of binding sites.

A simple, naive solution to the problem of locating binding partners is to iterate through all binding sites and, for each, iterate again through all binding sites of higher index, testing each pair for compatibility by the type and tolerance criteria. This method has the problem, however, that it requires time quadratic in the number of binding sites. Since a typical simulation has several binding sites per particle and this operation must be performed every time step, the run time of the naive algorithm could quickly become prohibitive.

Our solution uses a bucket-based system to locate binding partners efficiently. Each time the algorithm is run, it begins by assigning each binding site to a bucket in space depending on the location of its endpoint. The dimension of the buckets is the maximum distance tolerance of all defined edge types, guaranteeing that a binding site need only consider binding sites in its own bucket and those adjacent to it when looking for potential binding partners. Thus, once all binding sites have been placed in their buckets, only a constant number of buckets need to be searched in order to locate all possible binding partners. Any binding sites so located can then be checked according to the full criteria for potential binding partners in order to identify exactly those that are compatible.

In order to avoid high memory overhead, the full array of buckets is never created, but is replaced by a hash table. Some mechanism is needed to reduce memory overhead because the number of buckets could potentially be extremely high if the simulation volume is large or the distance tolerances are small. In order to address the high potential memory overhead, the simulator creates a hash table of buckets in which storage is allocated only for occupied buckets. The result is that memory overhead can be kept proportional to the number of binding sites, rather than the total number of buckets possible in the simulation workspace. The algorithm uses a universal hash function [12] in which integer x, y, and z bucket coordinates used as keys are each multiplied by a random coefficient, selected when the space for the hash table is allocated, with the sum of the resulting values taken modulo the size of the hash table specifying the hash line for the bucket. Collisions are handled via chaining [67, 30]. The hash table size allocated is 4 times the number of edges to be stored in it.

In order to analyze the expected theoretical performance of the algorithm, we need to define some variables:

- Let E = the total number of binding sites in the simulation.
- Let e_{max} = the maximum number of binding sites per sphere.
- Let L = the maximum distance of a binding site from its corresponding sphere.
- Let t = half the largest binding distance tolerance.
- Let r = the minimum sphere radius.
- Let R = the maximum sphere radius.
- Let H = the number of lines in the hash table.

For the purposes of this analysis, we assume that all spheres in a simulation are non-overlapping. If spheres can overlap, then there is no theoretical reason why all spheres, and all binding sites, could not be concentrated in an arbitrarily small space, reducing the algorithm to $O(N^2)$ worst-case performance in the number of binding sites. The assumption of non-overlapping spheres is not strictly accurate, since spheres have some elasticity on collisions under our model, but it is nearly true in practice.

Under these assumptions, we can first show that the number of edges per bucket is bounded by a function independent of E. The bucket size is defined to be $2t$ in each dimension. In order for a binding site to terminate inside the bucket, its sphere must either be within the bucket or entirely contained within a distance $L + R$ of it. The space within the bucket or within distance $L + R$ of it is illustrated in two dimensions in figure 6-1. The region first contains the bucket itself, which has volume $8t^3$. Next, it contains six rectangular solids surrounding the bucket, each of which has length $2t$ in two dimensions and $L + R$ in the third, for a total volume of $48(L + R)t^2$. Finally, the eight rounded corners of the volume together make up a sphere of radius $L + R$,

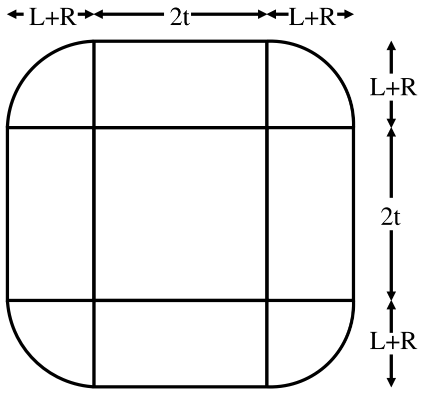

Figure 6-1: A two-dimensional illustration of the volume containing all possible spheres whose edges might terminate within a given bucket. The actual three-dimensional volume used in the proof consists of the cubic bucket, 6 rectangular solids projecting from the cube's faces, and 8 corners that are identical in shape to the eight octants of a sphere.

contributing volume $\frac{4}{3}\pi(L+R)^3$. Thus, the region has total volume

$$8t^3 + 48(L+R)t^2 + \frac{4}{3}\pi(L+R)^3 \ .$$

Each sphere has minimum volume $\frac{4}{3}\pi r^3$. Therefore, given the assumption that spheres do not overlap, the maximum number of spheres occupying the volume is given by

$$\frac{8t^3 + 48(L+R)t^2 + \frac{4}{3}\pi(L+R)^3}{\frac{4}{3}\pi r^3} \ .$$

Therefore, the number of binding sites terminating within the bucket is bounded by the constant

$$K_E = e_{max}\frac{8t^3 + 48(L+R)t^2 + \frac{4}{3}\pi(L+R)^3}{\frac{4}{3}\pi r^3}$$

This argument can be strengthened in two ways. If we weaken our initial condition that spheres do not overlap to the more realistic assumption that they never overlap by more than some fraction d of their radii, then the analysis proceeds identically, except that the constant K_E is replaced by

$$K_E = e_{max} \frac{8t^3 + 48(L+R)t^2 + \frac{4}{3}\pi(L+R)^3}{\frac{4}{3}\pi((1-d)r)^3} \ .$$

This bound can be made tighter at the cost of greater complexity. We can, for instance, note that spheres adding edges to one bucket may preclude others from being present to add edges to a neighboring bucket. The result, however, is presented only to prove that there exists a bound K_E independent of E, and we do not therefore need to make it as tight as possible.

We can now show that the process of inserting binding sites into the hash table takes expected linear time in E. The universal hash function ensures that the expected probability of collision for two different buckets is E/H [12]. Since we select H such that $H = \Omega(E)$, we can bound the expected number of collisions above by a constant. The time to insert a new node into a hash table involves the time to locate its hash line in the table, the time to locate its bucket within that hash line or to create it if it is not present, and the time to insert the binding site into the bucket. Locating the slot requires the constant time cost of evaluating the hash function. Locating the bucket takes constant time per bucket in the hash line, which is itself an expected constant, since the number of buckets per slot is bounded by one more than the number of collisions. Once the bucket is located, constant time is required to insert the binding site. Thus, inserting each binding site requires constant expected time. Inserting all E binding sites therefore requires $O(E)$ expected time.

The process of detecting potential binding interactions and updating binding status also takes expected $O(E)$ time. For each binding site, locating potential binding partners requires looking up all binding sites in a fixed number of buckets and testing each against the tolerances of the binding site to which it is being paired. Locating sites in a given bucket requires constant time to evaluate the hash function and expected constant time to locate the correct bucket and distinguish it from other buckets occupying its hash line. Testing tolerances of the resulting edges requires constant time per binding site contained in the bucket, for a bounded number of binding sites per bucket and a constant number of buckets, yielding constant time. Thus, expected constant time per binding site is required by the updating procedure to update a given binding site's status. For all E binding sites, this analysis gives an expected $O(E)$ run time. Putting together the cost creating the hash table and applying it to locate binding partners, the total expected run time of the bond updating algorithm is $O(E)$, that is, proportional to the number of binding sites.

We should also consider time to allocate memory for and clear the hash table, which will take time proportional to H. Our use of a hash table size satisfying $H = \Theta(E)$ gives asymptotically ideal performance. It requires $O(E)$ time to create and clear the table. In addition, the chosen size meets the requirement of $\Omega(E)$ size needed to give expected $O(E)$ performance for the remainder of the algorithm. Therefore, the $\Theta(E)$ hash table size gives an overall $O(E)$ expected run time for the algorithm even after allocating and clearing the hash table is taken into account. This choice of table size preserves overall $O(E)$ space complexity for the algorithm as well.

6.2 Collision Detection

Another problem similar to the binding interaction problem discussed in section 6.1 involves locating collisions between particles in a simulation. As the physical model specifies, single nodes belonging to different particles exert forces on one another when their radii overlap. This statement is equivalent to saying that they exert forces when the distance between them is less than the sum of their radii. The problem is thus similar to that of locating binding partners, in that it is necessary to find particles that are close together. There is again a naive $O(N^2)$ algorithm in the number of nodes, in which all pairs of particles are explicitly tested relative to each other. We can, however, improve on this algorithm by once again adopting a bucket-based approach using hashing to reduce memory overhead. Two different variations on a basic approach are needed for this problem, one for each of the numerical methods used. This section describes each method and analyzes it theoretically. It demonstrates that the methods require linear time and space in the number of nodes. In addition, each is parallelized with Cilk to give $O(N/\log N)$ available parallelism.

Collision-Detection with Extrapolation

The extrapolation method uses a straightforward bucket-based algorithm much like that used for bond formation. At each sub-step, each particle has a position based on the last sub-step at which it was updated. So, for example, at $\frac{7}{8}$ of the current time step, there may be some particles last updated at $\frac{1}{2}$ of the current time step, others at $\frac{3}{4}$, others at $\frac{13}{16}$, and so forth. At the beginning of each force calculation, the workspace is partitioned into buckets each of whose dimension is twice the largest single node radius. Each single node is then assigned to the bucket containing its center. Collisions between a single node and its neighbors can then be detected by examining only its own bucket and those immediately adjacent to it, as is done with binding partner detection. Once again, memory is conserved by storing buckets in a hash table, allocating memory only for those buckets that actually contain at least one particle. The hashing method is identical to that used for locating binding partners, with a universal hash function using the coordinates of the buckets as keys.

The theoretical analysis is also similar to that used with the binding partner algorithm. In order to perform the analysis, we begin by defining some variables:

- Let $N =$ the total number of spheres present.
- Let $n =$ the number of spheres being updated on the current sub-step.
- Let $r =$ the minimum sphere radius.
- Let $R =$ the maximum sphere radius.
- Let $H =$ the number of lines in the hash table.

We also need to make certain assumptions. For the purposes of the analysis, we assume as with the binding partner algorithm that spheres do not overlap. The analysis, however, still applies with a change in the constants if we assume instead that spheres can overlap by no more than a certain fraction of their radii.

We can now establish that no more than a constant number of spheres are present in a given bucket. All spheres centered within a given bucket must be contained entirely within a region bounded by a distance R outside the edge of a cube of size $2R$ in each dimension. By the same argument as was used in the binding partner algorithm, this region has a volume of

$$8R^3 + 48(R)R^2 + \frac{4}{3}\pi R^3 = \left(56 + \frac{4}{3}\pi\right) R^3 \ .$$

Any sphere contained within that region must have volume at least $\frac{4}{3}\pi r^3$, so the volume can contain at most $\left(56 + \frac{4}{3}\pi\right) R^3 / \left(\frac{4}{3}\pi r^3\right)$ spheres, a quantity we define to be K_C. If we assumed instead only a fraction d overlap between spheres rather than no overlap at all, then K_C would change to $\left(56 + \frac{4}{3}\pi\right) R^3 / \left(\frac{4}{3}\pi((1-d)r)^3\right)$.

Given that K_C provides a bound independent of N, we can establish that inserting spheres into the hash table requires expected $O(N)$ time. Each insertion requires time to evaluate the hash function, time to locate the correct bucket in the hash line or determine that it is not present, and time to insert the sphere into the bucket. Evaluating the hash function requires constant time per sphere. Locating the correct bucket requires constant time per bucket present in the hash table line. Given that we choose H such that $H = \Omega(N)$, the expected number of buckets per hash table line is constant for any particle distribution due to the universal hash function. Therefore, locating the bucket requires constant time per sphere. Finally, inserting spheres requires constant time per sphere. Thus, expected constant time is required for the insertion of each of the N spheres, for a total expected run time of $O(N)$ for all N spheres to be inserted.

Likewise, checking for collisions between nodes on a given sub-step requires expected $O(n)$ time. Checking for collisions requires looking up the contents of 27 boxes in the hash table and comparing the positions of all spheres in those boxes to that of the sphere for which collisions are being detected. Locating each box in the hash table requires constant time per box to evaluate the hash function and expected constant time per box to locate that box in its hash line. Because the number of spheres per box is bounded by K_C, constant time in N and n is required per box to evaluate collisions between a given sphere and the spheres in that box. Thus, expected constant time is required for each of the n spheres being evaluated, giving an algorithm that requires $O(n)$ expected time.

Once again, because the hash table size is $\Theta(N)$, having been set this time to 4 times the number of nodes it contains, allocating its memory and clearing its elements takes $O(N)$ time. Thus, these operations do not increase the expected overall run time derived from the other operations of the algorithm. Had the value been $o(N)$, the insertion and lookup would have expected time $\omega(N)$. Had the hash table had size $\omega(N)$, the allocation and clearing operations would take $\omega(N)$ time. Thus, the use of a value that is $\Theta(N)$ provides asymptotically optimal time complexity while maintaining $O(N)$ space complexity.

The method has also been parallelized with Cilk. The parallelization occurs in two stages. First, the initial insertion of single nodes into the hash table is explicitly parallelized by spawning off a tree of threads in which the leaves of the tree assign individual nodes to the hash table. In order to avoid errors from race conditions, the algorithm uses an array of mutual exclusion locks (mutexes) to lock any hash table line while a new element is being added to it. In the second stage, when the hash table is actually accessed to locate collisions, the parallelization is implicit; the code is accessed within the force field calculation code, which is itself parallelized to produce separate threads for each node. Because the second stage of collision detection is read-only, no locking is needed. Threads for each node can detect all of their collisions in parallel with threads for all other nodes.

We can evaluate the expected parallel performance of the node insertion portion of the algorithm by looking at the theoretical work, $T_1(N)$, and the critical path of the computation, $T_\infty(N)$. The work of the parallelized algorithm is given by the recurrence

$$T_1(N) = 2T_1(N/2) + O(1) \ ,$$

due to the constant overhead of the splitting at each level and the need to proceed recursively on two halves of the original set of nodes. The base case of the recurrence is $T_1(1) = O(1)$ expected run time by the same argument as for the purely serial algorithm. The solution of the recurrence is $T_1(N) = O(N)$ expected work. Given unlimited processors, run time is limited by the longest path of the tree of possibilities, giving the recurrence

$$T_\infty(N) = T_\infty(N/2) + O(1) \ .$$

The base case is again $T_\infty(1) = O(1)$, because each individual node must be handled serially. The possibility of hash table collisions slowing down the algorithm by requiring locking and serialization of accesses to some lines does not affect the expected asymptotic run time, because the expected number of collisions per line is constant. The critical path recurrence is solved to yield $T_\infty(N) = O(\log N)$. Combining these parallelized work and critical path measures, we can conclude that the theoretical available parallelism is $T_1(N)/T_\infty(N) = O(N/\log N)$.

Analyzing the parallel performance of the actual collision detection is more complicated because the spawning of threads is already handled by the parallel extrapolation method. To be fair, we assume that cost on a given sub-step is part of the cost of the collision detection algorithm. Therefore, on a sub-step in which n nodes must be evaluated for collisions, the algorithm requires spawning off n threads by binary division of an initial list, followed by performing the actual collision detection. The work of these operations is given by

$$T_1(n) = 2T_1(n/2) + O(1) \ ,$$

with a base case of $T_1(1) = O(1)$ expected work using the same argument as given for the purely serial analysis earlier in this section. Analogously to the filling of hash tables, the recurrence solves to $T_1(n) = O(n)$. The critical path is given by the recurrence

$$T_\infty(n) = T_\infty(n/2) + O(1) \ ,$$

with again $T_\infty(1) = O(1)$ expected run time as a base case. Thus, the expected asymptotic critical path is $T_\infty(n) = O(\log n)$. We can therefore conclude that the actual collision detection similarly has $O(n/\log n)$ expected available parallelism.

Collision Detection with Waveform Relaxation

The algorithm requires some adaptation for use with the waveform relaxation method. The problem with applying the bucket-based algorithm to waveform relaxation arises from the fact that the sub-steps used when calculating waveforms over one time step may differ from particle to particle. Thus, it may be necessary to determine whether collisions have occurred between two particles at a given time point even though one particle's position was never explicitly calculated at that time point. The algorithm deals with this problem by calculating, for each particle, a piece-wise cubic interpolant of particle positions over the entire waveform. The interpolant allows testing of when particles cross bucket boundaries, on the basis of which a modified time-stamped bucket system is used. Each node entered into a bucket carries a time-stamp recording when it is in that bucket. When a node needs to look up at some time point the other nodes with which it might be colliding, it retrieves those in nearby buckets, but discards those whose time intervals do not include the current time. Thus, it is possible to apply the bucket based system even when nodes are updated on different sub-steps.

Some changes are required to allow the algorithm to work with waveform relaxation. First, complete nodes rather than individual spheres are stored in the buckets containing their centers, in order to avoid calculating interpolants for each individual sphere of a rotating particle given the waveform of the particle itself. Consequently, the bucket size must be increased sufficiently to capture the farthest possible distance between the center of a node and any part of its mass. The code approximates this distance by determining a bounding box about each node type and using the distance between the center and the farthest corner of the bounding box as one half of the final bucket dimension. A more complicated change is that, for each node, piece-wise cubic interpolants are required to allow the node's position and rotation to be determined at arbitrary points along its waveform. Because a full waveform rather than a specific position is inserted into buckets, the algorithm must consider that a node may occupy more than one bucket over course of the time step. Thus, nodes are entered into all buckets they occupy, with a time stamp recording when they occupy them.

Converting position and velocity data calculated by the numerical method to piece-wise cubic waveforms is straightforward. Once a waveform is calculated, each sub-step has some initial and final positions and velocities. These four pieces of data uniquely specify a cubic polynomial over the interval for each coordinate. Calculating the cubic interpolant, $at^3 + bt^2 + ct + d$ for a given coordinate from the initial and final positions, $x(0)$ and $x(\tau)$, and the initial and final velocities, $\dot{x}(0)$ and $\dot{x}(\tau)$, is accomplished by the following equations:

$$\begin{aligned} a &= \frac{\tau\dot{x}(\tau) - 2x(\tau) + \dot{x}(0)\tau + 2x(0)}{\tau^3} \ , \\ b &= x(\tau) - a\tau^3 - \dot{x}(0)\tau - x(0)\tau^2 \ , \\ c &= \dot{x}(0) \ , \\ d &= x(0) \ . \end{aligned}$$

For evaluating interpolants for rotational motion, this formula requires the derivative of rotational position, not rotational velocity as we define it. Given the piece-wise cubic function, the position of a particle's center can be easily calculated at any time over the time interval it covers by locating the correct cubic interpolant, which we do by binary search over the set of time intervals for a given waveform, and plugging the current time into the polynomial.

Once the piece-wise cubic polynomials are known for a particle, they can be assigned to buckets over the course of the time step. The central algorithmic problem consists of locating all time points over the interval at which the particle crosses bucket boundaries. Because waveforms are made of piece-wise cubic polynomials, this problem can be easily converted to the problem of finding roots of cubic polynomials, a problem for which a general analytic solution exists. The overall detection of all bucket crossings over a time interval can be handled efficiently by repeatedly stepping to the next boundary crossing in any dimension and recalculating boundary cross-

```
for each node waveform w
  first_round ← true
  for each piece-wise cubic sub-step s of w
        if (first_round)
          bucket ← the bucket corresponding to s's starting position
          current ← 0
          first_round ← false
        end if
        next ← next intersection time of a cubic in s with a wall of bucket
        while (next < current)
          insert an entry for [current, next) in bucket
          current ← next
          bucket ← the new bucket entered at time next
          next ← next intersection time of a cubic in s with a wall of bucket
        end while
  end for
  final ← the final time in s
  insert an entry for [current, final] in bucket
end for
```

Figure 6-2: Pseudo-code for assigning a particle whose position is given by piece-wise cubic polynomials in three dimensions to buckets over the course of a time step.

ings in that dimension until the time step is exceeded. Figure 6-2 gives pseudo-code for the overall method for calculating buckets for the piece-wise cubic polynomials. The embedded Runge-Kutta method used for updating individual waveforms is restricted to third order accuracy, with fourth order error testing, in order to promote conceptual simplicity in dealing with this time-stamped bucket algorithm. Although higher-order methods can used, doing so requires storing more information during application of the numerical method in order to allow fourth or higher order polynomial interpolants to be calculated. In addition, calculating bucket crossings becomes more difficult, since finding roots of a general polynomial involves an impractically complicated formula at fourth order and cannot be accomplished analytically at higher orders, meaning that some more complicated and probably less robust root-finding method is required to locate bucket crossings.

The theoretical analysis is similar to that for the extrapolation version of the collision algorithm with some added complications. First, because full nodes rather than single nodes are entered into buckets, when analyzing the maximum number of nodes that may occupy a bucket, we need to substitute the maximum distance from the center of a node to a corner of its minimal bounding box, which we denote D, for the maximum sphere radius R. This substitution gives a maximum number of spheres per box, assuming maximum overlap fraction d, of $\left(56 + \frac{4}{3}\pi\right) D^3 / \left(\frac{4}{3}\pi((1-d)r)^3\right)$, with constants otherwise defined as in the analysis performed for collision detection with extrapolation.

A second issue results from the fact that a hash table is constructed once per waveform relaxation iteration, rather than once per sub-step as with extrapolation. Thus, we can analyze the cost of the method per time step rather than per sub-step. We must, however, consider three additional factors in the analysis: the number W of waveform relaxation rounds required, the average number S of sub-steps per node, and the average number B of times a node crosses a bucket boundary on a time step. We cannot theoretically argue that any of these values is bounded, since they depend on the behavior of the force function controlling the particles, which itself contains discontinuities. For the purposes of this analysis, we treat these values as constants, but I can only empirically assert that they are small in practice.

Given these complications, we can establish that insertions of nodes into hash tables for the evaluation of one time step takes expected linear time in the number N of nodes. (Note that N no longer refers to the number of spheres.) Each node insertion into a given bucket requires the constant time to evaluate the hash function, expected constant time to locate the correct bucket on the appropriate hash line, and constant time to insert the node, for a total expected constant time. Each node is inserted B times on average per round for a total of W rounds. There are N nodes total, giving an insertion cost per time step of $O(BWN) = O(N)$, since we treat B and W as constants.

Checking for collisions over the course of the time step also requires expected $O(N)$ time. By the same logic as in the analysis of collision detection with exrapolation, a node on a given force field evaluation must consider a constant number of buckets, each having a constant number of nodes. The time to look up those nodes and evaluate their effects will include the constant time per lookup to evaluate the hash function, expected constant time to locate the correct buckets, and constant time to evaluate the effects of each node in the bucket on the node being tested for collisions. A constant number of such evaluations are needed per sub-step for an expected constant time per node, per sub-step. Accumulating over each of the S sub-steps of each of the W rounds for each of the N nodes gives $O(SWN)$, which is $O(N)$ given our prior assumptions.

The waveform relaxation method uses two-stage parallelization similar to that used with the extrapolation method. Once waveforms for the nodes have been calculated, the first stage of the algorithm assigns each node to the time-stamped hash table in parallel with all other nodes through explicit spawning. Once again, locking is needed on individual hash lines, because all nodes must write to the same hash table. Later collision detection is implicitly parallelized as with the extrapolation-based method, because it occurs during the force-field calculation subroutines of the previously parallelized code for updating particle waveforms. Thus, each node's collision-detection calculations can be processed in parallel with those of all other nodes.

The analysis of the parallelization is identical for the assignment of nodes to the hash table to that done for extrapolation. To the $O(N)$ expected serial run time, we need to add the $O(N)$ work of spawning threads for each of the N nodes, giving expected work of $T_1 = O(N)$ for each insertion. The critical path of insertion is the $O(\log N)$ critical path of spawning the threads plus the expected constant work of inserting a single node, giving an expected critical path of $T_\infty = O(\log N)$. This result yields an expected available parallelism for the operation of $O(N/\log N)$.

To evaluate fairly the parallel performance of the actual collision detection on a time step, we account for the spawning of threads built into the waveform relaxation method by assuming it is part of the per-time step cost of the collision detection. Thus, detecting all collisions over the course of a time step requires the $O(N)$ work of spawning N threads for the particles involved plus the expected $O(N)$ serial work of detecting collisions, for a total expected work of $T_1 = O(N)$. The critical path consists of the $O(\log N)$ critical path of spawning the threads, plus the expected constant work of evaluating collisions for a single node, giving $T_\infty = O(\log N)$. Thus, the expected available parallelism for this portion of the algorithm is again $O(N/\log N)$.

Chapter 7

User Interface

User interface design is possibly the most neglected area of research in biochemical simulations, and a chapter is therefore devoted to describing how it is handled in the present work. The user interface is structured as an interpreted control language with a graphical front end. This design is intended to provide versatility through a powerful control language that allows a fine degree of control over the details of simulation runs while also providing ease of use through a simple graphical interface to most common operations. The interface is written in C as a single serial program. It connects via a network socket to the separate parallel numerical routines, which may or may not be running on the same computer. This chapter describes the user interface design in detail. It first explains the implementation and operation of the command interpreter. It then similarly explains the graphical user interface. Next, it briefly explains a network interface by which the user interface can connect via a client-server arrangement with a separate program running the parallel numerical routines. Finally, the chapter evaluates the user interface design and argues that it provides a powerful but generally easy-to-use interface, although it is still too complicated to be of general use.

```
(define add_node (procedure (type pos vel ang angvel) (sequence
 (make_node type pos vel ang angvel)
 (define t (get_array the_node_types type))
 (define x (make_list type pos vel ang angvel
            (if (= (first t) "single")
                (make_list (add_single type num_nodes)))
            (if (= (first t) "complex")
                (add_complex type num_nodes))
            (if (= (first t) "variable")
                (add_variable type num_nodes))))
 (set_array the_nodes num_nodes x)
 (set num_nodes (+ num_nodes 1))
 x)))
```

Figure 7-1: A sample of code in the simulator's control language. The code comes from the controller, a file of interpreter-level routines that implement a variety of essential node behaviors. This particular code defines a procedure `add_node` that creates a node of a defined type with a specific position, velocity, rotation, and angular velocity.

7.1 Command Interpreter

This section describes the user interface's command interpreter. It first provides a general introduction to the interpreter's syntax. It also explains some details of its implementation. The section concludes with a discussion of the various means of interacting directly with the command interpreter.

The command interpreter implements a Lisp-like control language through which all user commands are passed. The control language provides a number of primitive operations for interacting with the physical model. In addition, it contains a selection of more general commands and facilities for constructing user-defined procedures and data structures, making it possible to create complicated programs to assist with simulation operation and data analysis. In addition to a variety of primitive commands, several more complicated built-in functions of the user interface are implemented in the control language in terms of lower-level primitives. Figure 7-1 contains a sample of code written in the simulator control language. A full reference to the control language is provided in appendix A.

For the most part, operation of the interpreter is straightforward. The main function of the interpreter is handled by two procedures, one of which evaluates sub-expressions and the other of which performs procedure applications. The most significant algorithmic problem in the design of the interpreter is variable look-up, which is handled via a hash table of currently assigned variable names. As variables are added to a current execution frame, either by an explicit `define` command or by inclusion in a list of procedure arguments, each variable name is hashed by a multiplicative hash function on the ASCII values of all characters in the name and stored. Collisions are handled by chaining [67, 30]. Variables of the same name are stored in a linked list, with new incarnations added to the head of the list. This method works well with the dynamic allocation method used, since only the most recently allocated version of a variable is ever accessible; thus, only constant time is needed to add new variables, look up the current incarnation of a variable name, or remove variables that are no longer in scope, once the linked list corresponding to the variable name has been located. If we further assume the hash table size is large compared to the number of distinct variable names and the hash values of them are uniformly distributed, then this method is expected to give constant time variable look-up in the number of defined variables.

Data types are generally allocated by primitive procedures and are freed through automatic garbage collection when no longer accessible. Deallocation is handled by a mark-and-sweep garbage collector. Garbage collection can occur at top-level between execution of user commands and at the beginning of the evaluation of any expression or sub-expression. When garbage collection is performed, the garbage collector first passes through a list of all currently defined variables and marks each corresponding memory location as in use. In addition, variables with compound types, such as lists and arrays, are searched recursively so that any memory locations they point to are also preserved. The interpreter also maintains a list of memory locations that are being used internally at a given time even if they are not user-accessible. All memory values needed transiently by the interpreter during other evaluations are "registered" by the interpreter on this list and "de-registered" when no longer needed. Members of this list and other values reachable by them are also marked as in use. Once all values in use are marked, the garbage collector frees all those that are unmarked before returning control to the evaluation routine or to the top-level interface.

Users have several means of interacting with the interpreter. The primary means is a graphical user interface, described in section 7.2, which communicates with the interpreter via several pre-defined "hooks." Options exist, however, for situations where more detailed control is needed. Users can type commands directly into the interpreter via an interaction window produced by the graphical interface. Alternatively, a compilation option allows the user to create a text-only interface that provides direct access to the interpreter, which is useful for running automated jobs that do not require a graphical display. Finally, users can load files to control simulator behavior. One commonly used file type is a "rules file," which is used to set up a simulation. A simple example appears in figure 7-2. Another file type is the "save file," which is generated automatically by the simulator to record the simulation state. A sample save file appears in figure 7-3. The final file type is a "control file," used to configure aspects of the simulator's behavior. A special control file, `controller.dat`, that is loaded automatically by the simulator at start-up, implements the controller, which defines many of the basic functions of the simulator's default behavior in terms of lower-level primitives. The sample code of figure 7-1 is taken from this file. User control files may redefine default procedures such as the `add_node` procedure specified in the code sample, or may define other auxiliary procedures as needed. The distinction between the three file types is, however, based on how they are used and not on their format. In fact, all three are written in the simulator control language. They are loaded by the same mechanism and passed directly to the interpreter. As a result, it is possible to edit or mix elements any of them. For example, control constructs useful to a given simulation are commonly defined as part of a rules file for that simulation.

```
(resize_workspace 50 50 50)
(set_constant "damping" 2)
(set_constant "timestep" 0.1)
(set_constant "tolerance" 0.1)
(set_constant "temperature" 293)
(set_constant "steps_per_update" 10)
(define updates_per_run 100)
(add_single_node_type 5 0 10 (make_list) (make_list) (make_list))
(add_complex_node_type
 (make_list (make_list 0 0 0) (make_list -2 0 0)
            (make_list 2 0 0) (make_list 0 -2 0)
            (make_list 0 2 0) (make_list 0 0 -2)
            (make_list 0 0 2))
 (make_list (make_list 0 0 0) (make_list 0 0 0)
            (make_list 0 0 0) (make_list 0 0 0)
            (make_list 0 0 0) (make_list 0 0 0)
            (make_list 0 0 0))
 (make_list 0 0 0 0 0 0 0))
(define rproc (procedure () (add_random_node (make_list 1))))
(for rproc 10)
```

Figure 7-2: A rules file defining two node types and specifying the creation of ten nodes of the second type at random positions within the workspace

```
(resize_workspace 75 75 12)
(set updates_per_run 100)
(set_constant "e0" 1)
(set_constant "u0" 1)
(set_constant "g" 1)
(set_constant "timestep" 0.1)
(set_constant "tolerance" 0.1)
(set_constant "angle_tolerance" 0.1)
(set_constant  "temperature" 293)
(set_constant  "steps_per_update" 10)
(set_constant  "epsilon" 1e-08)
(set_constant  "damping" 2)
(add_single_node_type 5 0 10 (make_list ) (make_list ) (make_list ))
(add_node 0
(make_list -13.4696 -0.476514 1.13998)
(make_list 0.569152 0.0206 785 0.559002)
(make_list 0.0674091 -0.40032 0.771236)
(make_list -0.14926 -0.429504 0.169154))
(add_node 0
(make_list 26.0308 0.0167601 4.73207)
(make_list 0.448519 -0.4913 87 0.90004)
(make_list -0.0226069 0.474761 0.630207)
(make_list 1.548 0.3950 95 1.39946))
(add_node 0
(make_list 14.1057 -7.02974 -4.93581)
(make_list 0.328045 -0.927766 -0.313138)
(make_list -1.07768 0.445073 -1.16865)
(make_list 0.525595 -0.247645 0.469556))
(add_node 0
(make_list -22.5002 -26.8682 -1.04586)
(make_list 0.441015 -0.953341 -0.422396)
(make_list -0.877271 1.4498 -1.52976)
(make_list -0.078343 1. 29278 0.0184263))
(add_node 0
(make_list 7.56711 -0.0938968 2.72102)
(make_list 0.115133 0.638264 -0.078159)
(make_list 1.42447 1.13761 -1.17828)
(make_list 1.53793 -0.925061 -0.685163))
```

Figure 7-3: A save file saving the state of a simulation of five nodes. The file has been formatted for easier readability.

7.2 Graphical Interface

This section describes the graphical user interface, which provides a graphical display of simulation state and allows a simple means of performing many common simulator operations. The section first details the software systems used to implement the graphical interface. It then explains how a user can interact with the interface through menu, button, and mouse commands. Finally, it describes the lower-level primitives through with the graphical interface interacts with the command interpreter.

The graphical interface provides an easier-to-use front end to the command interpreter. It gives users a graphical display of the state of the simulation and a convenient means of performing common operations such as loading and saving simulations, querying and adjusting simulation constants, and adding and deleting nodes. Figure 7-4 shows a screen snapshot of the graphical interface. Like the rest of the user interface, the graphical interface is written in C. The window code is written for compatibility with the X Window System, Version 11 Release 5 or later [37]. In order to assist with design of buttons and pop-up windows, the graphical interface uses the Motif toolkit version 1.2 [80]. Rendering routines were written to use OpenGL [91]. The simulator has also been run with the free libraries Lesstif [111] in place of Motif and Mesa [83] in place of OpenGL, however, promoting portability.

The user interface provides a means of running several common commands. The `add` and `delete` buttons call up windows to allow easy adding or deleting of nodes. The `add` window allows users either to add individual nodes with user-specified positions, rotations, velocities, and angular velocities, or to add groups of nodes with randomly selected initial states. The `delete` window provides the options of deleting nodes by index, deleting all nodes in a region, or deleting a specific "selected" node.

The `load` and `save` buttons provide a convenient interface for loading files and saving simulation data. Both are implemented with the Motif `FileSelectionDialog` widget (or its Lesstif equivalent), allowing users to search directories and select filenames with a pointing device or to enter them manually. Because saved program files are output in the form of interpreter code that recreates the simulation state, a single

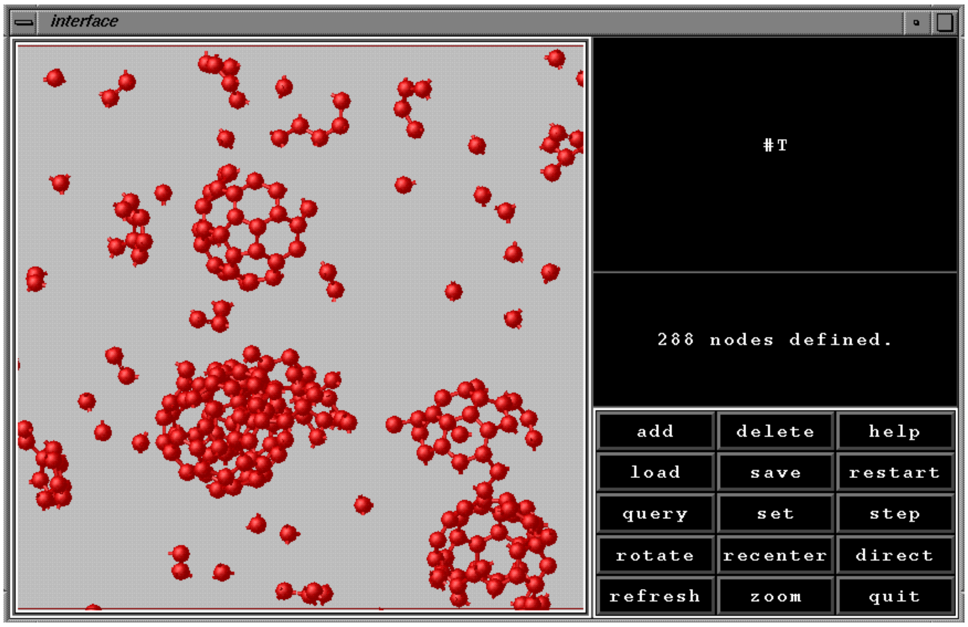

Figure 7-4: A screen snapshot of the graphical user interface. The interface contains a rendering window, which displays the visible state of the simulation; text boxes for providing some information about simulations; and buttons for activating various functions of the interface.

load interface serves for saved simulations and user-created program files.

The `rotate`, `recenter`, and `zoom` buttons produce dialog boxes for manipulating the graphics display. The `rotate` button pops up a menu for rotating an arbitrary number of degrees about the X, Y, or Z axis. The `recenter` button allows the user to center the display on an arbitrary point in three-space. The `zoom` button allows users to zoom in or out by an arbitrary factor.

The `direct` button pops up a window for direct interaction with the command interpreter. The window consists of an input field in which users can type arbitrary interpreter-level commands for evaluation and an output field, which displays the value returned by the command. This window allows users a convenient means of accessing the interpreter when the other graphical interface tools are insufficiently versatile, but loading a file of commands would be too cumbersome.

The remaining buttons provide assorted functions for interacting with the simulator. The `restart` button is used to clear all data and establish a new connection with the numerical portion of the code. The `refresh` button redraws the graphics display. The `query` button provides information on the current state of a "selected" node. The `set` button produces a window that allows examination and modification of several simulation constants. The `help` button produces a quick description of the various graphical interface commands. Similar help buttons have been added to all major sub-windows that the other buttons pop up, printing instructions on their use and listing buttons and keyboard commands to which they respond. Finally, the `quit` button causes the user interface portion of the simulator to disconnect from the numerical methods and exit.

In order to allow rapid manipulation of the user's view of the workspace, keyboard shortcuts for translations, rotations, and zooms have been added to supplement the pop-up window interfaces to those commands. The following keyboard commands are recognized:

- D: translate along the negative X axis by 5 times the current zoom factor
- F: translate along the positive X axis by 5 times the current zoom factor

- S: translate along the negative Y axis by 5 times the current zoom factor
- W: translate along the positive Y axis by 5 times the current zoom factor
- E: translate along the negative Z axis by 5 times the current zoom factor
- R: translate along the positive Z axis by 5 times the current zoom factor
- B: rotate counter-clockwise about the positive X axis by five degrees
- N: rotate clockwise about the positive X axis by five degrees
- G: rotate counter-clockwise about the positive Y axis by five degrees
- H: rotate clockwise about the positive Y axis by five degrees
- T: rotate counter-clockwise about the positive Z axis by five degrees
- Y: rotate clockwise about the positive Z axis by five degrees
- C: zoom in by a factor of 1.25
- V: zoom out by a factor of 1.25
- A: recenter the display on the selected node, if any

In addition, users can “select” a node by a mouse click on any sphere that is part of this node. Selection provides a convenient means to specify a node for such purposes as querying its current state or deleting it. When a node is selected, it is drawn in a highlighted version of its standard color, making selection useful for tracking a particular node during the progress of a simulation. A node can be de-selected by selecting another node or clicking on any part of the screen not containing a node.

Commands executed through the graphical interface are converted into commands in the control language that are passed directly to the command interpreter. Commands treated in this way include those executed by the various pop-up windows as well as the keyboard and mouse commands. Most of these commands call interpreter-level hooks whose default behaviors are described in appendix A. Their behavior can be redefined, however, by replacing their hooks with user-defined procedures, allowing customization of the graphical interface when desired.

The user interface handles the overall rendering process primarily through two primitive interpreter commands: `copy_to_render_buffer` and `render_from_buffer`. The `copy_to_render_buffer` command creates a data structure recording the information needed to draw spheres for all nodes, cylinders for all binding sites, and lines showing the edges of the workspace. The procedure is passed information retrieved from the numerical routines about what single nodes are present at that time and what their positions and rotations are. Based on this information, it records in a global data structure the specifications for spheres representing each single node and cylinders representing each edge. It similarly stores the data needed to draw the twelve lines marking the boundaries of the workspace. It is also passed the index of a "selected" node, if any, which it records. The second command, `render_from_buffer`, renders all objects specified by the data structure. Another pair of commands, `current_axes` and `current_zoom`, control the position and magnification from which the workspace is viewed. The axes and zoom can be adjusted between calls to `copy_to_render_buffer` and `render_from_buffer`, allowing the same workspace to be viewed from multiple positions without the simulator having to recompute the positions of all graphics objects in the workspace. This capability is a change from the previous version of the simulator, in which graphics control was handled by interpreter-level code calling simple graphics primitives. One significant advantage of the new arrangement is that `copy_to_render_buffer`, because it is a single hard-coded function, can execute substantially faster than the interpreter-level functions that originally implemented the same operations in terms of simpler primitives. In addition, allowing adjustment of the angle and zoom without having to perform the costly operations of recalculating the positions of graphic objects allows for faster manipulation of the display, improving ease of use. In fact, the improved speed allowed by this modification was essential to making the new keyboard commands for graphics manipulation practical. These benefits, however, come at the cost of a loss of versatility, in that the rendering code cannot easily be customized without editing and recompiling the simulator source.

If versatility is absolutely needed, it is still available; graphics primitives for directly drawing lines, cylinders, and spheres are still present in the control language, although not used by default, so a user could override any of the new primitive routines with a variant of the old interpreter-level code and then modify that. Doing so will, however, eliminate the performance advantage of higher-level rendering primitives.

7.3 Network Interface

This section describes one final component of the user interface: the code that transmits user commands to the parallel numerical routines and receives back information from those routines. The user interface uses a simple series of commands and queries to communicate with the parallel numerical routines via a TCP socket. The commands are executed by various primitive procedures of the control language, described in appendix A, which trigger the lower-level network interface commands. Each supported command is referenced by a number that is transmitted through the socket from the user interface to the numerical routines to inform them of the next command to be executed. After transmitting the number, the user interface may transmit more data and may wait for reply data from the numerical routines, depending on the command.

Data is sent and received as a series of raw bytes, with the two programs converting between the byte stream and the encoded data in accordance with a hard-coded protocol for each command. The full list of commands and their associated protocols is described in appendix B. In order to allow transmission between machines using different byte orders, data is transmitted in a canonical "network byte order," in which high-order bytes precede low order bytes for each multi-byte value transmitted. Floating point formats are assumed to be consistent between architectures in use modulo byte ordering. While this assumption limits portability, in practice all architectures used by this project are compliant with the IEEE 754 standard [48] for floating point numbers.

7.4 Evaluation

This section evaluates the effectiveness of the user interface described in the remainder of the chapter. The simulator at present has no "user community," nor does it appear likely it will in the near future, due to its specialized nature and the complexity of the model. As I am the only person who has extensive experience using the simulator, the evaluation necessarily consists primarily of anecdotal demonstrations of how aspects of the interface design have been useful to me. These anecdotes relate how features of both the command interpreter and the graphical interface are crucial to performing many of the applications described in chapter 3. On the basis of these anecdotes, this section concludes that the user interface design does not make the simulator usable by a general audience, as might be hoped, but does provide a good means of maintaining high versatility and ease-of-use for a user such as myself who has extensive experience with the model.

The graphical front end has been useful primarily in studying simulation progress and examining assembled structures. The ability to view the state of a simulation and to translate, rotate, and zoom the display easily provides a simple means of examining the progress of a dynamic process or the state of an assembled structure. In the absence of an easily-manipulated graphical display, such tasks would be considerably more difficult. These abilities are crucial to verifying the growth patterns of the different shell geometries described in section 3.1. They are also important for examining structures produced by all of the non-virus systems examined in section 3.5. Such capabilities are particularly valuable in detecting malformed structures, a task essential to the parameter variation experiments of section 3.2 and the growth disruption experiments of section 3.4. While detecting malformed structures is somewhat time-consuming even by visual inspection, it would be even more difficult to automate and nearly impossible to accomplish by inspection of raw numerical data. The applications therefore show how a graphical display and an interface allowing its easy manipulation can save considerable effort when one uses a simulation model such as the one described in this thesis.

The applications similarly demonstrate that the use of a programmable interpreted control language has succeeded in providing a versatile and powerful model adaptable to many sorts of tasks beyond those envisioned when the simulator was first designed. Code written in the simulator control language is used to automate running simulations and periodically saving the simulation state in the experiments on parameter variations described in section 3.2, the experiments comparing models for nucleation-limited growth described in setion 3.3, and the experiments on disrupting growth described in section 3.4. A more specialized control construct is used to automate data gathering for the actin experiment described in section 3.5.1. Another application of the interpreter is implementing the cooling schedule used by the borosilicate glass simulations, as described in section 3.5.2. The protein aggregation simulations of section 3.5.3 also require specialized interpreter code to create multi-subunit model proteins, giving the subunits contributing to a protein the correct relative positions and rotations before joining the pieces into a complete model protein. Possibly the most complicated use of interpreter-level control constructs is for enforcing a relatively constant free monomer concentration for the experiments on nucleation-limited growth described in section 3.3. As the above applications demonstrate, specialized code written in the control language greatly extends the versatility of the model and the ease of applying it to complicated tasks.

The user interface is not, however, adequate to allow a general user with minimal training to use the simulator. At present, there is no easy way to create a rules file specifying the parameters for a given simulation. A user therefore must still have some knowledge of the control language. Furthermore, the user interface cannot obviate the need for knowledge of how to select and fine-tune simulation parameters in order to give desired assembly behaviors and promote rapid convergence of the numerical methods. While the user interface does promote ease-of-use for many common operations, a user must still be knowledgeable regarding the simulation model, the simulator's control language, and its numerical methods in order to use it effectively. I had originally hoped the simulator would occupy a position in a research laboratory much like that of a desktop light microscope, a simple research tool that almost any biologist can be expected to operate. Instead, it occupies a position more like that of an electron microscope, a highly specialized tool that can be extremely useful, but that requires an operator specially trained in its use.

The evidence in this section suggests that the user interface satisfies the goal of creating a versatile tool that is generally easy to use, but only within the constraint that it be operated by knowledgeable, experienced users. The interface therefore falls short of its original goal of providing a general audience of biologists with a toolkit they can use in their work. It does, however, succeed in providing an experienced user with a simple means of performing its most common operations, such as those involved with manipulating the graphics display. It also succeeds in creating a powerful and versatile model that can be adapted to a variety of tasks and can simplify many operations that would be tedious to perform with a more rigid model. This interface design is therefore well-suited to a simulation model that is intended to be used only by those thoroughly versed in its operation.

Chapter 8

Conclusions and Discussion

This chapter concludes the body of the thesis. It begins with a brief summary of the material covered in the remainder of the document. It also presents some open problems that might be pursued in further advancing this line of research. Finally, it discusses the major lessons that I believe can be learned from the experience of developing and applying this simulation model.

8.1 Summary

This thesis has presented a framework for the efficient simulation of self-assembly phenomena. It has described the local rules dynamics model for self-assembly simulation. It has also discussed several applications of the model. Finally, it has presented details on an implementation of the local rules dynamics model as part of a versatile simulation tool.

The thesis described a simulation model based on local rules principles. The model relies on an abstraction of binding forces into local rules-based bond patterns as a means of producing computationally tractable simulations of self-assembly dynamics. It combines this abstraction with an off-lattice Newtonian dynamics model to provide versatile simulations of self-assembly phenomena. Within this framework, the model specifies a representation of the components of a self-assembly system, including a model of assembly subunits, rules for interactions between them, and a simulated environment.

The thesis also described several applications of the model. Simulation models were described for several forms of shells that build by various mechanisms. An application examining the effects of parameter variations on assembly behavior was described. Another application comparing different models for the source of observed shell assembly kinetics was also examined. In addition, several smaller applications were presented to demonstrate the value of the simulation model to different kinds of systems and different sorts of questions.

Finally, the thesis described an implementation of the model. Two parallel numerical methods were presented for numerical integration of particle state over time. The methods were analyzed and compared in practice, revealing that each has some advantages and disadvantages for the problem. A discrete algorithm for locating binding partners was also presented and analyzed. In addition, two variants of a discrete parallel algorithm for collision detection were described and examined theoretically. A programmable, graphical user interface for the simulator was also presented.

8.2 Open Problems and Future Work

This section discusses several areas in which this project might be continued. It first considers avenues for improvement in the physical model. Next, it looks at how the user interface might be made more accessible. It also considers how the numerical methods might be made more efficient or robust. Finally, it examines avenues for future work in applications, where local rules dynamics simulation can be applied to a greater variety of problems or in greater depth to the problems already considered.

The physical model might be improved in several ways. For some applications, more realistic bond potentials might be useful. A facility for allowing user-defined potentials, possibly from some restricted set of functions, might therefore be worthwhile. Realism might also be improved by accounting for bond stress when calculating probabilities of forming and breaking bonds. Accounting for bond stresses in this way could be particularly important in accurately measuring the likelihood of producing malformed structures, which typically have many stressed binding interactions. The Brownian motion mechanism used is also not satisfactory, but it remains unclear how to achieve realism in both high level behavior and the low-level mechanisms of actual Brownian motion. Paradoxically, the physical model also requires simplification. At present, the large number of adjustable parameters make it too difficult to develop simulations and to know that the results can be trusted. While the need to adjust many parameters likely to some degree reflects the fact that actual self-assembly systems must be finely tuned to avoid the various pitfalls they face, it would nonetheless be useful to determine what is actually essential to giving correct self-assembly behavior so that the need to perform empirical manipulation of simulation parameters could be kept to a minimum.

The user interface could also be improved. While running a simulation through the graphical interface is generally simple, designing one and writing the rules file for it can be difficult. A suite of graphical tools for designing simulation parameters and assembly subunits might be essential to making the simulator usable by a wider audience. Interactions with the interpreter could also be simplified by the addition of syntactic sugar for programming convenience and the development of better debugging facilities. A strategy that might be of greater general use would be rewriting the user interface as an extension to a more powerful and well-developed general-purpose interpreted language, rather than using a custom language developed exclusively for this application. Allowing a genuine user community to evaluate the interface and suggest improvements, rather than relying on personal experience, would also contribute significantly to the interface's development.

It also appears likely that the numerical methods could be made faster and more robust. Better methods of step-size selection might be especially useful in increasing execution speed. Stability was ignored in the original design on the assumption that the Brownian motion model would control instabilities; in practice, though, it appears that achieving stability can require unreasonably high damping or impractically small time steps. This problem is particularly acute for the waveform-relaxation method. Developing a method that was significantly more stable, even at the cost of some additional work, might therefore be useful in increasing robustness. One strategy, currently under investigation by Vinay Pulim, is to side-step the stability problem by replacing the continuous numerical methods entirely with a discrete-event method for updating particle positions. The hard-sphere problem has been treated as a purely discrete event problem of stepping between particle collisions [2], and several variants on that method have been developed to improve run time [33, 87, 66, 68, 60]. More efficient use of parallelism is a concern no matter what method is used. My experience also suggests, however, that efficient parallelism should not come at the expense of fast serial run-time. Learning how better to balance the relative demands of desktop computing and super-computing remains an important question, particularly as current hardware trends increasingly cause the two to converge.

Probably the greatest avenue for future work is in developing new applications. Larger shells would be useful, particularly T=7 because it is the best-studied icosahedral geometry where issues of kinetics are concerned. In addition, it would be helpful to develop simulation models of scaffolding proteins and other molecules, such as portal proteins and nucleic acids, that can be important to assembly. Several of the applications presented are in only their earliest stages and serve primarily as demonstrations; much more work is needed to determine reliably where local rules dynamics simulations will be applicable and to generate useful results. Probably the greatest challenge for all of the applications will be bringing them closer to laboratory experimental work, both by using simulations to create testable predictions that can be confirmed in the laboratory and by finding better ways to incorporate existing experimental data into simulation models.

8.3 Conclusions

This final section presents some closing remarks. In it, I discuss the conclusions I have drawn from my experience with this project. These conclusions touch on the feasibility and value of simulation models like the one described in this thesis and on what remains to be done in order to fully realize their potential.

A central conclusion of this project is that simulation of the dynamics of even complicated self-assembly phenomena is feasible. We have seen that a complex and general model can be constructed and can execute within a reasonable amount of time. We have also seen that this model is applicable to a variety of problems. Because no comparable self-assembly simulation system has to my knowledge been constructed, a demonstration of its feasibility should be useful in gauging the potential of future attempts to construct similar simulation tools.

We have also seen that such a simulator is useful in examining previously unexplored aspects of the systems it simulates. As I have argued, a simulation model provides capabilities that are not available through other means. The applications have shown how such capabilities as the ability to adjust simulation parameters precisely and to compare the behavior of hypothetical models provide a means of examining issues that are not practically accessible to laboratory experiments or purely theoretical analyses. Simulation can therefore provide a valuable check on theory and an important complement to laboratory experimentation as part of a broader research program.

I can, however, also conclude that much remains to be done to realize the full potential of such models. In the previous section, I discussed several open problems that must be addressed if simulation tools of the sort I describe here are going to be sufficiently fast, versatile, and accessible to be widely used by those most likely to benefit from them. Many of these issues are of concern beyond this specific project. For example, better solutions to the problems of user interface design and algorithms and numerical methods for particle simulations are likely to be applicable to many related problems.

My primary conclusion regarding the self-assembly systems simulated is that self-assembly appears to be an extremely difficult design problem. My experience with running simulations, including the successful experiments presented in this work and the many preparatory simulations and dead ends that led to them, suggests that correctly assembling systems occupy a small part of a complicated parameter space. We have observed many pitfalls that a successful self-assembly system must avoid:

nucleating too quickly to complete shells or too slowly to begin them, having bonds too weak to make growth kinetically favorable or too strong to allow it to escape kinetic traps, or being too tolerant of small structural deviations and thus ending up with malformed shells or too intolerant of structural deviations and thus being unable to grow at a reasonable rate. While the details of these problems may be simulator artifacts, it nonetheless seems likely that self-assembly systems must genuinely navigate around problems such as these, often making complicated trade-offs between such factors as stability of shells, growth rate, and fidelity of assembly. Simulation work provides a crucial means of gaining an appreciation for the tremendous difficulty involved in making a fast, reliable self-assembly system. Such an appreciation will be particularly useful for biologists who, because they see only nature's highly optimized systems, are likely not to appreciate the obstacles evolution has had to overcome. This appreciation will be important in understanding the mechanisms biological self-assembly systems have evolved to avoid the pitfalls they encounter, as such mechanisms become known. Designers of novel self-assembly systems will similarly benefit from an understanding of the problems they will face and how evolutionary mechanisms have solved them in nature.

A final note I wish to emphasize is the critical importance this project has demonstrated of improving avenues for interdisciplinary work. The field of mathematical modeling and simulation is inherently interdisciplinary, but this project suggests that, even so, efforts must be made to ensure that the various fields involved work well together. The input of natural scientists into a physical model is essential if it is to be expected to capture the system it models sufficiently well to be relevant to problems of current interest to scientific researchers. Moreover, such input is necessary to developing applications that are both useful and technically sound. The interdisciplinary work required to tackle this kind of problem successfully, however, extends even to different disciplines within computer science and mathematics. As my experience shows, addressing the needs of a real-world problem can require drawing on a diversity of techniques that are unlikely to obey the narrow bounds of intradisciplinary specialties of computer science and applied mathematics. This project has required techniques from fields ranging from graphics and user interface design to algorithms and numerical analysis. Any of these areas in isolation likely could have been better handled by appropriate specialists, had they been available. Creating the organizational conditions necessary to bring together the diversity of expertise ideally available to a physical simulation project is arguably a more important open problem than any of the technical issues addressed in this thesis.

Bibliography

[1] M. Agarwal, M. Arthur, R. D. Arbeit, and R. Goldstein. Regulation of icosahedral virion capsid size by the *in vivo* activity of a cloned gene product. *Proceedings of the National Academy of Science U.S.A.*, 87:2428–2432, 1990.

[2] B. J. Alder and T. E. Wainwright. Studies in molecular dynamics. I. general method. *Journal of Chemical Physics*, 31:459–466, 1959.

[3] E. Alm and D. Baker. Matching theory and experiment in protein folding. *Current Opinion in Structural Biology*, 9:189–196, 1999.

[4] T. S. Baker, W. W. Newcomb, N. H. Olson, L. M. Cowsert, C. Olson, and J. C. Brown. Structures of bovine and human papillomaviruses: analysis by cryo-electron microscopy and three-dimensional image reconstruction. *Biophysical Journal*, 60:1445–1456, 1991.

[5] B. Berger, J. A. King, R. Schwartz, and P. W. Shor. Local rule mechanism for selecting icosahedral shell geometry. *Discrete Applied Mathematics*, 2000. In press.

[6] B. Berger and P. W. Shor. On the mathematics of virus shell assembly. MIT Laboratory for Computer Science Technical Memo 519, 1994.

[7] B. Berger and P. W. Shor. Local rule switching mechanism for viral shell geometry. MIT Laboratory for Computer Science Technical Memo 527, 1995.

[8] B. Berger, P. W. Shor, L. Tucker-Kellogg, and J. King. Local rule-based theory of virus shell assembly. *Proceedings of the National Academy of Science U.S.A.*, 91:7732–7736, 1994.

[9] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, 1995. Association for Computing Machinery.

[10] B. Brooks, B. Buccoleri, D. Olafson, D. States, S. Swaminathan, and M. Karplus. CHARMM: a program for macromolecular energy minimization and dynamics calculation. *Journal of Computational Chemistry*, 4:187–217, 1983.

[11] C. L. Brooks III and M. Karplus. Solvent effects on protein motion and protein effects on solvent motion. *Journal of Molecular Biology*, 208:159–181, 1989.

[12] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 106–112, 1977.

[13] D. L. D. Caspar. Movement and self-control in protein assemblies: Quasi-equivalence revisited. *Biophysical Journal*, 232:103–138, 1980.

[14] D. L. D. Caspar and A. Klug. Physical principles in the construction of regular viruses. *Cold Spring Harbor Symposium on Quantitative Biology*, 27:1–24, 1962.

[15] G. M. Crippen. Easily searched protein folding potentials. *Journal of Molecular Biology*, 260:467–475, 1996.

[16] E. De Bernardez-Clark and G. Georgiou. *Protein Refolding*, chapter 1, pages 1–19. American Chemical Society, Washington, DC, 1991.

[17] L. R. De Young, A. L. Fink, and K. A. Dill. Aggregation of globular proteins. *Accounts of Chemical Research*, 26:614–620, 1993.

[18] J. M. Delaye and D. Ghaleb. Molecular dynamics simulation of a nuclear waste glass matrix. *Materials Science and Engineering*, B37:232–236, 1996.

[19] J. M. Delaye and D. Ghaleb. Molecular dynamics simulation of $SiO_2 + B_2O_3 + Na_2O + ZrO_2$ glass. *Journal of Non-Crystalline Solids*, 195(3):239–248, 1996.

[20] J. M. Delaye and D. Ghaleb. Molecular dynamics simulation of displacement cascades in matrix of French light water reactor waste containment glass. *Radiation Effects and Defects in Solids*, 142(1-4):471–481, 1997.

[21] J. M. Delaye and D. Ghaleb. Molecular dynamics simulation of low-energy atomic displacement cascades in a simplified nuclear glass. *Journal of Nuclear Materials. Journal des Materiaux Nucleaires*, 244(1):22–28, 1997.

[22] J. M. Delaye and D. Ghaleb. Molecular dynamics simulation of the trajectory of a recoil nucleus in a simplified nuclear glass. In *Materials Research Society Symposia Proceedings*, 1997.

[23] J. M. Delaye and D. Ghaleb. Molecular-dynamics simulation of ballistic damages caused by an energetic recoil nucleus on a nuclear glassy network. In *Materials Research Society symposia proceedings*, 1998.

[24] J. M. Delaye and D. Ghaleb. Molecular dynamics analysis of volume change in a nuclear glass model subjected to a displacement cascade. *Nuclear Instruments and Methods in Physics Research B*, 153(1):157–162, 1999.

[25] J. M. Deutch and I. Oppenheim. Molecular theory of Brownian motion for several particles. *Journal of Chemical Physics*, 54:3547–3555, 1971.

[26] A. Di Nola, D. Roccatana, and H. J. C. Berendsen. Molecular dynamics simulation of the docking of substrates to proteins. *Proteins: Structure, Function, and Genetics*, 19:174–182, 1994.

[27] K. A. Dill. Theory for the folding and stability of globular proteins. *Biochemistry*, 24:1501–1509, 1985.

[28] N. V. Doan. Etude en dynamique moléculaire des défauts créés par irradiation dans la silice viteuse. *Philosophical Magazine*, A49:683–687, 1984.

[29] S. Doniach and P. Eastman. Protein dynamics simulations from nanoseconds to microseconds. *Current Opinion in Structural Biology*, 9:157–163, 1999.

[30] A. I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5:6–9, 1956.

[31] W. Earnshaw and J. King. Structure of phage P22 coat protein aggregates formed in the absence of the scaffolding protein. *Journal of Molecular Biology*, 126:721–747, 1978.

[32] A. Elofsson and L. Nilsson. How consistent are molecular dynamics simulations? *Journal of Molecular Biology*, 233:766–780, 1993.

[33] J. J. Erpenbeck and W. W. Wood. *Statistical Mechanics, Part B*, chapter 1. Molecular dynamics simulation of flow past a plate, pages 1–40. Plenum Press, New York, 1977.

[34] M. L. Galisteo and J. King. Conformational transformations in the protein lattice of phage P22 procapsids. *Biophysical Journal*, 65:227–235, 1993.

[35] N. Gō and H. Taketomi. Respective roles of short- and long-range interactions in protein folding. *Proceedings of the National Academy of Sciences USA*, 75:559–563, 1978.

[36] J. M. Grimes, J. N. Burroughs, P. Gouet, J. M. Dirpose, R. Malby, S. Ziéntara, P. P. C. Mertens, and D. I. Stuart. The atomic structure of the bluetongue virus core. *Nature*, 395:470–478, 1998.

[37] The Open Group. X window system, version 11, January 1998. see **http://www.x.org/**.

[38] P. Gupta, C. K. Hall, and A. C. Voegler. Effect of denaturant and protein concentrations upon protein refolding and aggregation: a simple lattice model. *Protein Science*, 7:2642–2652, 1998.

[39] J. M. Haile and J. P. O'Connell. Internal structure of a model micelle via computer simulation. *Journal of Physical Chemistry*, 88:6363–6366, 1984.

[40] W. R. Hamilton. *Lectures on Quaternions: Containing a Systematic Statement of a New Mathematical Method; of Which the Principles Were Communicated in 1843 to the Royal Irish Academy; and Which Has Since Formed the Subject of Successive Courses of Lectures, Delivered in 1848 and Subsequent Years, in the Halls of Trinity College, Dublin: With Numerous Illustrative Examples.* Hodges and Smith, 1853.

[41] M.-H. Hao and H. A. Scheraga. Designing potential energy functions for protein folding. *Current Opinion in Structural Biology*, 9:184–188, 1999.

[42] C. L. Hill, T. F. Booth, B. V. V. Prasad, J. M. Grimes, P. P. C. Mertens, G. C. Sutton, and D. I. Stuart. The structure of a cypovirus and the functional organization of dsRNA viruses. *Nature Structural Biology*, 6(6):565–568, 1999.

[43] T. L. Hill. *Statistical Mechanics.* Dover, 1956.

[44] L. W. Hobbs, C. E. Jesurum, A. Coventry, V. Pulim, R. Schwartz, and B. Berger. Towards a topological description of poorly-crystalline networks. In *Advances in Materials for the 21st Century: The 1999 Julia R. Weertman Symposium*, 1999.

[45] L. W. Hobbs, C. E. Jesurum, V. Pulim, and B. A. Berger. Local topology of silica networks. *Philosophical Magazine A*, 78:679–711, 1998.

[46] L. W. Hobbs, A. N. Sreeram, C. E. Jesurum, and B. Berger. Structural freedom, topological disorder, and the irradiation-induced amorphization of ceramic structures. *Nuclear Instruments and Methods in Physics Research B*, 116:18–25, 1996.

[47] N. Horton and M. Lewis. Calculation of the free energy of association for protein complexes. *Protein Science*, 1:169–181, 1992.

[48] Institute of Electrical and Electronics Engineers. IEEE standard for binary floating-point arithmetic, 1985. ANSI/IEEE Std 754-1985.

[49] S. Istrail, R. Schwartz, and J. King. Lattice simulations of aggregation funnels for protein folding. *Journal of Computational Biology*, 6(2):143–162, 1999.

[50] C. E. Jesurum, V. Pulim, and L. W. Hobbs. Modeling the cascade amorphization of silicas with local-rules reconstructions. *Nuclear Instruments and Methods in Physics Research B*, 141:25–34, 1998.

[51] C. E. Jesurum, V. Pulim, and L. W. Hobbs. Topological modeling of amorphized tetrahedral ceramic network structures. *Journal of Nuclear Materials*, 253:87–103, 1998.

[52] W. Kabsch and J. Vandekerckhove. Structure and function of actin. *Annual Review of Biophysical and Biomolecular Structure*, 21:49–76, 1992.

[53] M. Karplus and G. A. Petsko. Molecular dynamics simulations in biology. *Nature*, 347:631–639, 1990.

[54] M. Karplus and D. L. Weaver. Protein-folding dynamics. *Nature*, 260:404–406, 1976.

[55] M. Karplus and D. L. Weaver. Diffusion-collision model for protein folding. *Biopolymers*, 18:1421–1437, 1979.

[56] I. Katsura. Structure and inherent properties of the bacteriophage lambda and head shell IV: Small-head mutants. *Journal of Molecular Biology*, 171:297–317, 1983.

[57] H. J. Kinosian, L. A. Selden, J. E. Estes, and L. C. Gershman. Actin filament annealing in the presence of ATP and phalloidin. *Biochemistry*, 32:12353–12357, 1993.

[58] A. Kitao and N. Go. Investigating protein dynamics in collective coordinate space. *Current Opinion in Structural Biology*, 9:164–169, 1999.

[59] E. D. Korn, M.-F. Carlier, and D. Pantaloni. Actin polymerization and ATP hydrolysis. *Science*, 238:638–644, 1987.

[60] A. T. Krantz. Analysis of an efficient algorithm for the hard-sphere problem. *ACM Transactions on Modeling and Computer Simulation*, 6(3):185–209, 1996.

[61] J. K. Krueger, A. M. Stock, C. E. Schutt, and J. B. Stock. *Protein Folding*, chapter 14: Inclusion bodies from proteins produced at high levels in Escherichia coli. American Association for the Advancement of Science, Washington, DC, 1990.

[62] I. Lafontaine and R. Lavery. Collective variable modelling of nucleic acids. *Current Opinion in Structural Biology*, 9:170–176, 1999.

[63] R. G. Larson, L. E. Scriven, and H. T. Davis. Monte Carlo simulation of model amphiphile-oil-water systems. *Journal of Chemical Physics*, 83:2411–2420, 1985.

[64] E. Lelarasmee, A. Rusehli, and A. Sangiovanni-Vincentelli. The waveform relaxation methods for time domain analysis of large scale integrated circuits. *IEEE Transactions on CAD IC Systems*, 3:131–145, 1982.

[65] M. Li, M. K. Lyon, and R. L. Garcea. *In vitro* phosphorylation of the polyomavirus major capsid protein VP1 on serine 66 by casein kinase II. *Journal of Biological Chemistry*, 270:26006–26011, 1995.

[66] B. D. Lubachevsky and F. H. Stillinger. Geometric properties of random disk packings. *Journal of Statistical Physics*, 60:561–583, 1990.

[67] H. P. Luhn. IBM internal memorandum, 1953.

[68] M. Marin, D. Risso, and P. Cordero. Efficient algorithms for many-body hard particle molecular dynamics. *Journal of Computational Physics*, 109:306–317, 1993.

[69] F. A. O. Marston. The purification of eukaryotic polypeptides synthesized in *Escherichia coli*. *Biochemical Journal*, 240:1–12, 1986.

[70] O. J. Marvik, P. Sharma, T. Dokland, and B. H. Lindqvist. Bacteriophage P2 and P4 assembly: alternate scaffolding proteins regulate capsid size. *Virology*, 200:702–714, 1994.

[71] C. J. Marzec and L. A. Day. Pattern formation in icosahedral virus capsids: the papova viruses and nudaurelia capensis β virus. *Biophysical Journal*, 65:2559–2577, 1993.

[72] Massachusetts Institute of Technology. *Cilk 2.0 Reference Manual*, July 1995. see **http://supertech.lcs.mit.edu/cilk**.

[73] Massachusetts Institute of Technology. *Cilk 5.2 Reference Manual*, July 1998. see **http://supertech.lcs.mit.edu/cilk**.

[74] R. H. Merson. An operational method for the study of integration processes. In *Proceedings of the Symposium on Data Processing*, pages 110–1–110–25. Weapons Research Establishment, 1957.

[75] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[76] L. Montross, S. Watkins, R. B. Moreland, H. Mamon, D. L. D. Caspar, and R. L. Garcea. Nuclear assembly of polyoma virus capsids in insect cells expressing the major capsid protein VP1. *Journal of Virology*, 65:4991–4998, 1991.

[77] D. Muir. Simulated construction of viral protein shells. Bachelor's Thesis. Massachusetts Institute of Technology, 1994.

[78] J. Novotný, R. Bruccoleri, and M. Karplus. An analysis of incorrectly folded protein models. *Journal of Molecular Biology*, 177:787–818, 1984.

[79] F. Oosawa and M. Kasai. A theory of linear and helical aggregations of macromolecules. *Journal of Molecular Biology*, 4:10–21, 1962.

[80] The Open Group, 11 Cambridge Center, Cambridge, MA 02142. *OSF/Motif Programmer's Manual*, revision 1.2 edition, August 1992. see **http://www.opengroup.org/motif/**.

[81] S. Y. Patro and T. M. Przybycien. Simulations of kinetically irreversible protein aggregate structure. *Biophysical Journal*, 66:1274–1289, 1994.

[82] S. Y. Patro and T. M. Przybycien. Simulations of reversible protein aggregate and crystal structure. *Biophysical Journal*, 70:2888–2902, 1996.

[83] B. Paul. Mesa 2.6. Available from **iris.ssec.wisc.edu** via anonymous ftp in directory **pub/Mesa**, 1993.

[84] P. E. Prevelige, J. King, and J. L. Silva. Pressure denaturation of the bacteriophage P22 coat protein and its entropic stabilization in icosahedral shells. *Biophysical Journal*, 66:1631–1641, 1994.

[85] P. E. Prevelige, D. Thomas, and J. King. Nucleation and growth phases in the polymerization of coat and scaffolding subunits into icosahedral procapsid shells. *Biophysical Journal*, 64:824–835, March 1993.

[86] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Massachusetts Institute of Technology, 1998.

[87] D. C. Rapaport. The event scheduling problem in molecular dynamic simulation. *Journal of Computational Physics*, 34:184–201, 1980.

[88] I. Rayment, T. S. Baker, and D. L. D. Caspar. Polyoma virus capsid structure at 22.5Å resolution. *Nature*, 295:110–115, 1982.

[89] V. S. Reddy, H. A. Geising, R. T. Morton, A. Kumar, C. B. Post, C. L. Brooks, and J. E. Johnson. Energetics of quasiequivalence: computational analysis of protein-protein interactions in icosahedral viruses. *Biophysical Journal*, 74:546–558, 1998.

[90] L. F. Richardson. The approximate arithmetical solution by finite differences of physical problems including differential equations, with an application to the stresses in a masonry dam. *Philosphical Transactions A*, 210:307–354, 1910.

[91] D. Rogelberg and J. C. Fullagar, editors. *OpenGL Reference Manual*. Silicon Graphics, Inc., 1995.

[92] D. M. Salunke, D. L. D. Caspar, and R. L. Garcea. Self-assembly of purified polyomavirus capsid protein VP_1. *Cell*, 46:895–904, 1986.

[93] D. M. Salunke, D. L. D. Caspar, and R. L. Garcea. Polymorphism in the assembly of polyoma virus capsid protein VP_1. *Biophysical Journal*, 56:887–900, 1989.

[94] R. Schwartz. A multi-threaded simulator for the kinetics of virus shell assembly. Master's thesis, Massachusetts Institute of Technology, 1996.

[95] R. Schwartz, R. L. Garcea, and B. Berger. "Local rules" theory applied to polyomavirus polymorphic capsid assemblies. *Virology*, 2000. In press.

[96] R. Schwartz, P. E. Prevelige, Jr., and B. Berger. Local rules modeling of nucleation-limited virus capsid assembly. MIT Laboratory for Computer Science Technical Memo 584, 1998.

[97] R. Schwartz, P. E. Prevelige, Jr., and B. Berger. Local rules modeling of the nucleation and growth phase of icosahedral virus capsid assembly, 1999. Unpublished manuscript.

[98] R. Schwartz, P. W. Shor, P. E. Prevelige, Jr., and B. Berger. Local rules simulation of the kinetics of virus capsid self-assembly. *Biophysical Journal*, 75:2626–2636, 1998.

[99] N. Selve and A. Wegner. Rate of treadmilling of actin filaments *in vitro*. *Journal of Molecular Biology*, 187:627–631, 1986.

[100] P. Sheterline, J. Clayton, and J. C. Sparrow. *Protein Profile: Actin.* Academic Press, London, 1995.

[101] D. Shore, G. Deho, J. Tsipis, and R. N. Goldstein. Determination of capsid size by satellite bacteriophage P4. *Proceedings of the National Academy of Science U.S.A.*, 75:400–404, 1978.

[102] J. Skolnick and A. Kolinski. Dynamic Monte Carlo simulations of a new lattice model of globular protein folding, structure and dynamics. *Journal of Molecular Biology*, 221:499–531, 1991.

[103] M. A. Speed, J. King, and D. I. C. Wang. Polymerization mechanism of polypeptide chain aggregation. *Biotechnology and Bioengineering*, 54(4):333–343, 1996.

[104] A. C. Steven, H. L. Greenstone, F. P. Booy, L. W. Black, and P. D. Ross. Conformational changes of a viral capsid protein: thermodynamic rationale for proteolytic regulation of bacteriophage T4 capsid expansion, co-operativity, and super-stabilization by soc binding. *Journal of Molecular Biology*, 228:870–884, 1992.

[105] H. Taketomi, Y. Ueda, and N. Gō. Studies on protein folding, unfolding and fluctuations by computer simulation. *International Journal of Peptide and Protein Research*, 7:445–459, 1975.

[106] S. Tanaka and H. A. Scheraga. Model of protein folding: Inclusion of short-, medium-, and long-range interactions. *Proceedings of the National Academy of Sciences USA*, 72(10):3802–3806, 1975.

[107] T. Tarnai, Z. Gáspár, and L. Szalai. Pentagon packing models for "all-pentamer" virus structures. *Biophysical Journal*, 69:612–618, 1995.

[108] D. Thirumalai and D. K. Klimov. Deciphering the timescales and mechanisms of protein folding using minimal off-lattice models. *Current Opinion in Structural Biology*, 9:197–207, 1999.

[109] P. A. Thuman-Commike, B. Greene, J. A. Malinski, M. Burbea, A. McGough, W. Chiu, and P. E. Prevelige, Jr. Mechanism of scaffolding-directed virus assembly suggested by comparison of scaffolding-containing and scaffolding-lacking P22 procapsids. *Biophysical Journal*, 76:3267–3277, 1999.

[110] P. A. Thuman-Commike, B. Greene, J. A. Malinski, J. King, and W. Chiu. Role of the scaffolding protein in P22 procapsid size determination suggested by T=4 and T=7 procapsid structures. *Biophysical Journal*, 74:559–568, 1998.

[111] C. Toshok, M. Miers, P. van Helden, J. Fo, R. McMullen, R. Scott, P. Williams, J. Christopher, K. Jensen, J. Gritton, and T. Straumann. LessTif 0.88.9, 1999. see **http://www.lesstif.org**.

[112] C.-J. Tsai, S. Kumar, B. Ma, and R. Nussinov. Folding funnels, binding funnels, and protein function. *Protein Science*, 8:1181–1190, 1999.

[113] C.-J. Tsai, D. Xu, and R. Nussinov. Protein folding via binding, and vice versa. *Folding and Design*, 3:R71–R80, 1998.

[114] W. F. van Gunsteren, H. J. C. Berendsen, and J. A. C. Rullmann. Stochastic dynamics for molecules with constraints: Brownian dynamics of n-alkanes. *Molecular Physics*, 44:69–95, 1981.

[115] F. K. von Gottberg, K. A. Smith, and T. A. Hatton. Stochastic dynamics simulation of surfactant self-assembly. *Journal of Chemical Physics*, 106:9850–9857, 1997.

[116] Y. Wang and D. Shortle. A dynamic bundle of four adjacent hydrophobic segments in the denatured state of staphylococcal nuclease. *Protein Science*, 5:1898–1906, 1996.

[117] K. Watanabe, M. Ferrario, and M. L. Klein. Molecular dynamics study of a sodium octanoate micelle in aqueous solution. *Journal of Physical Chemisty*, 92:819–821, 1988.

[118] W. J. Weber, R. C. Ewing, C. Austen Angell, G. W. Arnold, A. N. Cormack, J. M. Delay, D. L. Griscom, L. W. Hobbs, A. Navrotsky, D. L. Price, A. M. Stoneham, and M. C. Weinberg. Radiation effects in glasses used for immobilization of high-level waste and plutonium disposition. *Journal of Material Research*, 12(8):1946–1978, 1997.

[119] R. Wetzel, editor. *Advances in Protein Chemistry Volume 50: Protein Misassembly*. Academic Press, San Diego, CA, 1997.

[120] G. M. Whitesides. Self-assembling materials. *Scientific American*, September:146–149, 1995.

[121] B. Widom. Lattice model of microemulsions. *Journal of Chemical Physics*, 84:6943–6954, 1986.

[122] T. A. Witten, Jr. and L. M. Sander. Diffusion-limited aggregation, a kinetic critical phenomena. *Physical Review Letters*, 47(19):1400–1403, 1981.

[123] Z. Xie and R. W. Hendrix. Assembly *in vitro* of bacteriophage HK97 proheads. *Journal of Molecular Biology*, 253:74–85, 1995.

[124] A. Zlotnick. To build a virus capsid: an equilibrium model of the self assembly of polyhedral protein complexes. *Journal of Molecular Biology*, 241:59–67, 1994.

[125] J. A. Zonneveld. Automatic integration of ordinary differential equations. Report R743, Mathematisch Centrum, Postbus 4079, 1009AB, 1963.

Appendix A

Control Language Reference

This appendix describes the simulator control language in detail. It is an updated version of a similar appendix in Schwartz [94]. It begins with a general discussion of the language semantics. It then provides descriptions of all built-in commands, broken down by command type. Commands are subdivided into following categories: special forms, type manipulation, graphics, communication, arithmetic and logic, interacting with the operating system, and pre-defined non-primitive procedures.

The control-language uses a generally Lisp-like syntax. Statements in the language are of one of two types: atoms or applications. Atoms consist of a single object of any type defined by the control language. The list of allowed types is provided below. Applications are of the form ($operator$ [$operand$]*), where the operator and operands can be themselves either atoms or applications. Operands may evaluate to any value, but operators must either be one of a pre-defined set of special forms or evaluate to procedures, which may be either user-defined procedures or pre-defined primitive procedures. Procedure evaluation of an application proceeds by recursively evaluating the operator followed by its operands, then applying the operator to the operands. A few special forms described in section A.1 deviate from the normal evaluation mechanism.

The control language contains eight basic data types. Values of type "label" are strings of characters used to name other objects. Values of type "numeric" are numbers, stored as floating point values but used for both integer and floating point operations. Values of type "boolean" have either the values *true* or *false*. Values of type "string" are sequences of characters. Values of type "bitfield" are sequences of binary bits. Values of type "fileid" are used to point to files, which may be opened for reading, writing, or appending. Finally, values of type "none" are used only as place-holders to mark uninitialized elements of arrays.

In addition, there are two compound types: "list" and "array." Lists are linked lists of zero or more values, generally accessed through the procedures `first` and `rest`, which specify the first element of a list and the list minus its first element, respectively. Arrays consist of fixed-size sets of values, each accessible by specifying an index. The main advantage of lists is that their size can change at run-time. The main advantage of arrays is that the time to access an element is constant for an array, whereas the time to access a list element depends on its position in the list. An object of either type can contain elements of any type, including the two compound types, with the exception of the error type described below. In addition, multiple types of elements can be contained within a single list or array.

One additional type, "error," is used to implement an error-handling mechanism. Errors are treated the same as other first-class data types, with the exception of that it is not possible to store errors in lists or arrays or name them by labels. A procedure or special form can signal an error by returning an error value in place of its normal return type. Each error has associated with it a string that can be used to describe the nature of the error. All primitive procedures propagate their error values to whatever procedure called them, or to the user in the case of a user-called procedure. In addition, users have two mechanisms for interacting directly with the error subsystem: a primitive procedure, `make_error`, which allows a user to generate an error with a specified string; and a special form, `errortrap`, which allows a user to catch errors, examine their strings, and block their further propagation. Because errors are propagated by the same mechanism as normal procedure returns, they can be assumed to be of comparable efficiency, making error propagation and trapping useful for handling common-case occurrences rather than just the infrequent cases to which slower error handling might be restricted.

Memory deallocation is handled by automatic garbage collection. The interpreter uses a mark-and-sweep garbage collector, triggered to act at the beginning of the evaluation of any object if the total dynamic memory allocated since the last garbage collection exceeds a hard-coded value, currently set to 1,048,586 bytes.

A.1 Special Forms

Special forms are commands that are handled separately from normal procedure application. Typically, they require special processing of their input lists in ways not allowed by normal user procedures. Other special cases, such as `source` and `iterate`, require separate handling because they evaluate other objects or apply other procedures, which for convenience in design is not allowed for primitive procedures. Unlike with primitive procedures, the names of special forms are reserved words and cannot be redefined to refer to other objects. In addition, users cannot define new special forms. The currently supported special forms are as follows:

`define` = **procedure** (label *l*, any *a*) **returns** any
modifies: *l*
effects: `define` binds *l* to *a* and returns the value of *a*.

`errortrap` = **procedure** (any *a*) **returns** list
effects: `errortrap` returns a list of two items. If *a* evaluates to an error then the first item is the boolean *true* and the second is *a*'s associated string. Otherwise, the first item is the boolean *false* and the second is the value of *a*.

`if` = **procedure** (boolean *b*, any *e1*, any *e2*) **returns** any
effects: `if` evaluates *b* and returns the value of *e1* if *b* is true, or *e2* if *b* is false. `if` does not evaluate whichever of *e1* and *e2* is not returned. *e2* is optional and if it is not included and *b* is false then `if` returns *false*.

`iterate` = **procedure** (procedure *p*, numeric *n1*, numeric *n2*) **returns** any
modifies: anything
effects: `iterate` first evaluates *p*, *n1*, and *n2*, returning an error if *p* is not a procedure or *n1* or *n2* is not a number. It then executes *p* in order on all values between the integer part of *n1* and the integer part of *n2*, inclusive, propagating any errors that occur. If it terminates, `iterate` returns the value returned by the last invocation of *p*, or *false* if *p* was never invoked.

`make_list` = **procedure** (any *a1*, any *a2*, ...) **returns** list
effects: `make_list` returns a list containing the values of its arguments, in order.

`procedure` = **procedure** (list l, any a) **returns** procedure
effects: `procedure` returns a procedure with variables listed in l that evaluates to the value of a with its arguments substituted for any occurrences of the variables in l.

`quote` = **procedure** (any a) **returns** any
effects: `quote` returns a unevaluated.

`sequence` = **procedure** (any *a1*, any *a2*, ...) **returns** any
effects: `sequence` evaluates each argument in order, returning the value of the last argument.

`set` = **procedure** (label l, any a) **returns** any
modifies: l
effects: If l is bound to a value, `set` replaces that value with the value of a and returns the old value. Otherwise, it returns an error.

`set_array` = **procedure** (array a, numeric n, any x) **returns** array
modifies: a
effects: If $0 \leq n < |a|$, `set_array` sets the element of a whose index is the integer part of n, indexed from zero, to the value of x and returns the modified array.

`set_first` = **procedure** (list l, any a) **returns** any
modifies: l
effects: `set_first` sets the first element of l to the value of a and returns the replaced value.

`set_rest` = **procedure** (list *l1*, list *l2*) **returns** list
modifies: *l1*
effects: `set_rest` replaces all of *l1* after the first element with *l2* and returns the replaced value.

`source` = **procedure** (string s) **returns** boolean
modifies: anything
effects: `source` evaluates s as a string and attempts to open a file named by the string, read in its contents, and pass the file to the interpreter, returning an error if it is unable to open or read the file or *true* otherwise. `source` does not propagate errors produced by the execution of the contents of the file.

A.2 Type Manipulation

This section describes primitive procedures involved in creating, testing, and converting between different types. These procedures include several predicates to test for membership in different types. They also include procedures for converting certain types to others. In addition, they include a few special operations for creating and examining the compound types, list and array. Finally, they include the special operation `make_error`, used to signal user-defined errors.

`bit2double` = **procedure** (bitmap b) **returns** numeric
effects: `bit2double` interprets b as a floating point number and returns the number.

`bit2int` = **procedure** (bitmap b) **returns** numeric
effects: `bit2int` interprets b as an integer and returns its value.

`concat` = **procedure** (list *l1*, list *l2*) **returns** list
modifies: *l1*
effects: `concat` concatenates *l2* onto the end of *l1*.

`double2bit` = **procedure** (numeric n) **returns** bitmap
effects: `double2bit` returns a bitmap containing the floating point representation of n.

`first` = **procedure** (list l) **returns** any
effects: `first` returns the first element of l, if any, or an error if l is empty.

`get_array` = **procedure** (array a, numeric n) **returns** any
effects: If $0 \leq n < |a|$ and the element of a whose index is the integer part of n, indexed from zero, has been initialized, `get_array` returns the value of that element. Otherwise, it returns an error.

`int2bit` = **procedure** (numeric n) **returns** bitmap
effects: `int2bit` returns a bitmap containing the binary value of the integer part of n.

`isarray` = **procedure** (any a) **returns** boolean
effects: `isarray` returns *true* if a is an array, *false* otherwise.

`isboolean` = **procedure** (any a) **returns** boolean
effects: `isboolean` returns *true* if a is a boolean, *false* otherwise.

`isfileid` = **procedure** (any a) **returns** boolean
effects: `isfileid` returns true if a is a fileid, *false* otherwise.

`islabel` = **procedure** (any a) **returns** boolean
effects: `islabel` returns *true* if a is a label, *false* otherwise.

`islist` = **procedure** (any a) **returns** boolean
effects: `islist` returns *true* if a is a list, *false* otherwise.

`isnumeric` = **procedure** (any a) **returns** boolean
effects: `isnumeric` returns *true* if a is numeric, *false* otherwise.

`isprocedure` = **procedure** (any a) **returns** boolean
effects: `isprocedure` returns *true* if a is a procedure, *false* otherwise.

`isstring` = **procedure** (any a) **returns** boolean
effects: `isstring` returns true if a is a string, *false* otherwise.

`lab2str` = **procedure** (label l) **returns** string
effects: `lab2str` returns the string associated with l.

`length` = **procedure** (list l) **returns** numeric
effects: `length` returns the length of list l.

`make_array` = **procedure** (numeric n) **returns** array
effects: If $n \geq 0$, `make_array` returns an empty array whose length is the integer part of n. Otherwise, it returns an error.

`make_error` = **procedure** (string s) **returns** error
effects: `make_error` returns an error with associated string s.

`num2str` = **procedure** (numeric n) **returns** string
effects: `num2str` returns a user-readable string form for n.

`rest` = **procedure** (list l) **returns** list
effects: `rest` returns all of l except the first element if l is not empty. Otherwise, it returns an error.

`size` = **procedure** (array a) **returns** numeric
effects: `size` returns the number of elements a can contain.

`str2lab` = **procedure** (string s) **returns** label
effects: `str2lab` returns a label whose associated string is s.

`str2num` = **procedure** (string s) **returns** numeric
effects: If s contains the string representation of a number, `str2num` returns the number. Otherwise it returns an error.

`strcat` = **procedure** (string *s1*, string *s2*, ...) **returns** string
effects: `strcat` returns the result of concatenating all of its string arguments. If it does not have at least one argument, it returns an error.

`strlen` = **procedure** (string s) **returns** numeric
effects: `strlen` returns the number of characters in s.

`substr` = **procedure** (string s, numeric a, numeric b) **returns** string
effects: If $0 \leq a \leq b < |s|$, `substr` returns the substring of s with indices from the integer part of a to the integer part of b, starting from 0. Otherwise, it returns an error.

A.3 Graphics

A variety of primitive routines exist for creating and manipulating the graphics display. The following commands implement a series of generally simple graphics primitives that can be used for drawing objects:

`copy_to_render_buffer` = **procedure** (list *wl*, array *singles*, array *nodes*, array *types*, numeric *selected*, numeric *length*, numeric *width*, numeric *depth*) **returns** boolean

> **effects**: `copy_to_render_buffer` takes a series of arguments specifying the current state of the simulation and copies a collection of all lines, spheres, and cylinders that need to be drawn to render the workspace into a set of internal buffers for the purpose of accelerating rendering later. *length*, *width*, and *height* specify the desired workspace dimensions and *selected* gives the index of the currently selected node, if any, or -1 if no node is selected. *wl*, *singles*, *nodes*, and *types* are data structures that are maintained by the controller and are not generally directly modified by the user. *wl* is the output of the procedure `query_all`. *nodes* and *singles* are initially empty data structures updated by `add_node` to reflect all nodes and single nodes, respectively, in the simulation. *types* is an initially empty structure updated by the three type adding procedures: `add_single_node_type`, `add_complex_node_type`, and `add_variable_node_type`. `copy_to_render_buffer` returns *true* if its arguments are well-formed and it successfully updates the rendering buffer, or an error otherwise.

`draw_workspace` = **procedure** () **returns** boolean

> **effects**: `draw_workspace` draws all visible nodes and edges in accordance with the current focal point and eye point. It returns *true*.

`init_graphics` = **procedure** (numeric *wx*, numeric *wy*, numeric *wz*, numeric *ex*, numeric *ey*, numeric *ez*, numeric *cx*, numeric *cy*, numeric *cz*, numeric *ux*, numeric *uy*, numeric *uz*) **returns** boolean

> **effects**: `init_graphics` initializes the graphics display with workspace size (*wx*,*wy*,*wz*), eye point (*ex*,*ey*,*ez*), focal point (*cx*,*cy*,*cz*), and up vector (*ux*,*uy*,*uz*), then returns *true*.

`make_cylinder` = **procedure** (numeric *sx*, numeric *sy*, numeric *sz*, numeric *ex*, numeric *ey*, numeric *ez*, numeric *r*, numeric *c*) **returns** boolean

> **effects**: `make_cylinder` draws a cylinder with axis from (*sx*,*sy*,*sz*) to (*ex*,*ey*,*ez*), radius *r*, and color *c* and returns *true*.

`make_line` = **procedure** (numeric *sx*, numeric *sy*, numeric *sz*, numeric *ex*, numeric *ey*, numeric *ez*, numeric *c*) **returns** boolean

> **effects**: `make_line` draws a line from (*sx*,*sy*,*sz*) to (*ex*,*ey*,*ez*) with color *c* and returns *true*.

`make_sphere` = **procedure** (numeric *cx*, numeric *cy*, numeric *cz*, numeric *sr*, numerics *c*) **returns** boolean

> **effects**: `make_sphere` draws a sphere at center (*cx*,*cy*,*cz*) with radius *r* and color *c* and returns true.

`redraw` = **procedure** (numeric *c*) **returns** boolean

> **effects**: `redraw` redraws the screen, setting the background to color *c*, and returns *true*.

`render_from_buffer` = **procedure** () **returns** boolean

> **effects**: `render_from_buffer` renders all objects in the rendering buffer created by `copy_to_render_buffer` and returns *true*.

The following commands are used to control how the drawn objects are viewed:

`current_axes` = **procedure** () **returns** array
effects: `current_axes` returns an array specifying the current rotation of the view of the workspace in the form of the current directions of the true X, Y, and Z axes as vectors in screen coordinates. The array returned contains nine values of the form $[X_x, X_y, X_z, Y_x, Y_y, Y_z, Z_x, Z_y, Z_z]$, where the X axis has direction (X_x, X_y, X_z), the Y axis has direction (Y_x, Y_y, Y_z), and the Z axis has direction (Z_x, Z_y, Z_z).

`current_zoom` = **procedure** () **returns** array
effects: `current_zoom` returns an array specifying the current zoom factors in the X, Y, and Z directions. Since the simulator does not allow different scaling in different dimensions, the three elements will always be the same.

`recenter` = **procedure** (numeric x, numeric y, numeric z) **returns** boolean
effects: `recenter` sets the focal point to (x, y, z). It returns *true*.

`rotate_x` = **procedure** (numeric *ang*) **returns** boolean
effects: `rotate_x` rotates the eye point counter-clockwise around the X-axis by *ang* degrees relative to the focal point. It returns *true*.

`rotate_y` = **procedure** (numeric *ang*) **returns** boolean
effects: `rotate_y` rotates the eye point counter-clockwise around the Y-axis by *ang* degrees relative to the focal point. It returns *true*.

`rotate_z` = **procedure** (numeric *ang*) **returns** boolean
effects: `rotate_z` rotates the eye point counter-clockwise around the Z-axis by *ang* degrees relative to the focal point. It returns *true*.

`translate` = **procedure** (numeric x, numeric y, numeric z) **returns** boolean
effects: `translate` translates the focal point of the graphics display by (x, y, z). It returns *true*.

`zoom` = **procedure** (numeric z) **returns** boolean
effects: `zoom` multiplies the distance between the virtual eye point and the focal point by z. It returns *true*.

`zoom_in` = **procedure** () **returns** boolean
effects: `zoom_in` multiplies the distance between the virtual eye point and the focal point by .8. It returns *true*.

`zoom_out` = **procedure** () **returns** boolean
effects: `zoom_out` multiplies the distance between the virtual eye point and the focal point by 1.25. It returns *true*.

A.4 Communication

The following primitive procedures are used for direct interactions with the physical model implemented by the numerical methods portion of the simulator. They consist of a series of queries for examining the current simulation state and commands for modifying it. The following queries are supported:

`query_all` = **procedure** () **returns** list
effects: `query_all` requests simulator data on the positions of all nodes and returns it in the form of a list whose first element is an array of the indices of all single nodes, whose second element is an array of their positions, whose third element is an array of their orientations, and whose final element is an array of arrays of the status of their edges, in which an element is -1 if the corresponding edge is unconnected and is otherwise the index of the edge to which it is connected.

`query_confs` = **procedure** () **returns** array
effects: `query_confs` queries the numerical simulator about the conformation of each variable node and returns an array of all conformations.

`query_constant` = **procedure** (string s) **returns** numeric
effects: `query_constant` queries the numerical simulator about the value named by s and returns the result or an error if s does not correspond to any simulator constant.

`query_edges` = **procedure** () **returns** array
effects: `query_edges` queries the numerical simulator about which edges are connected and returns an array in which each element i is -1 if edge i is not connected to any other edge and is j if edge i is connected to edge j.

`query_node` = **procedure** (numeric n) **returns** array
effects: `query_node` queries the numerical simulator about the position, velocity, rotation, and angular velocity of the node whose index is the integer part of n and returns an array of twelve numeric values containing that data, in order.

The control language also supports the following communication commands:

`clear_all` = **procedure** () **returns** boolean
effects: `clear_all` instructs the numerical simulator to clear all data and return to a startup state. It returns *true*.

`connect` = **procedure** (string h, numeric p) **returns** boolean
effects: `connect` attempts to establish a connection to the host h, checking at ports from p to $p + 31$. It returns *true* if it establishes the connection, and an error if it cannot.

`disconnect` = **procedure** () **returns** boolean
effects: `disconnect` closes the connection, if any, and returns *true*.

`init_workspace` = **procedure** (numeric x, numeric y, numeric z) **returns** boolean
effects: `init_workspace` instructs the numerical simulator to set the workspace size to (x, y, z) and returns *true*.

`make_edge` = **procedure** (numeric *n1*, numeric *n2*) **returns** boolean
effects: `make_edge` instructs the numerical simulator to attach the edge whose index is the integer part of *n1* to the edge whose index is the integer part of *n2* and returns *true*.

`make_edge_type` = **procedure** (list o, numeric kt, numeric kr, numeric ks, numeric er, numeric ed, numeric r, numeric a) **returns** boolean
effects: `make_edge_type` instructs the numerical simulator to define a new edge type. o is a list of numeric values specifying the other types to which this edge can attach. kt, kr, and ks are the three spring constants. er and ed are binding energies. r and a are the distance and angle tolerances, respectively. `make_edge_type` returns *true*.

`make_node` = **procedure** (numeric n, list $l1$, list $l2$, list $l3$, list $l4$) returns boolean
effects: `make_node` instructs the numerical simulator to create a node of type n with position, velocity, angle, and angular velocity given by $l1$, $l2$, $l3$, and $l4$, respectively, and returns *true*.

`make_node_type` = **procedure** (list l) **returns** boolean
effects: `make_node_type` instructs the numerical simulator to define a new type of node. The first element of l is one of "single", "complex", and "variable", for defining single, complex, and variable nodes, respectively. For single nodes, the remainder of the list specifies the radius, a currently unused numerical parameter, the mass, and lists of the positions, up vectors, and types of the edges. For complex nodes, the remainder of the list consists of lists of the offsets, rotations, and types of the sub-nodes. For variable node types, the remainder of the list is two lists, the first listing conformation energies and the second listing the corresponding conformations. `make_node_type` returns an error if its argument does not conform to the above format, or *true* otherwise.

`remove_edge` = **procedure** (numeric n) **returns** boolean
effects: `remove_edge` instructs the numerical simulator to break any bond containing the edge whose index is the integer part of n and returns *true*.

`remove_node` = **procedure** (numeric n) **returns** boolean
effects: `remove_node` instructs the numerical simulator to remove the node whose index is the integer part of n and returns *true*.

`set_conformation` = **procedure** (numeric $n1$, numeric $n2$) **returns** boolean
effects: `set_conformation` instructs the numerical simulator to set the variable node whose index is the integer part of $n1$ to the conformation specified by the integer part of $n2$ and returns *true*.

`set_constant` = **procedure** (string s, numeric n) **returns** boolean
effects: `set_constant` instructs the numerical simulator to set the value named by s to n and returns *true*.

`step` = **procedure** () **returns** boolean
effects: `step` instructs the numerical simulator to advance the time by `steps_per_update`·`timestep` and returns *true*.

A.5 Arithmetic and Logic

The following is a collection of primitive procedures involved with handling arithmetic and logic operations. These procedures include some boolean operators and standard scalar arithmetic operators. In addition, they include special vector operations. Although vectors are not in fact a distinct type, being represented internally as arrays of three numeric elements, the importance of fast vector arithmetic led me to create primitive procedures for common vector operations rather than implementing them in terms of the scalar arithmetic and array manipulation operators.

`+` = **procedure** (numeric *n1*, numeric *n2*, ...) **returns** numeric
effects: `+` returns the sum of all arguments, or 0 if the argument list is empty.

`-` = **procedure** (numeric *n1*, numeric *n2*, ...) **returns** numeric
effects: `-` returns the result of subtracting all arguments but the first from the first. Unlike `+`, `-` must have at least one argument or it returns an error.

`*` = **procedure** (numeric *n1*, numeric *n2*, ...) **returns** numeric
effects: `*` returns the product of all arguments, or 1.0 if the argument list is empty.

`/` = **procedure** (numeric *n1*, numeric *n2*, ...) **returns** numeric
effects: `/` returns the result of dividing the first argument by all remaining arguments. Unlike `*`, `/` must have at least one argument or it returns an error.

`=` = **procedure** (any *a1*, any *a2*) **returns** boolean
effects: `=` returns *true* if *a1* and *a2* have the same type and are equal, *false* if they have the same type and are not equal, and an error if they do not have the same type. For compound objects, equal means that they point to the same location in memory. For simple objects, it means that they have the same value.

$>$ = **procedure** (any *a1*, any *a2*) **returns** boolean
effects: If *a1* and *a2* are of different types, $>$ returns an error. If *a1* and *a2* are both numeric, it returns *true* if $a1 > a2$, *false* otherwise. If *a1* and *a2* are both strings or both labels, $>$ returns *true* if *a1* is lexicographically greater than *a2*, *false* otherwise. For other types, the behavior is undefined.

$<$ = **procedure** (any *a1*, any *a2*) **returns** boolean
effects: If *a1* and *a2* are of different types, $<$ returns an error. If *a1* and *a2* are both numeric, it returns *true* if $a1 < a2$, *false* otherwise. If *a1* and *a2* are both strings or both labels, $<$ returns *true* if *a1* is lexicographically less than *a2*, *false* otherwise. For other types, the behavior is undefined.

`acos` = **procedure** (numeric *n*) **returns** numeric
effects: `acos` returns the Arccosine of *n* radians.

`and` = **procedure** (boolean *b1*, boolean *b2*) **returns** boolean
effects: `and` returns the logical and of *b1* and *b2*.

`asin` = **procedure** (numeric *n*) **returns** numeric
effects: `asin` returns the Arcsin of *n* radians.

`atan` = **procedure** (numeric *n*) **returns** numeric
effects: `atan` returns the Arctangent of *n* radians.

`atan2` = **procedure** (numeric *n1*, numeric *n2*) **returns** numeric
effects: `atan2` returns the Arctangent of $n2/n1$ radians.

`compose_rotations` = **procedure** (vector *r1*, vector *r2*) **returns** vector
effects: `compose_rotations` returns a rotation vector whose effect is equivalent to rotating by *r1* followed by rotating by *r2*. Rotating by a vector is defined as rotating counter-clockwise around the vector by a number of radians equal to the length of the vector.

`cos` = **procedure** (numeric n) **returns** numeric
effects: `cos` returns the cosine of n radians.

`exp` = **procedure** (numeric n) **returns** numeric
effects: `exp` returns e^n.

`log` = **procedure** (numeric n) **returns** numeric
effects: `log` returns the natural log of n.

`not` = **procedure** (boolean b) **returns** boolean
effects: `not` returns the logical negation of b.

`or` = **procedure** (boolean *b1*, boolean *b2*) **returns** boolean
effects: `or` returns the logical or of *b1* and *b2*.

`pow` = **procedure** (numeric a, numeric b) **returns** numeric
effects: `pow` returns a^b.

`sin` = **procedure** (numeric n) **returns** numeric
effects: `sin` returns the sine of n radians.

`svprod` = **procedure** (numeric n, vector v) **returns** vector
effects: `svprod` returns the vector produced by multiplying v term-wise by n.

`tan` = **procedure** (numeric n) **returns** numeric
effects: `tan` returns the tangent of n radians.

`trunc` = **procedure** (numeric n) **returns** numeric
effects: `trunc` returns the integer portion of n.

`vadd` = **procedure** (vector *v1*, vector *v2*) **returns** vector
effects: `vadd` returns the vector sum of *v1* and *v2*.

`vcross` = **procedure** (vector *v1*, vector *v2*) **returns** vector
effects: `vcross` returns the cross product of $v1$ and $v2$.

`vdot` = **procedure** (vector *v1*, vector *v2*) **returns** numeric
effects: `vdot` returns the dot product of *v1* and *v2*.

`vecrot` = **procedure** (vector v, vector r) **returns** vector
effects: `vecrot` returns the result of rotating v around r counter-clockwise by $||r||$ radians.

`vecx` = **procedure** (vector v) **returns** numeric
effects: `vecx` returns the X component of v.

`vecy` = **procedure** (vector v) **returns** numeric
effects: `vecy` returns the Y component of v.

`vecz` = **procedure** (vector v) **returns** numeric
effects: `vecz` returns the Z component of v.

`vlen` = **procedure** (vector v) **returns** numeric
effects: `vlen` returns the length of v.

`vnorm` = **procedure** (vector v) **returns** vector
effects: `vnorm` returns a vector with the same direction as v and a length of 1.

`vrotx` = **procedure** (vector v, numeric a) **returns** vector
effects: `vrotx` returns the result of rotating v counter-clockwise by a degrees around the X axis.

`vroty` = **procedure** (vector v, numeric a) **returns** vector
effects: `vroty` returns the result of rotating v counter-clockwise by a degrees around the Y axis.

`vrotz` = **procedure** (vector v, numeric a) **returns** vector
effects: `vrotz` returns the result of rotating v counter-clockwise by a degrees around the Z axis.

`vsub` = **procedure** (vector *v1*, vector *v2*) **returns** vector
effects: `vsub` returns the vector difference between *v1* and *v2*.

A.6 Interacting with the Operating System

These commands control a few special-purpose operations: file access, execution of shell commands, and access to the terminal. The commands are as follows:

`closef` = **procedure** (fileid f) **returns** fileid
effects: `closef` closes f and returns it.

`debug` = **procedure** (string s) **returns** string
effects: `debug` outputs s to the standard output and returns s.

`open` = **procedure** (string f, string t) **returns** fileid
effects: `open` opens a file of name f with type t, where t is the file type string as defined by the underlying C, and returns the file's id. If `open` cannot open the file, it returns an error.

`readf` = **procedure** (fileid f, string s) **returns** string
effects: `readf` reads in characters from f until it reaches the end of the file or finds a character in s. It returns a string containing all characters read, including the termination character, if any.

`system` = **procedure** (string s) **returns** numeric
effects: `system` passes its string argument via the C `system` command to the operating system for execution as a shell command. It returns the value returned by the shell.

`writef` = **procedure** (fileid f, string s) **returns** string
effects: `writef` writes s to f, provided f is open for writing, and returns s. If f is not open for writing, `writef` returns an error.

A.7 Pre-Defined Non-Primitive Procedures

There are several important commands that are not primitive procedures, but rather are implemented in the control language primarily by two files — `initial.dat` and `controller.dat` — automatically loaded when the simulator is started. These procedures are frequently accessed as "hooks" by the graphics routines, in save files created by the simulator, or as a convenience in user-created rules files.

`add_complex_node_type` = **procedure** (list *pos*, list *ang*, list *nodes*) **returns** string
modifies: `the_node_types`, `num_node_types`
effects: `add_complex_node_type` creates a complex node type with parameters as specified in the argument list and adds this type to the final position in `the_node_types`, incrementing `num_node_types`. It returns a string signifying that it has finished.

`add_edge_type` = **procedure** (list o, numeric kt, numeric kr, numeric ks, numeric er, numeric ed, numeric r, numeric a) **returns** string
modifies: `the_edge_types`, `num_edge_types`
effects: `add_edge_type` creates an edge type capable of connecting to edges listed in o; with spring constants kt, kr, and ks; energies er and ed; distance tolerance r; and angle tolerance a. It returns a string signifying that it has finished.

`add_node` = **procedure** (numeric t, list p, list v, list r, list a) **returns** list
modifies: `the_nodes`, `num_nodes`, `the_edges`, `num_edges`, `the_singles`, `num_singles`, `num_variables`
effects: `add_node` adds a node with type t, position p, velocity v, angular rotation r, and angular velocity a. p, v, r, and a are expressed as lists of three numerical values each. `add_node` adds the new node to the list of nodes and its edges and single sub-nodes to the appropriate lists and increments counters for the number of nodes, edges, singles, and variables. It returns a list of the node parameters followed by the newly defined node.

`add_random_node` = **procedure** (list n) **returns** list
modifies: `the_nodes`, `num_nodes`, `the_edges`, `num_edges`, `the_singles`, `num_singles`, `num_variables`
effects: `add_random_node` behaves just as `add_node` does, except that the added node has a random type chosen from a list of node types in n, its position is randomly chosen within the workspace, its angular rotation is random, and it has a random velocity and angular rotation of magnitude between 0 and 1 in each component. All distributions are uniform.

add_single_node_type = **procedure** (numeric r, numeric c, numeric m, list *poses*, list *ups*, list *edges*) **returns** string

modifies: the_node_types, num_node_types

effects: add_single_node_type creates a single node type with parameters as specified in the argument list and adds this type to the final position in the_node_types, incrementing num_node_types. It returns a string signifying that it has finished.

add_variable_node_type = **procedure** (list *energy*, list *nodes*) **returns** string

modifies: the_node_types, num_node_types

effects: add_variable_node_type creates a variable node type with parameters as specified in the argument list and adds this type to the final position in the_node_types, incrementing num_node_types. It returns a string signifying that it has finished.

current_x_zoom = **procedure** () **returns** numeric

effects: current_x_zoom returns the current zoom along the X axis.

current_y_zoom = **procedure** () **returns** numeric

effects: current_y_zoom returns the current zoom along the Y axis.

current_z_zoom = **procedure** () **returns** numeric

effects: current_z_zoom returns the current zoom along the Z axis.

delete_node = **procedure** (numeric n) **returns** numeric

modifies: the_nodes, num_nodes, the_edges, num_edges, the_singles, num_singles, num_variables

effects: delete_node deletes the node whose index is the integer part of n from the simulation, removing the node, its edges, and its singles from the relevant arrays and decrementing the counters of nodes, edges, singles, and variables. It returns the number of nodes present prior to the deletion.

delete_region = **procedure** (numeric *xmin*, numeric *xmax*, numeric *ymin*, numeric *ymax*, numeric *zmin*, numeric *zmax*) **returns** boolean

modifies: the_nodes, num_nodes, the_edges, num_edges, the_singles, num_singles, num_variables

effects: delete_region behaves as does delete_node, except that it deletes all nodes with position (x, y, z) such that *xmin*$\leq x \leq$*xmax*, *ymin*$\leq y \leq$*ymax*, and *zmin*$\leq z \leq$*zmax*. It returns *false*.

do_reposition = **procedure** () **returns** boolean

effects: do_reposition sets the graphics display to dimensions defined by the numeric variables workspace_width, workspace_height, and workspace_depth; eye point defined by the vector variable graphics_eye; focal point defined by the vector variable graphics_center; and up vector defined by the vector variable graphics_up. It also redraws the display and returns *true*.

do_save = **procedure** (string f) **returns** fileid

effects: do_save attempts to open a file with name f, write out a save file capable of reconstructing the current state of the numerical simulator, close the file, and return its fileid. If it is unable to open a file with the specified name for writing, it returns an error.

`do_step_forward` = **procedure** () **returns** string
modifies: workspace_data
effects: `do_step_forward` instructs the numerical simulator to advance the time `updates_per_run`·`steps_per_update` steps. `do_step_forward` updates the display after each `steps_per_update` steps and leaves updated values of the current data in the global variable `workspace_data`. It returns the string "Stepped forward".

`draw_complete` = **procedure** () **returns** boolean
modifies: anything
effects: `draw_complete` queries the numerical routines for the current single node positions and rotations, updates the rendering buffer, and redraws the workspace. It returns *true.*

`process_node_query` = **procedure** (numeric n) **returns** string
effects: `process_node_query` queries the numerical simulator for the position, velocity, angular rotation, and angular velocity of the node whose index is the integer part of n and returns a string containing the results in the form:
"Pos: *position*
Vel: *velocity*
Ang: *angular rotation*
AngVel: *angular velocity*".

`query_region` = **procedure** (numeric *xmin*, numeric *xmax*, numeric *ymin*, numeric *ymax*, numeric *zmin*, numeric *zmax*) **returns** string
effects: `query_region` queries the numerical simulator for the positions, velocities, angular rotations, and angular velocities of each node with x coordinate between *xmin* and *xmax*, y coordinate between *ymin* and *ymax*, and z coordinate between *zmin* and *zmax*. It averages all values for each of the four fields and returns a string containing the results in the form:
"Average values:
Pos: *average position*
Vel: *average velocity*
Ang: *average angular rotation*
AngVel: *average angular velocity*".

`recenter_on_node` = **procedure** (numeric n) **returns** boolean
effects: `recenter_on_node` translates the focal point to the node whose index is the integer part of n and redraws the display. It returns *true.*

`resize_workspace` = **procedure** (numeric x, numeric y, numeric z) **returns** boolean
modifies: `workspace_width`, `workspace_height`, `workspace_depth`, `graphics_eye`
effects: `resize_workspace` sets `workspace_width`, `workspace_height`, and `workspace_depth` to x, y, and z, respectively. It moves `graphics_eye`, the eye point of the graphics display, so that the direction from it to the focal point is unchanged, but the distance is 1.44 times the maximum of x, y, and z. It returns *false.*

`restart` = **procedure** () **returns** boolean
modifies: anything
effects: `restart` sets simulation parameters to their initial values, clears the display, closes the connection to the numerical simulator, if any, and attempts to establish a new connection at the machine and port specified by the global variables `host` and `port`. It returns *false*.

`rotate_x_left_fixed` = **procedure** () **returns** boolean
effects: `rotate_x_left_fixed` rotates the eye point counter-clockwise around the X-axis by five degrees. It returns *true*.

`rotate_x_right_fixed` = **procedure** () **returns** boolean
effects: `rotate_x_right_fixed` rotates the eye point clockwise around the X-axis by five degrees. It returns *true*.

`rotate_y_left_fixed` = **procedure** () **returns** boolean
effects: `rotate_y_left_fixed` rotates the eye point counter-clockwise around the Y-axis by five degrees. It returns *true*.

`rotate_y_right_fixed` = **procedure** () **returns** boolean
effects: `rotate_y_right_fixed` rotates the eye point clockwise around the Y-axis by five degrees. It returns *true*.

`rotate_z_left_fixed` = **procedure** () **returns** boolean
effects: `rotate_z_left_fixed` rotates the eye point counter-clockwise around the Z-axis by five degrees. It returns *true*.

`rotate_z_right_fixed` = **procedure** () **returns** boolean
effects: `rotate_z_right_fixed` rotates the eye point clockwise around the Z-axis by five degrees. It returns *true*.

`translate_xminus_fixed` = **procedure** () **returns** boolean
effects: `translate_xminus_fixed` translates the display a distance of 5 times the current X zoom along the negative X-axis. It returns *true*.

`translate_xplus_fixed` = **procedure** () **returns** boolean
effects: `translate_xplus_fixed` translates the display a distance of 5 times the current X zoom along the positive X-axis. It returns *true*.

`translate_yminus_fixed` = **procedure** () **returns** boolean
effects: `translate_yminus_fixed` translates the display a distance of 5 times the current Y zoom along the negative Y-axis. It returns *true*.

`translate_yplus_fixed` = **procedure** () **returns** boolean
effects: `translate_yplus_fixed` translates the display a distance of 5 times the current Y zoom along the positive Y-axis. It returns *true*.

`translate_zminus_fixed` = **procedure** () **returns** boolean
effects: `translate_zminus_fixed` translates the display a distance of 5 times the current Z zoom along the negative Z-axis. It returns *true*.

`translate_zplus_fixed` = **procedure** () **returns** boolean
effects: `translate_zplus_fixed` translates the display a distance of 5 times the current Z zoom along the positive Z-axis. It returns *true*.

Appendix B

Network Interface Protocols

The network interface uses a fixed series of commands and queries to control communication between the user interface and the numerical routines. This section documents the network interface commands and queries. The basic protocol for communication begins with the user interface sending a code specifying the command or query to be executed. The user interface may have to send additional data, depending on the command or query. In the case of a command, the user interface is finished as soon as all data is sent, as the numerical routines do not send any response. In the case of a query, the user interface blocks while waiting for a response from the numerical routines. The protocol terminates in that case once all response data has been received. Behavior is unspecified if the transmitted data is not well-formatted or contains values inconsistent with the current simulation state. The supported commands, the associated input data, and the required output are as follows:

CLEAR_ALL:
input: none
output: none
effects: instructs the numerical routines to reset the simulation to its initial state by deleting all defined nodes, edges, and node and edge types and freeing any dynamically allocated memory associated with them.

CONSTANT:
input: an integer i and a double precision or integer value d as described under **effects**
output: none
effects: instructs the numerical routines to change a global constant, specified by i. The format of d depends on the value of i, according to the following list of allowed values, input formats, and functions:

- 0-2: double precision floating point, currently unused
- 3: double precision floating point, changes the time step
- 4: double precision floating point, changes translational error tolerance
- 5: double precision floating point, changes temperature
- 6: integer, changes the number of steps executed for each STEP command issued
- 7: double precision floating point, changes the value of ε, a constant used by some mathematical routines to represent a number "close to zero"
- 8: double precision floating point, changes the damping value
- 9: double precision floating point, changes rotational error tolerance

MAKE_EDGE:
input: two integers e_1 and e_2
output: none
effects: instructs the numerical routines to form a bond between the edges of indices e_1 and e_2.

MAKE_EDGE_TYPE:
input: an integer n, n integers $o_1 \cdots o_n$, and double precision floating point values k_t, k_r, k_s, e_a, e_d, t_t, t_r, and t_s
output: none
effects: instructs the numerical routines to define a new edge type with translational, rotational, and bending spring constants k_t, k_r, and k_s; translational, rotational, and bending tolerances t_t, t_r, and t_s; association energy e_a; dissociation energy e_d; and allowed binding partners of types $o_1 \cdots o_n$.

MAKE_NODE:
input: an integer i and 12 double precision floating point values $p_x, p_y, p_z, v_x, v_y, v_z, r_x, r_y, r_z, a_x, a_y, a_z$
output: none
effects: instructs the numerical routines to create a node of type i with position (p_x, p_y, p_z), velocity (v_x, v_y, v_z), rotation (r_x, r_y, r_z), and rotational velocity (a_x, a_y, a_z).

MAKE_NODE_TYPE:
input: see below
output: none
effects: instructs the numerical routines to create a new node type. The first argument is an integer containing one of three constants — SINGLE, COMPLEX, or VARIABLE — which denote which of the three kinds of node type is being defined. The remaining arguments depend on the kind of type as follows:

- SINGLE: a double precision floating-point value specifying radius, a double precision floating-point value specifying a currently unused parameter, a double precision floating-point value specifying mass, an integer n specifying the number of edges, $3n$ double precision floating-point values specifying n position vectors for the edge end-points, $3n$ double precision floating-point values specifying n edge "up" vectors, and n integers specifying the types of the edges.
- COMPLEX: an integer n specifying the number of sub-nodes, $3n$ double precision floating-point values specifying n position vectors for the sub-nodes, $3n$ double precision floating-point values specifying n rotation vectors for the sub-nodes, and n integers specifying the types of the sub-nodes.
- VARIABLE: an integer n specifying the number of conformations, n double precision floating-point values specifying the activation energies of the conformations, n double precision floating-point values specifying the deactivation energies of the conformations, and n integers specifying the types of the conformations.

The new type is created as specified by the arguments for the kind of type created.

QUIT:
input: none
output: none
effects: instructs the numerical routines to terminate communication with the user interface, free all dynamically allocated memory used by the current simulation, reset to its starting state, and wait for a new network connection.

REMOVE_EDGE:
input: an integer e
output: none
effects: instructs the numerical routines to break any bond between edge e and its binding partner, if any.

REMOVE_NODE:
input: an integer n
output: none
effects: instructs the numerical routines to delete node n from the simulation.

SET_CONF:
input: an integer n and an integer c
output: none
effects: instructs the numerical routines to set the conformation of the nth variable node, counting from zero, to c.

STEP:
input: none
output: none
effects: instructs the numerical routines to advance the time by a number of time steps given by the value `steps_per_update`, which is set by the CONSTANT command.

WORKSPACE:
input: three double precision floating point values x, y, and z
output: none
effects: instructs the numerical routines to change the workspace dimensions to x, y, and z.

The control language also supports the following query commands by which the user interface can obtain information from the numerical routines:

QUERY_ALL:
input: none
output: an integer n specifying the number of single nodes currently present in the simulation; n integers specifying the indices of the single nodes; $3n$ double precision floating-point values specifying n positions for the single nodes; $3n$ double precision floating-point values specifying n rotations for the single nodes; and a sequence of integers equal in number to the number of edges projecting from the n single nodes, with each integer 1 if the corresponding edge is bound to another edge and 0 otherwise.
effects: none

QUERY_CONFS:
input: none
output: an integer n specifying the number of variable nodes present in the simulation and n integers each specifying the conformation of the corresponding variable node.
effects: none

QUERY_CONSTANT:
input: an integer i
output: an integer or double precision floating-point value. The output value depends on the integer input, i, which specifies a global constant, as described in the following list of allowed values, output formats, and functions:

- 0-2: double precision floating point, currently unused
- 3: double precision floating point, specifies the time step
- 4: double precision floating point, specifies translational error tolerance
- 5: double precision floating point, specifies temperature
- 6: integer, specifies the number of steps executed for each STEP command issued
- 7: double precision floating point, specifies the value of ε, a constant used by some mathematical routines to represent a number "close to zero"
- 8: double precision floating point, specifies the damping value
- 9: double precision floating point, specifies rotational error tolerance

effects: none

QUERY_EDGES:
input: none
output: an integer n specifying the number of edges in the simulation and n integers each specifying the binding partner of the corresponding edge or -1 if the edge has no binding partner.
effects: none

QUERY_NODE:
input: an integer n
output: three double precision floating point values specifying the position of node n, three double precision floating point values specifying the velocity of node n, three double precision floating point values specifying the rotation of node n, and three double precision floating point values specifying the rotational velocity of node n.
effects: none