Automated Self-Assembly Programming Paradigm

by Lin Li, BSc.

Thesis submitted to The University of Nottingham
for the degree of Doctor of Philosophy

May 2008

Abstract

Self-assembly is a ubiquitous process in nature in which a disordered set of components autonomously assemble into a complex and more ordered structure. Components interact with each other without the presence of central control or external intervention. Self-assembly is a rapidly growing research topic and has been studied in various domains including nano-science and technology, robotics, micro-electro-mechanical systems, etc. Software self-assembly, on the other hand, has been lacking in research efforts.

In this research, I introduced Automated Self-Assembly Programming Paradigm ($ASAP^2$), a software self-assembly system whereby a set of human made components are collected in a software repository and later integrated through self-assembly into a specific software architecture. The goal of this research is to push the understanding of software self-assembly and investigate if it can complement current automatic programming approaches such as Genetic Programming.

The research begins by studying the behaviour of *unguided* software self-assembly, a process loosely inspired by ideal gases. The effect of the externally defined environmental parameters are then examined against the diversity of the assembled programs and the time needed for the system to reach its equilibrium. These analysis on software self-assembly then leads to a further investigation by using a particle swarm optimization based embodiment for $ASAP^2$. In addition, a family of network structures is studied to examine how various network properties affect the course and result of software self-assembly. The thesis ends by examining software self-assembly far from equilibrium, embedded in assorted network structures.

The main contributions of this thesis are: (1) a literature review on various approaches to the design of self-assembly systems, as well as some popular automatic programming approaches such as Genetic Programming; (2) a software self-assembly model in which software components move and interact with each other and eventually autonomously assemble into programs. This self-assembly process is an entirely new approach to automatic programming; (3) a detailed investigation on how the process and results of software self-assembly can be affected. This is tackled by deploying a variety of embodiments as well as a range of externally defined environmental variables. To the best of my knowledge, this is the first study on software self-assembly.

Publications Produced

While pursing this Ph.D. research programme, I produced several publications that represented my ongoing research for the degree. The following publications are based on my research conducted for this thesis:

L. Li, N. Krasnogor, J. Garibaldi. Automated Self-Assembly Programming Paradigm: Initial Investigation. In Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems, pages 25-34, Potsdamn, Germany, 2006. IEEE Computer Society.

L. Li, N. Krasnogor, J. Garibaldi. Automated Self-Assembly Programming Paradigm: A Particle Swarm Realization. In Proceedings of the Workshop on Nature Inspired Cooperative Strategies for Optimization, pages 123-134, Granada, Spain, 2006.

L. Li, N. Krasnogor, J. Garibaldi. Automated Self-Assembly Programming Paradigm: The Impact of Network Topology. Special Issue on Nature Inspired Cooperative Strategies for Optimization in the International Journal of Intelligent Systems, in press, 2007

L. Li, P. Siepmann, J. Smaldon, G. Terrazas and N. Krasnogor. Systems Self-Assembly: Multi-Disciplinary Snapshops, chapter Automated Self-Assembling Programming, Elsevier, In press, 2007.

Acknowledgements

For me, it is the most challenging thing to accomplish Ph.D. I have paid a lot of efforts into it, but I simply would not be able to achieve this without the help of so many people. I am very fortunate to have all the kind and generous help from you.

I would like to express my faithful thankfulness to my academic supervisors, Dr. Natalio Krasnogor and Dr Jon Garibaldi. In particular, I want to show my greatest gratitude to my primary supervisor Dr Natalio Krasnogor, who has been my supervisor since I was an undergraduate student. I know that a “thank you” is simply not enough, but thanks so much for your patience, your encouragement, your help and yours 5 years of supervision on my work. Even after I was late for my thesis submission, you are still encouraging me and providing me with all the help I ever needed. Thanks for all the weekends meetings, I am so fortunate to have you as my supervisor.

I want to express my gratefulness to the entire ASAP group for providing me the very best environment for doing a PhD.

I would like to thank my friends Jason Atkin and his wife Rachel Atkin. Although you are so busy with your own thesis correction and your RA job, you devoted your time on my thesis correction. Your kind and generous help really means a lot to me. Thank you, my friends!

To my friends Chahao Psu, German Terrazas, thanks for bringing me support, joy, and happiness for all these years while I was doing my PhD.

To my girl friend, Sky (Yijia Li), thank you as you give me love, sweetness, and a lot of happiness. Thanks to all your support and understanding before my thesis submission, the busiest time in my life so far.

Also, a thanks to my friend Yizhe Song (yes, you of course, how can I ever forget?). For all the years we are in the UK, we have finally made it this far. And since I have finished my doctoral dissertation while you are about to start yours, I want to tell you: hold on to it, I am sure you can finish it.

Finally, I would like to express my sincere thankfulness to my parents who provide me unlimited support all the time. Without them, I would not be the person I am today. Although they are not staying in England with me, I always feel you are standing by my side and giving me love and strength. Thank you are just too small words to express my thankfulness to you.

To my parents, Mr Weiliang Li, and Mrs Guoying Zhu

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis examines the development of software self-assembly. Self-assembly is a ubiquitous process in nature in which a disordered set of components autonomously assemble into a complex and more ordered structure. Components interact with each other without the presence of central control or external intervention. I present in this thesis the Automated Self-Assembly Programming Paradigm ($ASAP^2$), a software self-assembly model that simulates a self-assembly process for software components. An introduction to self-assembly is given next, followed by an explanation of the research goals in this thesis.

1.1 Dissertation Scope and Goals

Self-assembly is a general term used across various domains from molecular nano-science to biology and astronomy. Self-assembly is a process generally recognized as the autonomous formation of complex structures using elementary components.

Molecular self-assembly [23, 95] is one of most popular studied fields of self-assembly. As a design and engineering principle, self-assembly has been applied to robotics

[84, 97], mechanical systems [80], nanotechnology [23, 69, 95], and in a variety of other fields [61, 83, 94]. Although self-assembly has been studied in many research areas, there has been a lack of research effort on software self-assembly. The work on self-reconfigurable software [5, 48] and self-adaptive software [24], are few examples of recent efforts from a software engineering perspective. These incorporate some, but not all of the potentially appealing features behind self-assembly.

Although there has been an enormous amount of research on the evolvablity of complex program structures (for example all the research on genetic programming (`http://www.cs.bham.ac.uk/~wbl/biblio/`), here I propose to take a step back from the investigation of automatic program synthesis by evolutionary methods such as Genetic Programming [49, 78], or grammatical evolution [79] and to focus instead on the role that self-assembly could have in the automatic synthesis of program parse trees. The motivation for this is that there are strong arguments claiming that life originated from the interactions of inanimate matters through self-assembly. And that self-assembly preceded evolution: replicators undergoing natural selection needed for evolution to be kick started by self-assembly first. Hence, rather than using an evolutionary approach, I intend to focus on software self-assembly instead.

I propose in this thesis that software self-assembly should be vigorously investigated. The investigations shall not only aid the understanding of how self-assembly can be used in the automatic generation of programs, but also provide insights into how self-assembly can be complementary to existing automatic programming methodologies such as GP and grammatical evolution. To be more specific, this thesis is composed of the following studies:

- Propose a software self-assembly model that mimics the self-assembly of systems, both natural and artificial. Software self-assembly is expected to inherit *some*, but probably not all, of the features in natural self-assembly.
- Study software self-assembly using different embodiments, as well as different environments.
- Investigate static (i.e. close to equilibrium) software self-assembly and dynamic (i.e. far from equilibrium) software self-assembly.

1.2 A Note on Methodology

To study the points mentioned above, different methodologies are used.

The first point is tackled by a literature review. This includes examples of both natural and artificial self-assembly systems, as well as self-assembly as a design principle. The background knowledge obtained from the literature review is then integrated into a model for the automatic generation of programs using self-assembly.

The second and third points, are carried out by systematic experimentation and analysis of the models proposed. In order to gain a good understanding of software self-assembly in different circumstances, a range of embodiments are used to guide the software self-assembly process, as well as a variety of environments where self-assembly takes place.

1.3 Structure of Thesis

This thesis is organized as follows:

Chapter 1 gives a brief introduction on the objectives and contributions of this thesis (this chapter).

Chapter 2 gives an overview and background information on self-assembly.

Chapter 3 introduces the Automated Software Self-Assembly Programming Paradigm ($ASAP^2$), a model that simulates a self-assembly process for software components. Software components are treated as gas molecules and their interactions, within a confined area with specific temperature and pressure constraints. This gives rise to a variety of program *architectures*. Experimental results are presented to show how different factors affect the efficiency of the software self-assembly process and the diversity of the self-assembled programs.

Chapter 4 presents an extended $ASAP^2$ model based on particle swarm intelligence, illustrating how software self-assembly can be guided using different embodiments. Experimental results indicate an improvement in software self-assembly efficiency in terms of average time to equilibrium at the expense of diversity of generated structures.

Chapter 5 extends the $ASAP^2$ model, by embedding it into a network of compartments. A family of graph structures is used to illustrates what and how various network properties affect the course and evolution of software self-assembly.

Chapter 6 shows how the self-assembly dynamics of software components change within a network that is kept far from equilibrium.

Hence the dissertation progresses from simple to complex settings by first considering a single compartment and total random walks. Then it introduces swarming, followed by a complex network structure. But all of these systems are near equilibrium. Finally the network topology is pushed far from equilibrium and the impact on $ASAP^2$ investigated.

Chapter 7 summarizes this thesis and overviews various possible future work on software self-assembly.

1.4 Contributions

This thesis has made the following contributions:

1. A survey and review of natural and artificial self-assembly systems demonstrates the issues and advantages of self-assembly as a design and engineering principle for software engineering.

2. A software self-assembly model is presented, in which software components interact with each other and self-assemble into programs.

3. An analysis using various embodiments demonstrates how software self-assembly is affected under different externally defined environments.

4. An analysis of the topological impacts on software self-assembly suggests that network properties can affect the results of self-assembled programs.

5. $ASAP^2$ model is developed to study on software self-assembly close to equilibrium, as well as far from equilibrium.

To the best of my knowledge this is the first time that self-assembly - a ubiquitous natural phenomenon - has been proposed and studied as a (potentially) viable alternative to other automated program synthesis methodologies like, for example, genetic programming.

Chapter 2

Background Information

In this chapter, a brief review of self-assembly is presented. This chapter starts with an overview of the concepts and features of self-assembly in section 1. In section 2, a taxonomy of self-assembly systems categorized by their design in various domains is reviewed, thus giving an overview on (possible) design of software self-assembly systems. In section 3, current popular automatic programming approaches are briefly reviewed. The aim is to find out how self-assembly can be used to complement the existing automatic programming approaches. In Section 4, software self-assembly is introduced, with the research motivation, scope and goals explained. And section 5 summarizes this chapter.

2.1 Overview of Self-assembly

2.1.1 Introduction

Self-assembly is a ubiquitous process in nature in which a set of disordered pre-existing components assemble to one or more complex and ordered structures. The final structure is encoded in the design of components, in their interactions and in the environment in which the components live. Neither a central control mechanism nor human intervention is needed in this process. Components are generally autonomous and do not have pre-existing plans of how to reach the final structure. During the self-assembly process, components can only interact with each other and their local environment.

The self-assembled structures are determined by the statistical exploration of alternative configurations among components. More specifically, the information on how to assemble the final products is implicitly encoded in the way components interact with each other, and these interactions are embodied in the structures and properties of the individual components and the environment where they live in. Hence, as an engineering methodology, the design of the individual components and their environment is the key to successful control of self-assembling systems.

Nature presents vast examples of both organic and inorganic self-assembly systems at all scales. Amphiphilic molecules is one popular example of molecular self-assembly. These are comprised of two ends: the hydrophobic ends tend to repel water molecules and to be close to similar chains. On the contrary, the hydrophilic heads tend to be close to water molecules. As can be seen from Fig. 2.1, the circle-shaped ends are hydrophilic heads and the tails are hydrophobic heads. As a result, various structures can be self-assembled when amphiphilic molecules are placed into water, like for example lipid membrane, giant vesicles, micelles, bilayers, etc [35, 36, 42, 91].

Protein folding [81] is another example of organic self-assembly: the sequentially arranged amino acids self-assemble (i.e. folds) into a three-dimensional shape of the protein known as the native or tertiary structure. The final three dimensional folded structure

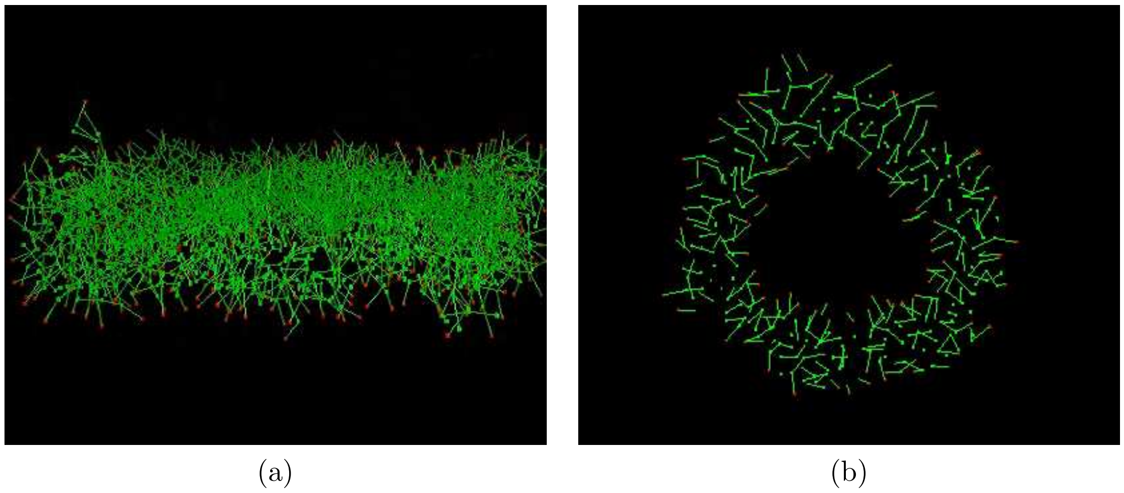


FIGURE 2.1: (a) Miscelle structure and (b) Bilayer structure formed by amphiphilic molecules when they are placed into water

is determined by the amino acids sequence as well as their interacting environment (e.g. solvent molecules, temperature, etc). In turn, the amino acids interactions are constrained by their polarities relative affinity [56].

On the other hand, flocking birds and schooling fish are examples of self-assembly systems in a macroscopic scale in nature. The movement of individuals in the group is dependent on the other members of the group as well as its own experience. Thus, individuals self-assemble to form a movement pattern as an emergent behaviour.

2.1.2 A closer look at the features of self-assembly

Although systems where self-assembling processes manifest themselves are remarkably varied, some common principles for self-assembly are starting to be discerned [44, 90]. Systems that self-assemble have some basic features in common: (A) Firstly, the system must be decomposable into 2 or more components. (A system of one component is, by definition, already assembled.) (B) Under certain circumstances the same set of components could be used to build a variety of structures and a goal structure can be composed by different combinations of components. (C) Each component should have the ability to compute and communicate (albeit in a limited way) with other components. (D) The constituent components are assembled together in various configurations, so that stable configurations form, tend to persist, and eventually become predominant [15]. The self-assembled system can be thought as the “halting state” of a distributed computation.

Self-assembly can be seen as an advantageous manufacturing process because given an appropriate selection of components and careful design of their interactions, components will autonomously assembly into a desired system. In addition, self-assembly systems are regarded as being robust and versatile [37]. These two features come from the fact that self-assembly involves the utilization of a potentially large, perhaps simple, set of components where only some will be involved in constructing the final structure. Self-assembly systems are versatile because a given structure can be achieved using different configurations of components, that is, it may be prone to alternative specifications [37]. Self-assembly systems are also considered to be robust because, if a certain part of the system fails, other components can be used to replace the failed part so as to ensure the functionality and integrity of the system as a whole.

In addition, systems that are sufficiently large and connected are usually complex. While current engineering methodologies uses a “top-down” approach which rely heavily on the precise design of each components and their interactions, it can be even harder to model the components and their interactions if these systems exhibit emergent behaviour. Self-assembly on the other hand can be seen as a distributed, not necessarily synchronous "bottom-up" design principle. There is no central control mechanism or master plan for components to follow in order to achieve the intended system. Instead, the control mechanism is distributed across the components, which have very limited computational capability and local sensing to instruct their behaviour under a reduced set of well defined conditions. It has been argued in [3] that "*intricate self-assembly processes will ultimately be used in circuit fabrication, nano-robotics, DNA computation [54], amorphous computing [3] etc*".

Albeit the advantages of self-assembly as a design process described above, there are a number of difficulties in the design of self-assembly model. Some basic issues in self-assembly research are illustrated in [37]:

- How to define a precise set of components and interactions that will result in a given global structure.
- How to avoid local optimal. (i.e. undesirable intermediate states)
- How to achieve robustness of the self-assembly system. The nature of self-assembly systems brings the advantage that the same set of components can form different goal structures. However, this can also mean that an optimal global structure may easily "downgrade" to a less satisfying structure due to environmental disturbances.

The first two can be summarized as the difficulty of "getting there", where the third one can be seen as the difficulty of "staying there".

2.2 A Brief Review on Self-Assembly Applications

Self-assembly is currently an active research area that has been studied in a wide range of areas including nanotechnology [23, 69, 95], biochemistry [23], artificial physical life forms

[33], mechanical system [80], robotics [7, 84, 86], network design [61, 83]. Albeit the large amount of research on self-assembly in literature, they can be categorized in certain ways. In this section, I introduce a conceptual framework for the design of self-assembly systems, followed by a short review on the design of self-assembly systems in various domains.

2.2.1 Conceptual framework

Whitesides and Grzybowski [94] divided self-assembly systems into 2 main categories: static or dynamic. In static self-assembly systems, an equilibrium will eventually be reached when either the minimum energy has been achieved or no more binding among components is possible. Dynamic self-assembly does not have equilibrium of state and the ordered assembled structure occurs while minimum energy is obtained.

Besides the modeling and simulation of self-assembly systems in nature [42, 69], self-assembly is studied as a powerful engineering methodology. Hence, self-assembly can also be categorized according to the design approach. As has been mentioned before, the final self-assembled product is encoded in the interactions among the components themselves as well as the interactions with the external environment. The design of these interactions is usually either embedded into the careful design of each components (i.e. the geometry or the binding sites of components) or by specifying rules (with constraints) for the self-assembling components to follow. The latter approach uses different rules on the same components with a given goal, while the former approach focuses on the configuration of components with a given goal while the interaction rules do not change.

The reviewed systems below are among the vast number of self-assembly design problems in literature, but they are what I believe to be the most illustrative examples of self-assembly achieved by a taxonomy of design approaches. The software self-assembly system presented in this thesis is inspired from some but not all of those design principles, and I believe the ideas presented in these reviewed systems can provide insights for future research on software self-assembly.

2.2.2 Self-assembly by manual design of rules / constraints

Self-assembly in robotics is a relatively new concept. Such robotic systems consist of simple identical modules usually limited to local sensing. Modules can attach and detach from each other to alter their emergent overall shape / structure. Each robot uses a simple and distributed mechanism to enable communication. These robots dynamically adapt their shapes to achieve locomotion or desired structures as an emergent self-assembling behaviour.

In [97], a self-assembling robotic system capable of constructing three-dimensional shapes is presented. The system is composed of a large number of robots of rhombic dodecahedron shapes, a three-dimensional equivalent of hexagon. The system is based on a three-dimensional grid and the objective is to have all the goal slots on the grid filled by modules. Modules maneuver around each other and move toward the closest available goal location while receiving information from adjacent robots about the status of that goal. The various constraints placed on the robots are the key to the design of this self-assembling robotic system.

A goal ordering constraint, for example, ensures that all goal positions can be filled by specifying the order of the locations to be visited by modules. Moreover, in order to avoid modules moving to a same goal and collide with each other, a module can reserve a certain goal if it is near the goal location. If a goal is reserved by a certain module, other modules that take the same goal are forced to take another goal. In this way, the module that reserves the goal can eventually fill the location. Figure 2.2 is an example illustrating the process of components self-assembling into a cup shape structure in the 3-dimensional grid structure. Components that have not reached a goal location are shown in pink wired frames, and white solid objects otherwise.

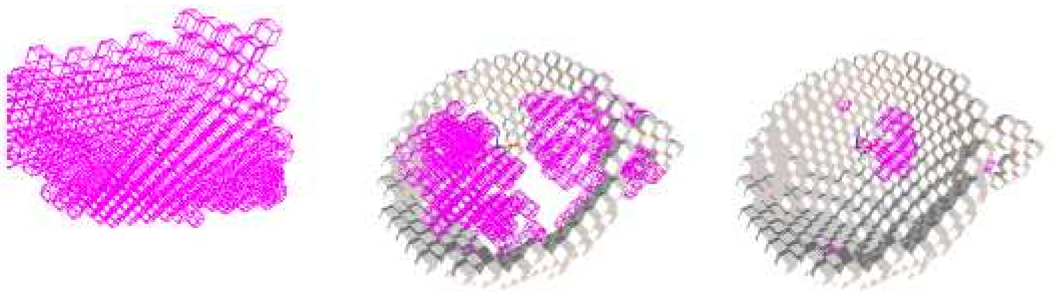


FIGURE 2.2: Rhombic dodecahedron robots self-assemble from a square place into a tea cup shape. (This figure is illustrated from [97].)

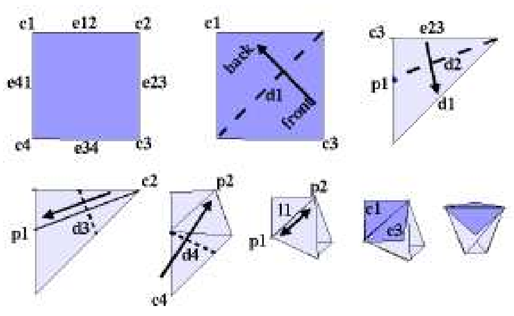
Bojinov et al. demonstrates another example of self-assembly applied in robotics in [7]. The goal is to have a set of dodecahedron shaped robots self-assemble to create a structure with the correct properties. The problem is tackled by manually providing a set of rules for modules, which in turn self-reconfigure themselves to a set of *modes* based on the rules. A mode is a set of pre-specified primitive operations including waiting, growing, and communicating with other modules. Results have shown that the developed system is capable of self-assembling into a range of structures aimed at satisfying a given task such as gripping an object. A similar approach is adopted in [86], where robots are assigned different roles in the system. A role represents the motion of a module and how it synchronizes with connected robots. It has been shown in [86] that caterpillar-like, snake-like quadruped walker locomotion has been successfully performed.

The self-assembly robotic systems presented in [7, 86] demonstrate *self-reconfiguration* properties of individual modules. The self-reconfiguration approach based on a set of manually specified rules has been shown to be capable of generating robust self-assembled structures. However, those self-assembly systems tackle very specific class of problems. This is because a set of rules have to be manually provided for the modules to work out its role / mode for every structure to be self-assembled.

2.2.3 A global-to-local compiler approach for self-assembly design

In [68], the authors present a programming methodology for self-assembling complex structures from vast numbers of locally-interacting and identically-programmed agents. The notion of morphogenesis and developmental biology is used to translate the global goal to local behaviour in order to achieve the global goal. Two examples are given to illustrate this methodology towards the design of robust self-assembling systems in [68]. In each case, the desired goal shapes are *compiled* to yield programs for the components, which then execute the generated program to self-assemble into the target shape.

The first example is to form shapes on a reconfigurable sheet composed of vast amount of simple, identical, and interacting agents. Those agents coordinate and fold the sheet along straight lines. In this system, the desired global shape is specified as a folding construction on a continuous sheet using an abstract geometry based programming language. The program for an individual agent is then automatically compiled from the global shape description. The agent program is composed of a set of primitives: gradients, neighbourhood query, cell-to-cell contact, polarity inversion and flexible folding. These primitives



(a)

```
;; OSL Cup program
;;---------------------
(define d1 (axiom2 c3 c1))
(define front (create-region c3 d1))
(define back  (create-region c1 d1))
(execute-fold d1 apical c3)

(define d2 (axiom3 e23 d1))
(define p1 (intersect d2 e34))
(define d3 (axiom2 c2 p1))
(execute-fold d3 apical c2)

(define p2 (intersect d3 e23))
(define d4 (axiom2 c4 p2))
(execute-fold d4 apical c4)

(define l1 (axiom1 p1 p2))
(within-region front
  (execute-fold l1 apical c3))
(within-region back
  (execute-fold l1 basal c1))
```

(b)

FIGURE 2.3: Programmable sheet: (a) Agents fold a programmable sheet into a cup structure (b) The corresponding program to create the folding. (This example figure is illustrated from [68].)

are used for communication between agents, measuring distance, as well as ascertaining local directions. Figure 2.3(a) illustrates this reconfigurable sheet folding into a cup shape, and the folding operations can be defined using a programming language called the Origami Shape Language (OSL) as shown in Fig. 2.3(b). OSL is based on a set of paper-folding rules and can be used to fold a large class of shapes [40].

The second example is self-assembling two dimensional structures through replication. A predetermined global shape is compiled to produce a program for a seed agent that *grows* the structure. The goal shape is represented as a network of overlapping circles, which is directly compiled from a graphical description of the goal shape. Agents replicate themselves to "grow" into this network of overlapping circles. Figure 2.4 illustrates this process in which agents replicate themselves and create a cross shape.

Both examples of self-assembly systems introduced in [68] compile the goal shape

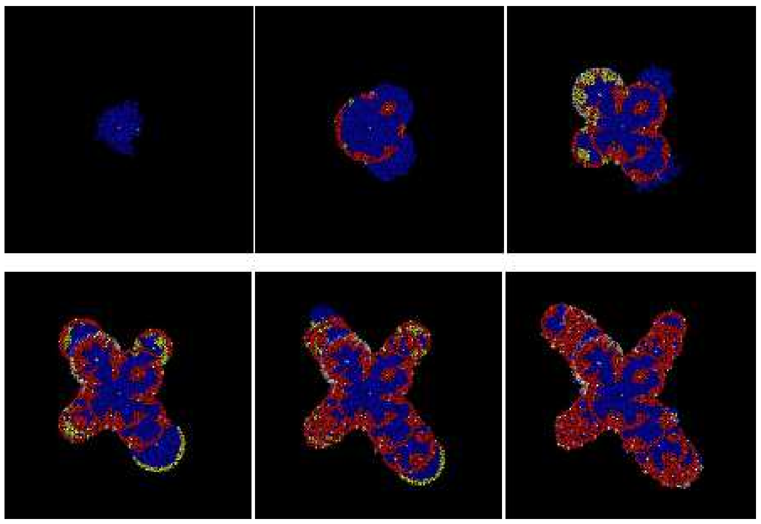


FIGURE 2.4: An example of components "grow" into a target structure using the compiler approach. (This example figure is illustrated from [68].)

and produce rules for components to coordinate themselves and self-assemble into the target structure. This self-assembly system has the important *self-healing* property that, if certain components fail, the remaining components will still adjust their interactions (as in the first example) or replicate themselves in the correct direction (as in the second example) to reach the desired shape.

A similar global-to-local compiler approach is presented in [43], where assembly components move to specific locations in 2D square lattices and connect to each other to achieve a pre-specified shape. Components move around randomly in the square lattices until an assembly has been formed. Each component has a state value and a look up table for the transition rule set (TRS). Components connect to each other according to the transition rules, which specify under what conditions the rules should be executed and the state value of the component is after executing the rule.

The TRS compiler specifies which of the neighbouring positions must be already connected in order for a component to be connected at that position. Firstly, components are assigned an identical state value. Next, a transition rule is randomly generated for each component and added to the TRS. The rule is then checked for consistency by verifying that only one transition rule is active at any location at any stage. This process repeats to update any inconsistent locations until a consistent TRS is built.

Experimental results shown in [43] indicate that the TRS compiler approach is capable of generating a consistent transition rule set for a variety of structures. However, this approach cannot handle some structures, for example those that do not have any adjacent neighbours.

The compiler approach has the advantage of modeling complex behaviour by specifying the desired goal structure at an abstract level. However, this approach can again have the danger of being domain-specific because the translation between the final structure and rules for components are entirely dependent on the problem it is working on.

2.2.4 Automated self-assembly design by evolutionary algorithms

Although major advances in robust self-assembly systems have been reported and reviewed as above, those self-assembly systems all tackle a very specific class of problems bound to analytical solution. It has been argued in [53, 88] that "*it is unrealistic to expect that each and every self-assembly system will have properties that make it agreeable to a manual design. And as the number of applications for self-assembly (and their complexity) increases in the near future, a point will be reached where humans cannot design the set of components and their interactions*".

What follows are a set of automated self-assembly designs using evolutionary algorithms. An evolutionary algorithm is a population based approach aiming to find a solution in the search space to a given problem by means of evolution. The principle of an evolutionary algorithm is based on the Darwinian theory of survival of the fittest and uses biological mechanisms such as recombination and mutation. Recombination is used to generate new solutions that are biased towards regions of the promising search space which has already been explored, and mutation is used to avoid local optima by exploring new regions. The following are some illustrative examples of using genetic algorithm to model self-assembly behaviour or to use self-assembly as a design methodology. Similarly to other problems tackled by evolutionary algorithm, the key is how to translate the design of self-assembly into an individual, as well as an evaluation function to measure the fitness of solutions represented by individuals in the population.

Protein folding prediction with Self-assembly

Protein folding prediction [81] is one of the most challenging problems in molecular biology. Protein folding can be regarded as an instance of self-assembly. Protein structure prediction is concerned with how the three dimensional folded structure is derived from the linear string of amino acids that constitute it. There are various approaches to the problem such as [6, 52, 89]. In [51, 52], the protein folding prediction problem is addressed by means of genetic algorithm applied to the automated design of protein self-assembly by Cellular Automata (CA). A cellular automaton consists of a collection of cells and a set of transition rules. Each cell has a state associated and transition rules update cell states depending on the state of the neighbours. CA are commonly used to model natural phenomena involving sophisticated rules. CA will not be introduced in great detail here as this is not the main focus and is not directly related to the research presented in this thesis. A brief review of cellular automata can be found in [66].

In [51], protein folding prediction is represented in a two dimensional lattice model, called the HP model, in which amino acids are classified in a simple manner, whether they are hydrophobic (**H**) or hydrophilic (**P**). Proteins are embedded in this model where the constituent amino acids are shown as squares (Fig 2.5).

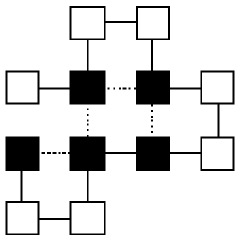


FIGURE 2.5: HP model based on rectangular lattices. White squares represent hydrophilics and black squares represent hydrophobics. (This figure is from [51].)

The operation that guides a linear sequence of amino acids into the folded structure is considered to be the transition rules of the CA. The states of a cell represent (A). the type of the amino acid, H or P, (B). move of the amino acid relative to the position of the previous one in the sequence, which are: Up, Down, Left, and Right. A rule determines a cell's relative position in the next time step. In turn, each rule depends on the neighbour's type and relative position. The aim is to find the set of transition rules that leads to the minimum energy of final folded structures.

A genetic algorithm is used to evolve the transition rule set, where each individual in the GA represents a set of rules. An individual is evaluated by running the CA on the sequential amino acids and obtaining the corresponding energy value of the folded structure, and the energy is determined by the number of bonds, number of collisions and compactness of the amino acids. Experimental results indicate that the genetic algorithm evolves rules from which the CA can reach high conformation values. The protein folding system reviewed here shows the automated design of self-assembling rules using an evolutionary approach can yield promising results.

Wang tiles self-assembly

Wang tiles [92] is a popular model used to study the complexity of self-assembly [85] as well as self-assembly as a design methodology [3, 54, 88]. A Wang tile system consists of a population of two dimensional squares, with a specific glue type associated to each side. Wang tile self-assembly model is initialized with system's temperature and an interaction matrix for the binding strength for each glue type. Tiles perform random motion in the plane and when two tiles collide, they will either self-assemble into an aggregate compound or continue their Brownian motion. That is, two components remain stationary if an assembly has been made. The decision on whether they will self-assemble is based on their glue types and strength on their colliding edges in relation to the system temperature. Figure 2.6 illustrates a snapshot of an example Wang tile system with some partially self-assembled structures.

In [87, 88], a genetic algorithm is used for the design of Wang tiles for the successful assembly towards the target shape. The population (Pop) is randomly initialized with a set of tiles with random colors associated to each side. That is, $Pop = \{Ind_1, Ind_2, ..., Ind_n\}$ with $Ind_i = \{T_1, T_2, ..., T_j\}$, where $T_k = \{t | t = (c_1, c_2, c_3, c_4)\}$ with c_i representing glue type on each side of the tile. The goal of the genetic algorithm is to find the population of components that will self-assemble into the desired shape. The evolutionary algorithm uses

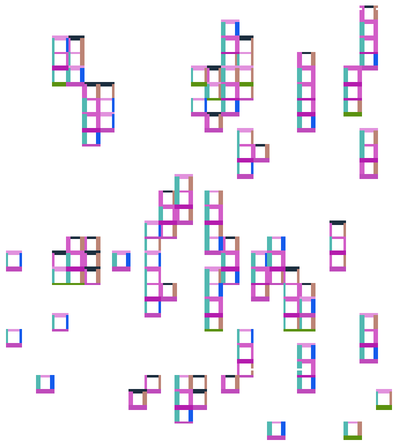


FIGURE 2.6: An illustration of a Wang tile self-assembly model. (This figure is from [88].)

the Morphological Image Analysis method as a fitness function. The simulation runs for a fixed number of steps, and the assembled shapes are compared with the target shape using compressing algorithms. The experimental results presented illustrates automated design of self-assembly components which yield successful self-assembly patterns.

As can be seen from the two self-assembly systems mentioned above, with a set of carefully defined individuals and fitness function, the evolutionary approach for the automated design of self-assembling systems is capable of finding a family of binding rules or components for the successful assembly of a target structure / shape. This self-assembly design principle is a potential research avenue for software self-assembly.

2.3 Current Automated Program Synthesis Methodologies

In this section, some of the most popular automatic programming techniques are reviewed. By studying how software self-assembly differ from them, I seek to answer how software self-assembly can *complement* and provide any insights to those existing techniques.

2.3.1 Genetic programming

A brief introduction on GP

Genetic programming (GP) is an evolutionary algorithm that generally breeds populations of computer programs to solve user specified problems. Genetic programming was pioneered by J. Koza [49] and has become one of the most popular methodologies for automated program synthesis since then [1]. Genetic programming is applied to a wide range of problems [2, 4, 50, 62].

Genetic programming uses a similar approach to genetic algorithms by using crossover and mutation on members of the population. The feature that makes genetic programming different from genetic algorithms is that the chromosomes use a program representation, i.e. syntax trees, and special operators rather than the simple sequential string representation as in genetic algorithm. Thus, the crossover operation is applied on an individual by replacing one of its nodes with a node from another individual in the population. As the individuals are based on a tree representation, replacing a node means replacing the (sub)branch of the tree.

As in a genetic algorithm, the population is initialized for GP to evolve the solutions. There are various methods to initialize the initial population and the common practice is to have a random and uniform distribution. With crossover and mutation, new individuals are generated at each time step and evaluated by a fitness function. The fitness function is the most important part in genetic programming and usually takes most of the computational time. The fitness function measures how well a program solves a given problem and it varies greatly depending on the problem it tries to solve. A fitness of an individual is typically decided by running the solution through one or more *fitness case(s)*. This process is iterated until the termination condition is satisfied. The termination condition can either be specified by a maximum number of iterations or when an ideal solution has been found. The following algorithm describes the above process.

Algorithm 1 Genetic Programming

```
1: Genetic_Programming
2: Initialize the population of solutions
3: At each time step
4: while Termination condition not met do
5:    Produce new individuals with the existing population and operators
6:    Place new individuals into the existing populations
7:    Assign fitness values to individuals and update the population
8:    Test termination condition
9: end while
10: Return the best (set of) solution(s)
```

Diversity in GP studies

One of the main challenges in tackling various GP issues is to avoid local optima. The loss of diversity during a GP run is regarded as the main reason behind premature convergence to local optima [12, 22, 30, 41]. This is because the individuals of the population get arbitrarily close to each other in the search space and converge to a local extreme.

Diversity in genetic programming thus measures the amount of variety between individuals of a population. A large variety of diversity measures exist to identify different aspects of the population structure in order to investigate the potential causes for the premature convergence that leads to local optima. For example, genetic variety [49] measures the number of genotypes in the population. The tree distance measure compares the syntactic difference between programs [74]. Diversity can also be measured as a proportion of the number of unique measure individuals over population size [45]. Finally, the phenotypic measure compares the number of unique fitness values in the population. It has been suggested that the phenotype diversity measure is more useful than genotype diversity in capturing the dynamics of the population [11, 45].

Various techniques are used to maintain and encourage diversity. One of the most popular techniques is fitness sharing [22], in which fitness is regarded as a shared resource of the population. Similar individuals share their fitness, and therefore the fitness of each individual is inversely proportional to the number of neighbours in the search space. Moreover, a fitness uniform selection strategy [39] was presented to encourage diversity. Island models are another commonly suggested ways to improve diversity [10].

2.3.2 Grammatical Evolution

Grammatical Evolution (GE) is a relatively new evolutionary computation technique pioneered by Conor Ryan, JJ Collins and Michael O'Neill. Grammatical evolution is an evolutionary algorithm that "*evolves complete programs in an arbitrary language using a variable length linear genome to govern the mapping of a Backus Naur Form grammar definition to a program.*" [72, 79].

A Backus Naur Form grammar can be represented by a system (N, T, R, S), where N and T represent the set of non-terminals and terminals respectively. R is a set of derivation rules that map the elements in N to T. S is subset of N, and it is a start symbol from which all programs are constructed. A simple example of Backus Naur form can be described as follows:

$$\begin{aligned}
T &= \{+, -, \times, /, log, x, 1\} \\
N &= \{< expression >, < binary\_operator >, < unary\_operator >, < var >\} \\
S &= \{< expression >\}
\end{aligned}$$

R is thus defined as:

$$\begin{array}{rcr}
< expression > & ::=< expression >< binary\_operator >< expression > & (0) \\
| & < unary\_operator >< expression > & (1) \\
| & < var > & (2) \\
< binary\_operator >::= & + & (0) \\
| & - & (1) \\
| & \times & (2) \\
| & / & (3) \\
< unary\_operator >::= & log & (0) \\
< var >::= & x & (0) \\
| & 1 & (1)
\end{array}$$

210	32	77	23	103	7

FIGURE 2.7: An example individual in GE

In GE, a chromosome consists of a sequence of binary codons, each of which denotes a selection of production rule. Figure 2.7 shows an example individual, where each number is the integer equivalent of the binary codon. The selection of a production rule is an iterative process on the specified grammar based on the codons in the chromosome. The mapping between codon and a production rule can be described as:

$$selected\ production\ rule \quad = \quad (codon\ integer)\ MOD\ (number\ of\ rule\ instances),$$

where the selected production rule denotes a selected rule in the current non-temerinal, and number of rule instances is the number of choices of rules for the current non-terminal. Given the example described in the above grammar, if the current non-terminal is $< expression >$, the selected production rule can be $< expression >< binary\_operator >< expression >$, $< unary\_operator >< expression >$, or $< var >$. The number of production rule instances for $< expression >$ is 3.

The iterative mapping process starts from S, and at the end of this rule selection process, the phenotype (final program) is generated. The following is a simple example to illustrate the mapping between the codons of integer and the selection of the production rules. Given the simple grammar described above and an individual as shown in Fig. 2.7, $< expression >$ has three rules to choose from and the first codon 210 gives selection rule 210 MOD 3 = 0. Hence rule 0 is used. Next choice for the first *expression* in the rule

$< expression >< binary\_operator >< expression >$, $32\ MOD\ 3 = 2$, therefore $< var >$ is used. This rule selection mechanism iteratively executes encodings in the gene until a complete program is formed.

A genetic algorithm is used to control the selection of rules in GE. GE separates genotype and phenotype by using linear genome rather than tree structure as in GP. In GP, the search algorithm operates on the same objects as the fitness function. In GE, however, while the fitness function is operating on the same phenotype (generated programs) as in GP, the search algorithm operates on the specified grammar, and hence reduces the search space. In addition, domain knowledge can be more easily incorporated by carefully user specified grammar.

2.4 Software Self-Assembly: Research Perspective

Inspired from a myriad of both natural and artificial self-assembly systems, I propose in this thesis a preliminary study on software self-assembly. Software self-assembly can be explicitly recognized as *a process within which software components autonomously integrate into one or more complex program architectures (parsing trees) using natural self-assembly as a metaphor.* Software self-assembly can be seen as a "bottom-up" manufacturing methodology as opposed to traditional software engineering techniques. I intend to push forward the understanding on software self-assembly, which breaks the tradition of **static** and **pre-specified** software architectures.

Software self-assembly differs from evolutionary approaches for automatic programming such as GP and GE in the following two main aspects. Firstly, software self-assembly depends on a set of assembling principles to guide the construction process while GP and GE is based on evolution of programs / grammars. As can be seen from the previous sections, evolutionary means is merely one of the embodiments that can be used to guide the self-assembly process. Various self-assembly design principles and approaches exist to guide a self-assembly process. Hence, it can be argued that software self-assembly is more general than Genetic Programming. Secondly, rather than trying to "evolve" simultaneously the components and the architecture as GP suggests, software self-assembly relies on a set of human made software components and subcomponents from where it can select the useful ones and search for the best architecture to integrate those components. The concept of a software component is general, such that it can be as simple as an assignment operation or arbitrarily large, such as a software package that takes user inputs and performs complex operations.

2.4.1 Research aims and motivations

This study, the first attempt at software self-assembly, is specifically concerned with *structure* rather than *function* of the assembled programs, given the persistent effort in the GP literature to improve trees' diversity. The functionality of assembled programs is clearly the ultimate and more challenging goal. Although the functionality of assembled programs is not dealt with in this thesis, the important correlation between structural diversity and quality of the solutions has already been pointed out in [30, 31].

In addition, it is widely argued that in nature, structure precedes function. There are strong arguments claiming that self-assembly occurs before evolution starts and the self-assembled **structures** provide the essential material for evolution to start [16, 63, 73].

It has been suggested that diversity and complexity of genomes arises from a hierarchy of interactions between genes and between interacting genes complexes [29, 64]. While Genetic Programming and Grammatical Evolution uses evolutionary means to achieve automatic program generation, I take "a step back in time" and focus on structure in software self-assembly as a precursor for novel functionality.

In particular, diversity of generated structures is important in the context of software self-assembly because it is expected that by being able to understand and control diversity, one could explore a larger sample of the space of potential program structures. This, in turn, could prove to be a key step towards achieving specific program functionality, as indeed has been shown for the case of GP [30, 31, 32]. Thus, the point I make in this dissertation is that what is important is to be able to predict, and ultimately control diversity. In turn this would allow to either increase it or decrease it depending on the specific requirement of the problem at hand.

2.4.2 Research methodology in details

I propose in this thesis a software self-assembly model which collects a set of human-made (or otherwise) software components from a software repository. Software components are later integrated through self-assembly into a software architecture. In this way, software self-assembly produces programs automatically without external intervention other than the pre-specification of the component set.

I systematically analyze how various embodiments and environments affect software self-assembly in terms of its process and the resulting program architectures. To be more specific, the following approaches are adopted to compare and analyze the dynamics of $ASAP^2$:

1. program gases versus PSO inspired approach: Program gas is a metaphor used for software components such that components perform random walk and their behaviour is similar (but not identical) to ideal gas molecules. PSO inspired approach introduces leaders and dynamic neighourhood.

2. single compartment versus a network of compartments: In a single compartment, software components move around in a single confined area and self-assemble to autonomously build programs. In a network of compartments, software self-assembly occurs simultaneously in the network and software components can freely move to a neighbouring compartment via a connecting edge.

3. close system versus open system: In a close software self-assembly system, software components are initialized at the beginning of the simulation and no components will be added or removed during the assembly process. Hence, an equilibrium will eventually be reached in a closed system. In an open system, on the other hand, software components are continuously added and removed from the system. Hence, the system is far from equilibrium.

The diversity of the assembled programs and the so-called "time to equilibrium" are tested under each comparison mentioned above [1]. I explain in detail the research motivation, system implementation and experimental results and analysis in the following chapters.

[1]Time to equilibrium analysis is replaced with diversity analysis at various time step in an open (dynamic) software self-assembly system because the system will not reach stable equilibrium while components are added and removed constantly

2.5 Summary

In this chapter, we concentrated on the study of self-assembly as a natural phenomenon as well as an engineering principle. Self-assembly is a robust and versatile process to build complex structures in nature, and it is also an advantageous fabrication process as an engineering methodology.

A taxonomy of design principles for self-assembly systems is studied. And it can be concluded that one of the most important issues in the design of a self-assembly system is to understand and utilize the translation from the interactions among a large set of components to the global behaviour of the system (i.e. the final assembled structure). While each component remains autonomous, limited to interactions in local environment, and with no pre-programmed master plan of the assembly of the final structure, the translation from global behaviour to local interaction is successfully handled by the design of rules or components that are involved in the self-assembly process.

It can be summarized that self-assembly can be achieved by the careful design of either the assembly components or their interaction rules. These can in turn be achieved by a taxonomy of approaches. As shown in [7, 86], manual design of assembly rules yields robust self-assembly results, however the systems lack generality and are usually highly problem specific. In [43, 68], a compiler approach is presented to translate the global goal into program / rules. Assembly components then execute the compiled rules to realize self-assembly of the goal structure. On the other hand, [52, 88] illustrate examples of automated self-assembly design using evolutionary approaches. The reviewed taxonomy of self-assembly design provides ideas and inspiration for the current development proposed in the following chapters, as well as ideas of possible future research on software self-assembly.

In addition, we surveyed current automatic programming approaches (GP and GE). In particular, diversity studies in GP is reviewed. This explains how studies on the diversity of self-assembled programs presented in the following chapters fit with the current research on automatic programming.

Chapter 3

Program Gases: A Software Self-assembly Approach with Unguided Dynamics

In this chapter, we introduce the Automated Self-Assembly Programming Paradigm ($ASAP^2$), a software self-assembly model which integrates software components and automatically builds program structures. The research focus at this preliminary stage is to study the behaviour of software self-assembly using *simple* dynamics. As ideal gases resemble to our desired system, the first $ASAP^2$ is based on a loose version of ideal gases. We show how different environments (as defined in details in the following sections) affect an *unguided* process of programs collisions and (eventual) self-assembly.

This chapter is organized as follows. In Section 1, we introduce ideal gas as a metaphor for our software self-assembly model. The $ASAP^2$ model and the system implementations are described in detail in Section 2. We explain how experiments are conducted and present results in Section 3. In Section 4, a predictive model for $ASAP^2$ is introduced and statistics to evaluate this predictive model are illustrated. Finally, Section 5 summarizes this chapter.

3.1 Program Gases: A Metaphor for Software Self-Assembly Based on the Equation of State for Ideal Gases

3.1.1 Kinetic theory of ideal gas

In 1738, Bernoulli published *Hydrodynamica*, explaining air pressure from a molecular point of view. Hydrodynamics laid the foundation of *kinetic theory of gases* by arguing that a gas consists of a great number of molecules performing random motion. Their interaction and motion results in pressure on the surface of the container of the gas, and the kinetic energy of their motion is perceived as heat. In 1857 German physicist Clausius published a theory that is nowadays known as kinetic theory.

An ideal gas (also called perfect gas) is one that exactly conforms to the kinetic theory. Kinetic theory of ideal gases is a theoretical model to study the behaviour of gas molecules[9, 98]. In this model, a large number of ideal gas molecules perform random motion in a closed container. Ideal gas molecules collide with each other and with the walls of the container. As molecules collide with the walls of the container, the collision results in a measurable pressure.

Ideal gas is a theoretical model and it differs from real gas molecules. The ideal gas model is based on the following assumptions:

- The total volume of particles are negligible in comparison with the size of the container, which means the average distance between molecules is significantly greater than their sizes.

- The collisions of gas molecules with each other and the wall of the container holding them are elastic. That is, the energy of the gas molecules is entirely translational kinetic energy as no deformation or chemical reaction is modeled.

- The molecules are perfectly spherical in shape, and elastic in nature.
- The average kinetic energy of the gas molecules depends only on the temperature of the system, meaning that molecules have a higher kinetic energy at a higher temperature.
- The time during collision of molecule with the container's wall is negligible as comparable to the time between successive collisions.
- The equations of motion of the molecules are time-reversible.
- The space is isotropic, i.e. it looks the same in any direction.

As is well-known, ideal gases have a simple equation of state relating the pressure on the wall of the container P, absolute temperature T, volume V and number of molecules n:

$$PV = nRT, \tag{3.1}$$

where R is a constant for each particular gas. This law describes the following relationship between Pressure (P) and the other environment variables: (1) If the temperature and volume remain constant, then the pressure of the gas changes is directly proportional to the number of molecules of gas present. (2) If the number of gas molecules and the temperature remain constant, then the pressure is inversely proportional to the volume. (3) If the volume and the number of gas molecules are kept constant, then the pressure will change in direct proportion to the temperature. [1]

3.1.2 Program gases

A loose version of ideal gases is used as a metaphor for $ASAP^2$ at the preliminary stage of the research [59]. This software self-assembly model was developed in order to study the behaviour of software self-assembly and how it can be affected by numerous factors.

In this $ASAP^2$ model, software components are treated as ideal gas molecules and the $ASAP^2$ model behaves in similar patterns. The interactions between software components gives rise to a variety of program *architectures*. More specifically:

- Software components are placed into a compartment within which they move randomly. The compartment used in software self-assembly plays the same role as a gas container. However, in order to simplify the domain, the compartment is two-dimensional rather than three-dimensional as in ideal gas theory.
- Software components are linked to molecules for which the probability of movement rises with temperature.
- The area of the compartment is a free parameter of the model.
- The temperature is another free parameter of the model.
- Finally, the number of copies of components placed within the compartment is the third and last parameter.

[1]However, note that real gases do not exhibit the properties listed in the above assumptions, and their behaviour deviates significantly from ideal gases at high pressure or low temperatures.

We measured to what degree this equation holds for $ASAP^2$. We investigated how the free parameters temperature, area, and number of components affect both the speed and diversity of self-assembled programs. Because ideal gas is used as a metaphor for $ASAP^2$, the equation of states for ideal gases is used as a crude approximation to link the three free parameters described above to the dependent variable, i.e. the pressure in the walls of the compartment exerted by the software components.

Although software components in $ASAP^2$ resembles the kinetic theory of perfect gases, it must be noted that program gases differ from an ideal gas in the following ways. Firstly, unlike the kinetic theory of a perfect gas where it is assumed that the physical system contains a large number of molecules, our software self-assembly system has a more limited number of software components. Moreover, software components may *bind* as a result of collisions within the compartment thus violating the assumptions of elastic collisions and chemical reactions in kinetic theory. Also, the sizes of the (partially) self-assembled components of a program gas grow as more components collide and get attached during the self-assembling process. Consequently, the assumption that the distance between molecules is far greater than their sizes is also violated. Therefore, it is expected that our software system is affected by the environment in a similar but not identical way as a perfect gas which follows Eq. 3.1.

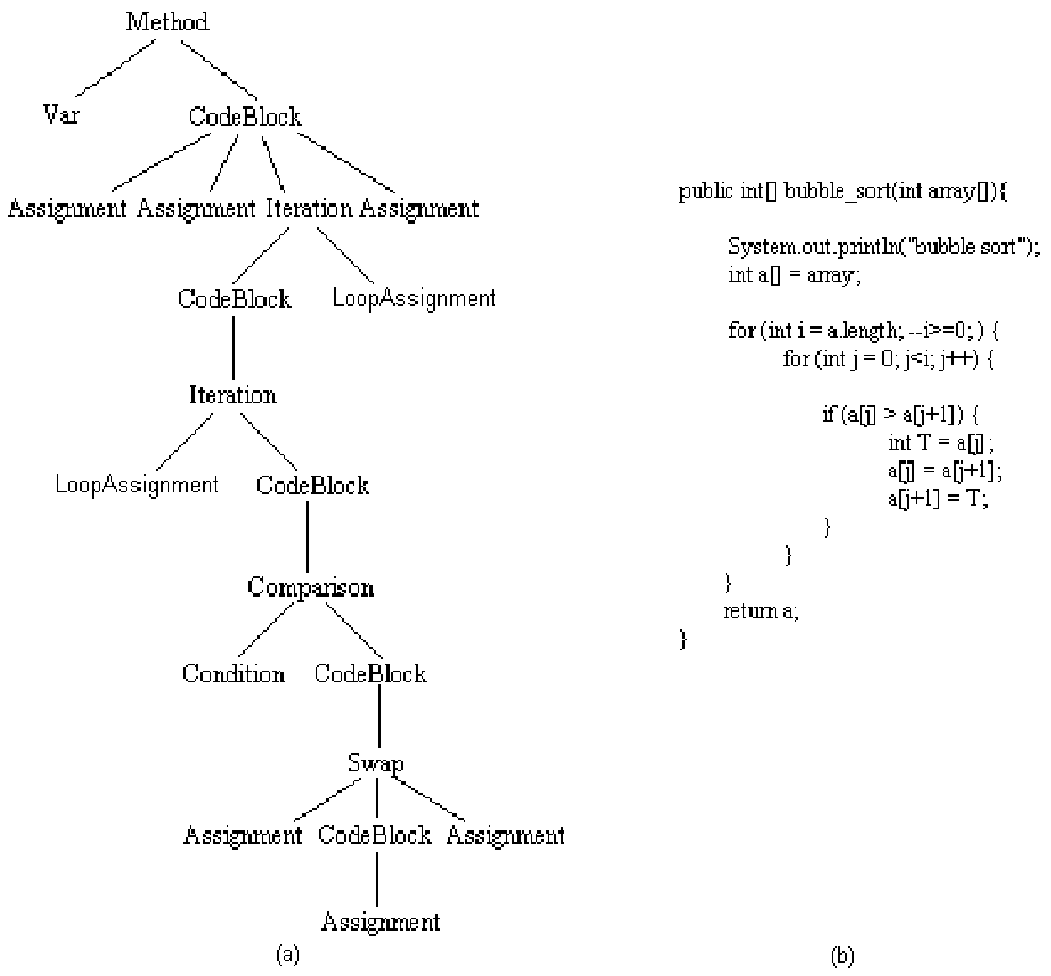
3.2 System Implementation

Self-assembly is a process in which a disordered set of components self-organize into a specific structure. Components interact with each other and form the global structure without external control. Software self-assembly simulates a self-assembly process for software components. Software components interact with each other within this process and are autonomously integrated into certain architectures.

The conventional definition of software components comes from software engineering discipline. A software component is a system element offering a predefined service and able to communicate with other components [65]. A software component in software engineering usually refers to service object written to some specification such as COM or Java beans.

In software self-assembly, software components are more general. In addition to the description given as a software engineering discipline, a software component can be referred to as blocks of specific code decomposed from a program. Hence, a software component can be arbitrarily large or small. A large software component can be a service provider in the form of an EJB object for a database connection, and a conditional statement is an example of a small software component. Albeit their difference in size, software components in software self-assembly should provide means for communicating and connecting to each other. In this research, however, the term software component refers to software codes decomposed from a given program. As the focus of this research is on the diversity and complexity of self-assembled programs, it is easier to measure various properties of self-assembled programs using small software components.

A software program can be syntactically represented by a hierarchical tree structure. Nodes in this parse tree represent software components. Figure 3.1 shows a bubble sort program with its corresponding tree representation. The construction of the tree is de-



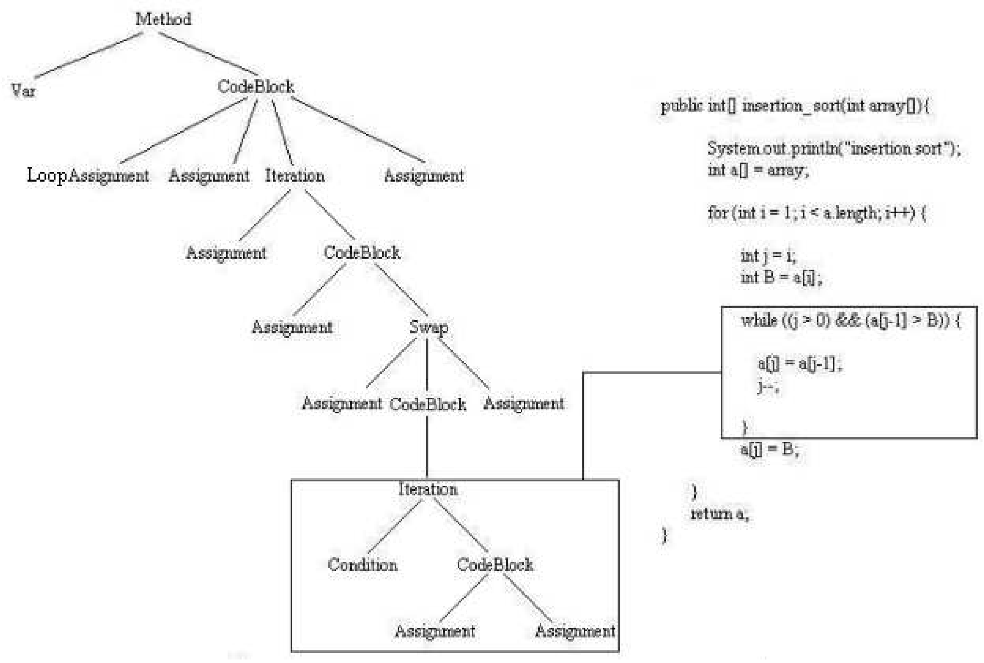
```
public int[] bubble_sort(int array[]){

     System.out.println("bubble sort");
     int a[] = array;

     for (int i = a.length; --i>=0; ) {
          for (int j = 0; j<i; j++) {

               if (a[j] > a[j+1]) {
                    int T = a[j];
                    a[j] = a[j+1];
                    a[j+1] = T;
               }
          }
     }
     return a;
}
```

FIGURE 3.1: Bubble sort program code and its parsing tree

pendent on the ways software components are categorized and defined. An example Backus Naur Form is illustrated to show the productions rules upon which the parsing tree shown in Fig. 3.1 is constructed.

$$
\begin{array}{rcl}
< Method > & ::= & < Var >< CodeBlock >< expression > \\
< CodeBlock > & ::= & < Assignment >< Iteration >< Assignment > \\
 & | & < Iteration > \\
 & | & < Comparison > \\
 & | & < Swap > \\
 & | & < Assignment > \\
< Iteration > & ::= & < LoopAssignment >< CodeBlock > \\
< Comparsion > & ::= & < Condition >< CodeBlock > \\
< Swap > & ::= & < Assignment >< CodeBlock >< Assignment > \\
< Var > & ::= & \text{int array[]} \\
< Condition > & ::= & \text{a[j] > a[j+1]} \\
< LoopAssignment > & ::= & \text{int i = a.length; –i>=0;} \\
 & | & \text{int j=0; j<i; j++} \\
< Assignment > & ::= & \text{System.out.println(”bubble sort”);} \\
 & | & \text{int a[] = array;} \\
 & | & \text{return a;} \\
 & | & \text{int T = a[j];} \\
 & | & \text{a[j+1] = T;} \\
 & | & \text{a[j] = a[j+1];}
\end{array}
$$

Three standard sorting algorithms are used as software repositories of components for all the simulations and experiments in this thesis. Hence, software components are categorized and defined based on the structure of these sorting algorithm: *Comparison, Iteration, Method, Swap, CodeBlock, Assignment, Condition, LoopAssignment.* Among the 8 component types, *Assignment*, *LoopAssignment* and *Condition* are the leaf nodes in the parse tree representation. Figure 3.2 shows a mapping of a node on the parse tree to the original insertion sort program.



```
public int[] insertion_sort(int array[]){

    System.out.println("insertion sort");
    int a[] = array;

    for (int i = 1; i < a.length; i++) {

        int j = i;
        int B = a[i];

        while ((j > 0) && (a[j-1] > B)) {

            a[j] = a[j-1];
            j--;

        }
        a[j] = B;

    }
    return a;
}
```

FIGURE 3.2: Mapping between a software component of type *Iteration* and its code in the original insertion sort program.

Software components are manually decomposed from the three sorting algorithms and are stored into a software repository. Software components are then retrieved from this repository and later integrated through self-assembly. In turn a software component is represented by a “wrapper” that is used to mediate the (eventual) assembly of a component with another component. A wrapper consists of a software component that is “wrapped” inside and ports that act as loci where other wrappers bind. There are two types of ports, input ports and output ports, and all ports have a corresponding data type. The data type on the output port is associated with the type of the software component, and the data types on the input ports are determined by the types associated with their expected child nodes in the parse tree representation. In order to study only valid or meaningful configurations, an input port can only connect with an output port of the same data type. That is, the system components are “non-elastic” when their ports are incompatible and “elastic” in the other case. A wrapper can have only one output port and any number of input ports (including zero). Those wrappers that have one or more input ports can be seen as the internal nodes in the parse tree. On the contrary, wrappers with zero input ports are leaf nodes. Figure 3.3 illustrates an abstract view of a wrapper.

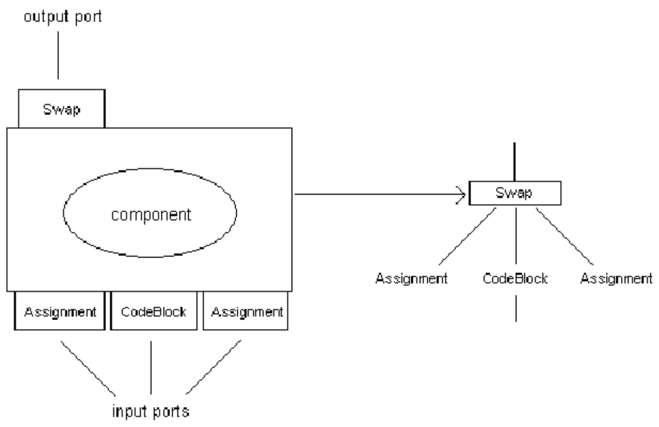


FIGURE 3.3: A wrapper that represents the node Swap in the tree

Figure 3.4 provides an overview of the $ASAP^2$ system. Software self-assembly starts by placing all the wrappers (i.e. software components) from a given software components repository into the compartment. Once in the compartment, the wrappers move

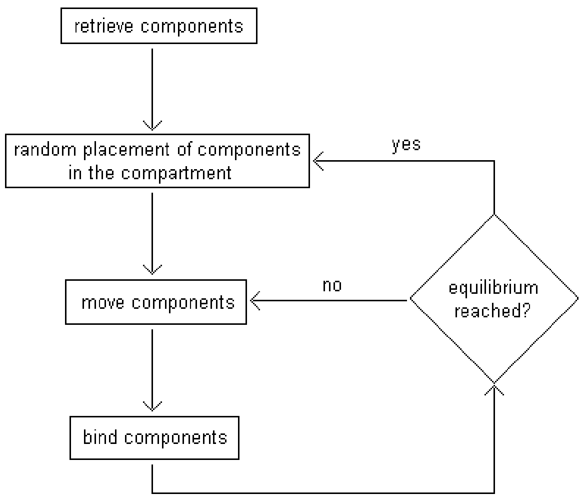


FIGURE 3.4: Flow chart for $ASAP^2$ system implementation.

randomly with a probability that is a function of the temperature. Each wrapper has a spatial coordinate value (x, y) to record its position at time t_i. When two wrappers are within certain Euclidean distance d_δ and the data types on their vacant input and output ports match, they self-assemble. In this way, a data type constraint is imposed to the model in such a way that if two wrappers with incompatible ports collide they repel each other. The threshold distance (d_δ) is directly proportional to the wrappers' size (i.e. number of nodes in their tree representation). The binding algorithm is formally detailed as follows:

In the graphical user interface shown in 3.5, 3.6, a circle represents a wrapper, and different colours are used to represent different wrappers classes. The class of a wrapper is determined by the type of software components it represents, i.e. one of the types illustrated in Table 3.4. The left panels in Fig. 3.5, 3.6 show the container where software self-assembly occurs, while in the right panels the program code associated to a user-selected self-assembled program are displayed. As an example of the software self-assembly process,

Algorithm 2 Binding algorithm: d_δ is a threshold distance for components to bind, and μ_i indicates displacement of component C_i

```
for (every pair of components c1 and c2 in the component set $S_A$) do
  $d_\delta$ = max( size(c1), size(c2) )
  if ( distance (c1, c2) $\leq d_\delta$ ) then
    success = attemptBind(c1, c2)
    if ( success == true ) then
      add assembled component to the set $S_A$ and delete c1, c2.
    else
      c1.move(currentLocation, $\mu_1$)
      c2.move(currentLocation, $\mu_2$)
    end if
  end if
end for
```

Fig. 3.5 shows an early stage of software self-assembly where few partially self-assembled programs are formed. Figure 3.6 shows a latter stage of software self-assembly where several larger tree-shaped aggregated structures have been formed. The simulation is deemed to have reached *equilibrium* when no more bindings can occur between the remaining (partial) self-assembled parse trees thus equilibrium is the stopping condition for the simulations. Checking whether equilibrium has been reached, i.e. no pair of components can bind, could be a computationally expensive task, especially when a large number of components exist in the compartment. Hence, some caution must be used when checking for equilibrium so as to avoid incurring an $O(n^2)$ cost at each iteration.

Formally, a system has reached its equilibrium if for any of the output ports of type t_i, there are no available input ports of the same type to bind with, and vice versa. Hence, evaluation on whether the system has reached its equilibrium can be expressed mathematically as:

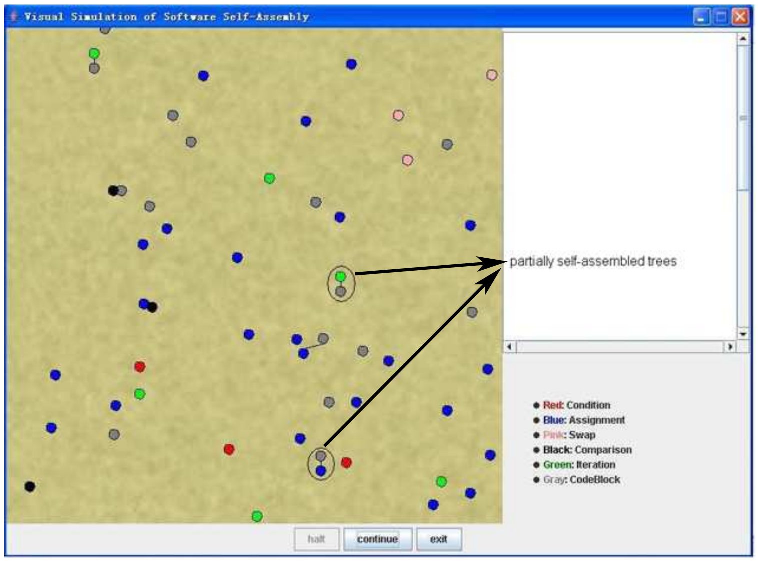
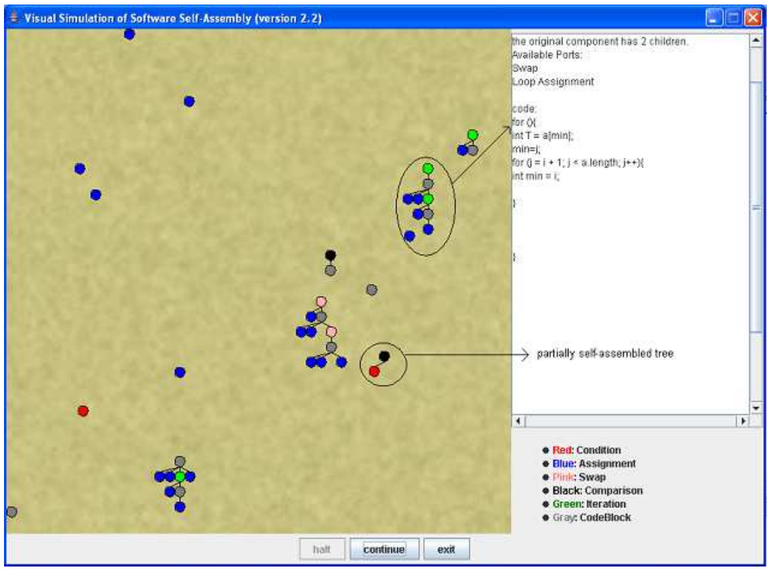


FIGURE 3.5: Early stage of software self-assembly.



```
the original component has 2 children.
Available Ports:
Swap
Loop Assignment

code:
for (){
int T = a[min];
min=j;
for (j = i + 1; j < a.length; j++){
int min = i;

}


}
```

FIGURE 3.6: Latter stage of software self-assembly.

$$
\begin{aligned}
Equilibrium \quad = \quad & (input\_port(t_1) == 0 \vee output\_port(t_1) == 0) \\
& \wedge (input\_port(t_2) == 0 \vee output\_port(t_2) == 0) \\
& \wedge (input\_port(t_3) == 0 \vee output\_port(t_3) == 0) \\
& \wedge \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
& \wedge (input\_port(t_n) == 0 \vee output\_port(t_n) == 0)\,,
\end{aligned}
$$

where $input\_port(t_i)$, $output\_port(t_i)$ represents the number of available input and output ports of type t_i respectively. The number of available input / output ports of each type are stored in a table and initialized at the start of the simulation when components are placed into the compartment. The table is updated while binding action occurs. To examine whether the system has reached its equilibrium, the above mathematical expression is evaluated and this makes the computational cost linear in the number of port types, rather than square as a function of components as a naive implementation could require. However, there is a problem associated with this method. An example is shown in Fig. 3.7, supposing the aggregate structure being the only remaining wrapper in the compartment. In this example, the number of available input port and output port of type *Iteration* are both equal to one. According to the above algorithm, the equilibrium condition is not satisfied. However, the system has reached equilibrium because there are no other wrappers to bind with and this remaining wrapper cannot bind to itself.

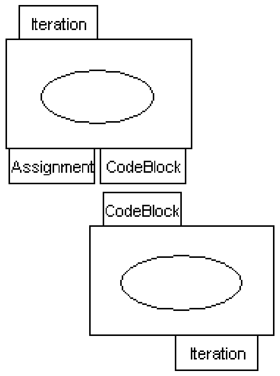


FIGURE 3.7: An example to show why merely keeping record of available input ports and remaining wrappers is insufficient.

As no recursive binding is allowed, the problem is due to having available input and output ports of the same type existing in the same structure. In order to solve this, the number of input ports falling into the above category should be monitored while binding actions occur. Those input ports that match the type on the output port in the same structure are referred as "circular port". The modified algorithm for testing the equilibrium condition is shown as the following:

$$\begin{aligned}
Equilibrium \;=\; & (input\_port(t_1) - circular\_port(t_1) == 0 \vee output\_port(t_1) == 0) \\
& \wedge (input\_port(t_2) - circular\_port(t_2) == 0 \vee output\_port(t_2) == 0) \\
& \wedge (input\_port(t_3) - circular\_port(t_3) == 0 \vee output\_port(t_3) == 0) \\
& \wedge \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
& \wedge (input\_port(t_n) - circular\_port(t_n) == 0 \vee output\_port(t_n) == 0)\,,
\end{aligned}$$

where $circular\_port(t_i)$ gives the number of available input ports that have the same type as the output port in the wrapper.

A step-by-step example is illustrated in order to explain how the equilibrium condition is evaluated based on the above formulae. Figure 3.8 shows a compartment containing initially wrappers of types *Comparison*, *CodeBlock* and *Condition*, with Table 3.1 recording the ports information accordingly. It can be known that time to equilibrium has not been reached as there are available input and output ports of type *Comparison*, *CodeBlock* and *Condition* without circular ports of the same type.

Figure 3.9 shows an intermediate stage where the wrappers of type *Comparison* and *Condition* have self-assembled. The input and output port of type *Condition* are no

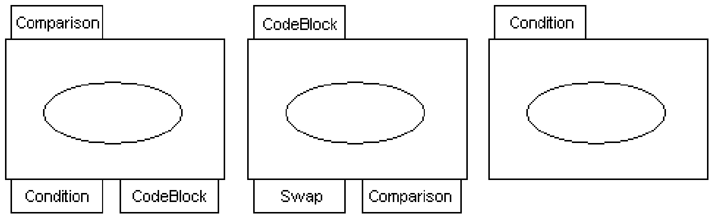


FIGURE 3.8: Illustrative example to show how equilibrium is reached: Step 1 with three intial wrappers in the compartment

TABLE 3.1: An example data structure to observe whether the system has reached its equilibrium at step 1

	Available Output Ports	Available Input Ports	Circular Ports
Comparison	1	1	0
Swap	0	1	0
CodeBlock	1	1	0
Condition	1	1	0

longer available as a result. The corresponding ports information is updated in Table 3.2. It can be known that equilibrium has not been reached because there exists available input and output port of type *Comparison* and *CodeBlock* without circular ports of the same type.

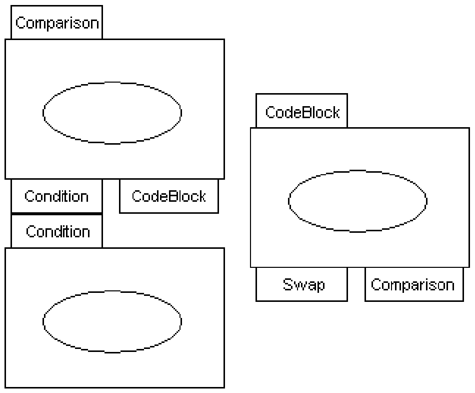


FIGURE 3.9: Illustrative example to show how equilibrium is reached: Step 2 with the wrappers of type Comparison and Condition self-assembled

Figure 3.10 shows the self-assembled product while the two wrappers left in the

TABLE 3.2: An example data structure to observe whether the system has reached its equilibrium at step 2

	Available Output Ports	Available Input Ports	Circular Ports
Comparison	1	1	0
Swap	0	1	0
CodeBlock	1	1	0
Condition	0	0	0

compartment bind with each other via the port of type *CodeBlock*. According to the ports information updated in Table 3.3, the equilibrium has been reached with the calculation results shown below:

$$\begin{aligned} Equilibrium &= (input\_port(Comparison) - circular\_port(Comparison) = 1 - 1 == 0) \\ &\quad \wedge (output\_port(Swap) == 0) \\ &\quad \wedge (output\_port(CodeBlock) == 0) \\ &\quad \wedge (output\_port(Condition) == 0) \\ &= true \end{aligned}$$

TABLE 3.3: An example data structure to observe whether the system has reached its equilibrium at step 3

	Available Output Ports	Available Input Ports	Circular Ports
Comparison	1	1	1
Swap	0	1	0
CodeBlock	0	0	0
Condition	0	0	0

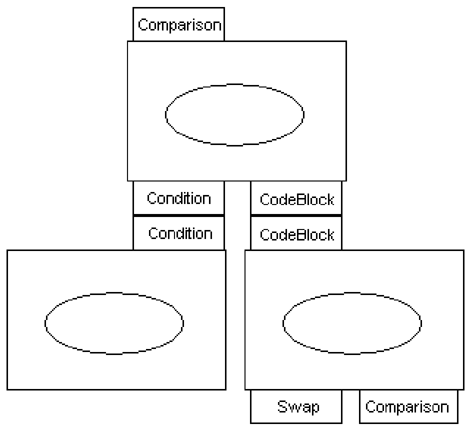


FIGURE 3.10: Illustrative example to show how equilibrium is reached: Step 3 with all the three initial wrappers self-assembled together

3.3 Methods

We aim to find out what is the relation between temperature (T), area (A), number of copies of software components in the compartment (N), pressure (P), time to equilibrium (t_ε) and diversity of the self-assembled trees at equilibrium (D_ε). That is we would like to assess to what degree Eq. 3.1 holds in $ASAP^2$. More specifically, P is measured by counting the number of wrappers that hit one of the four walls of the compartment per unit of time. As mentioned above, T is a free parameter and it affects a wrapper's moving probability, $p(M)$, where $p(M) = 1 - e^{-T}$. The area A of the compartment and number of copies of each wrapper, N, are the other two free parameters of the model.

Besides P, we also measure "time to equilibrium" (t_ε), which records how long it takes the system to reach equilibrium, and the total number of different parse tree classes (D_ε) at equilibrium. D_ε is used to assess the diversity of the emergent parse trees under different settings of N, A and T. For measuring diversity, two trees are considered to belong to the same parse tree class if both their *structures* and *content* are identical, following the recommendation in [11, 22]. Pressure (P), time to equilibrium (t_ε) and diversity of the self-assembled trees at equilibrium (D_ε) are the three observed variables for the experiments.

Three standard sorting algorithms, namely, bubble sort, insertion sort and selection sort are used as sources of software components for the experiments. Each of these, in turn, were cut in an *ad hoc* manner into software components and placed into a component repository of their own. For each of the three software component repositories we executed 20 replicas for each (A, T, N) triplet studied measuring the dependent variables. The ranges for A, T and N were: $A \in \{160k, 250k, 360k, 490k\}$ arbitrary square units, $T \in \{0.25, \ldots, 4\}$ arbitrary temperature units in 0.25 increments and $N \in \{1, 2, 3, 4, 8, 16, 24, 32\}$ copies of each components retrieved from the repository. Table 3.4 shows the total number and classes of wrappers in each repository.

TABLE 3.4: Number of different types of components from insertion sort, bubble sort and selections sort

	Bubble Sort	Insertion Sort	Selection Sort
CodeBlock	5	4	5
Iteration	2	2	2
Comparison	1	0	1
Swap	1	1	1
Condition	1	1	1
Assignment	6	8	9
LoopAssignment	2	1	2
Total Number of Components	18	17	21

3.4 Results

3.4.1 Bubble Sort

The software components used in this experiment arise from the bubble sort related repository. Figure 3.11 shows the relationship between pressure and temperature, and in this experiment, the area of the compartment is fixed to $360k$ (600*600) units. As can be seen from Figure 3.11, regardless of the number of copies of components that are present in the compartment, the pressure on the walls of the compartment increases as the temperature rises. This is due to the increase in $p(M)$ that in turn affects the kinetic energy of a wrapper increasing the likelihood a component hitting one of the compartment's borders. Eq. 3.1 suggests a linear relation between P and T, however, Figure 3.11 illustrates that the rate of increase of pressure declines as temperature rises. This is because programme gas differs from an ideal gas as has been previously explained.

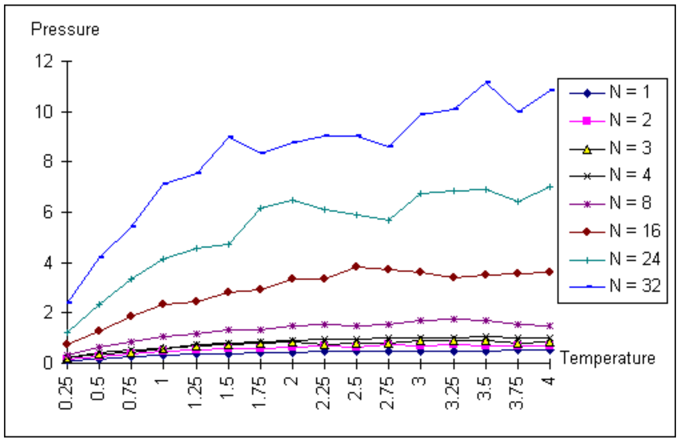


FIGURE 3.11: Relationship between pressure and temperature with different number of copies of components from bubble sort program

Figure 3.12 illustrates the relation between time to equilibrium and temperature using software components from this repository. A shorter time to equilibrium is more desir-

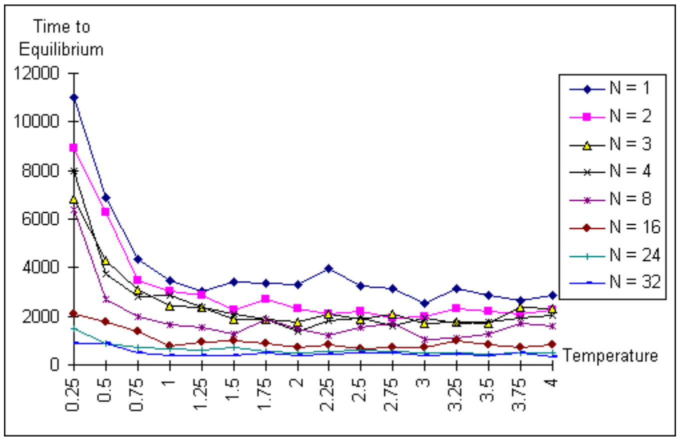
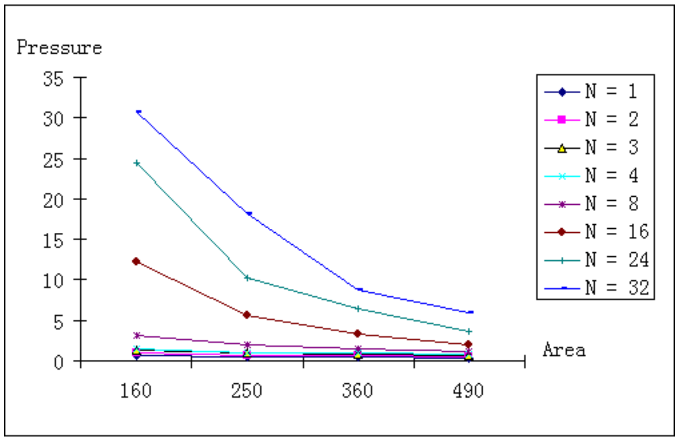


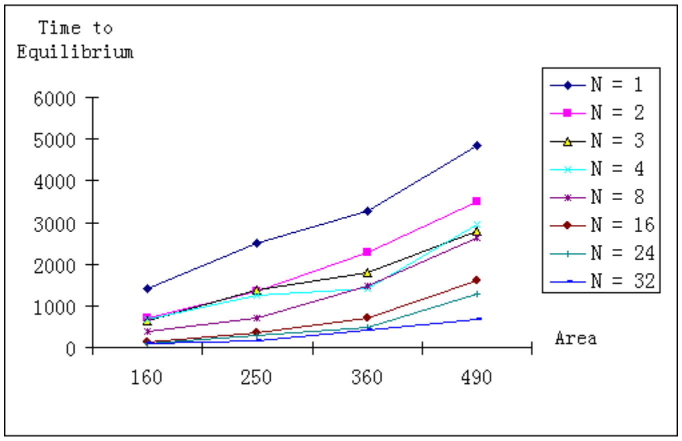
FIGURE 3.12: Relationship between time to equilibrium and temperature with different number of copies of components from bubble sort program

able for fast self-assembly. As can be seen from Figure 3.12, the average time to equilibrium decreases as temperature increases. When temperature rises, components are more likely to move and hence collide and possibly bind with each other. The increase in pressure and decrease in time to equilibrium is dramatic from temperature 0.25 to temperature 1.0. This is because move probability of each component rises significantly while temperature rises from 0.25 to 1.0. It can also be seen from Figure 3.12 that with an increase in the number of components, the time required for the system to reach its equilibrium, t_ε decreases as it is more likely for a component to meet with another component when the pool is densely populated. Moreover, during the course of the simulations it is possible to observe a "run-away" effect: as the size of the (partially) self-assembled structure increases the distance cut-off for binding also increases, which in turn makes it even more likely to collide (and eventually assemble) with other components.

The pressure in the environment will decrease when the area of the container increases, roughly according to Eq. (3.1). Figure 3.13(a) is the experimental result that



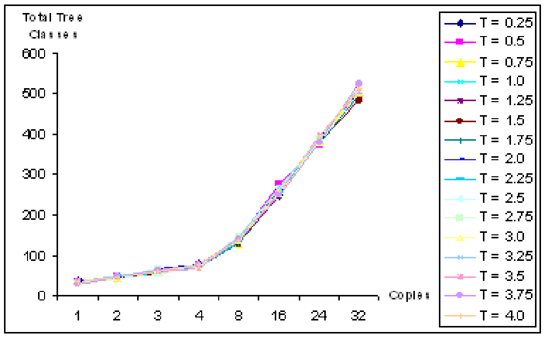
(a)



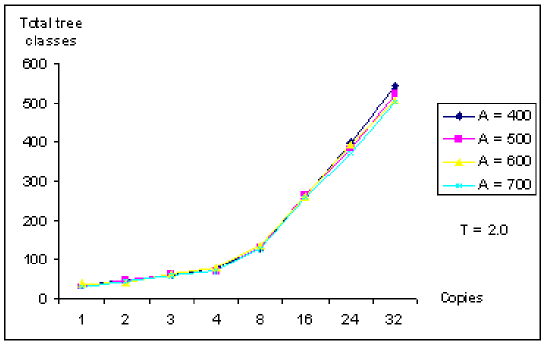
(b)

FIGURE 3.13: (a) Relationship between pressure and area with different number of copies of components extracted from bubble sort program. (b) Relationship between time to equilibrium and area with different number of copies of components extracted from bubble sort program.

illustrates the relation between area of the pool and pressure of the environment. In this experiment, the temperature is fixed to 2.0. Figure 3.13(a) shows that pressure decreases as area increases. Figure 3.13(b) shows how average t_ε is affected by area. As the area of the pool increases, the average time to equilibrium increases as well. The reason is rather straightforward, the pool becomes larger and the probability for a component to bind with another component decreases.



(a)



(b)

FIGURE 3.14: Using components extracted from bubble sort program: (a) Relationship between total different tree classes and number of copies ($A = 360k$); (b) Relationship between total number of different tree classes and number of copies ($T = 2.0$).

Figure 3.14(a) shows the relation between the total number of different parse tree classes (D_ε) in 20 replicas and the number of copies of components in different temperature settings for an area fixed to $360k$ (600*600) arbitrary units of area. Fig. 3.14(b) shows the relation between D_ε and N for various area settings and temperature fixed to 2.0 arbitrary temperature units. Generally, it can be seen that the diversity of assembled parse tree classes rises as more components are put into the pool. However, Fig. 3.14(a) and 3.14(b) show that the diversity of the self-assembled trees is not greatly affected by either temperature or pool area. We show a "forest" of self-assembled program trees when the temperature is set to 2.0, and the area is set to $360k$ units with 8 copies of each component decomposed from the original bubble sort program in Fig. 3.15.

3.4.2 Insertion Sort

A similar set of experiments has also been carried out using components obtained from the insertion sort repository. Figure 3.16 shows the relation between pressure and temperature. Similar to the case of bubble sort, pressure rises steadily as more copies of components are placed into the pool. Figure 3.17 shows that time to equilibrium decreases as temperature increases. Figure 3.18(a) shows that pressure decreases as area of the pool increases. The decrease in pressure becomes steadier with fewer number of copies of components. Figure 3.18(b) illustrates that the average equilibrium time increases as the area of the pool becomes greater. Figure 3.19(a) illustrates total number of different parse tree classes in the 20 replicas against number of copies of components for different temperature setting with a fixed $360k$ arbitrary area units. The results are similar to the ones obtained for the bubble sort repository, that is, diversity is mainly affected by the number of copies of components placed into the pool. Figure 3.19(b) shows the relation between the number of copies of components and diversity using different areas when temperature is set to 2.0 arbitrary temperature units.

3.4.3 Selection Sort

Same experiments are performed with the selection sort repository. Figure 3.20 shows how pressure is affected by temperature and the number of copies of each component. Similar to the results shown in the previous experiments on bubble sort and insertion sort, given a fixed number of components, pressure rises as temperature rises. Moreover, given a fixed

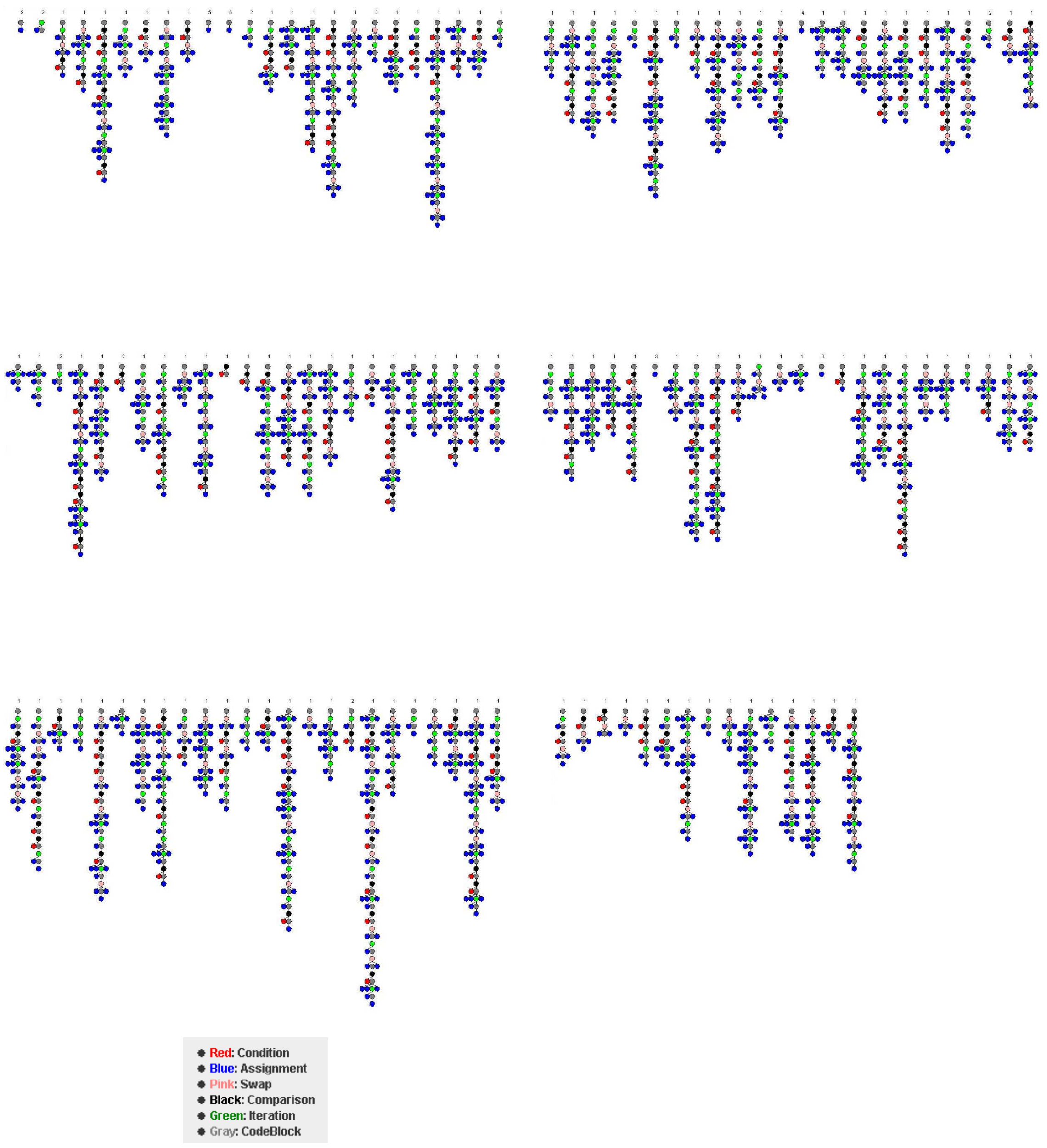


FIGURE 3.15: A "forest" of self-assembled parse trees using components from bubble sort repository, and under the environment settings $A = 360k, T = 2.0, N = 8$. The number on top of each tree indicates the number of occurrences of that tree in the 20 experimental replicas.

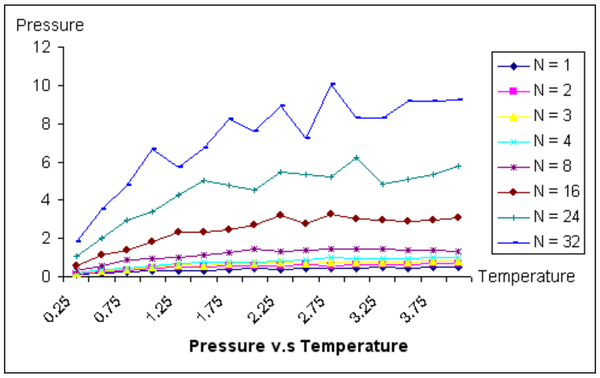


FIGURE 3.16: Relationship between pressure and temperature with different number of copies of components from insertion sort program

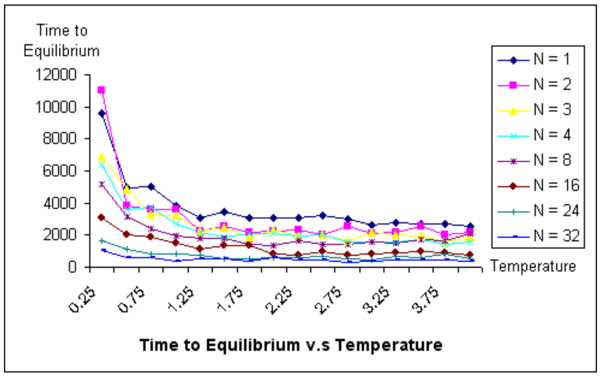
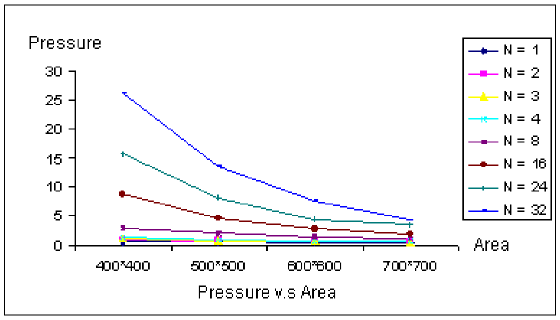
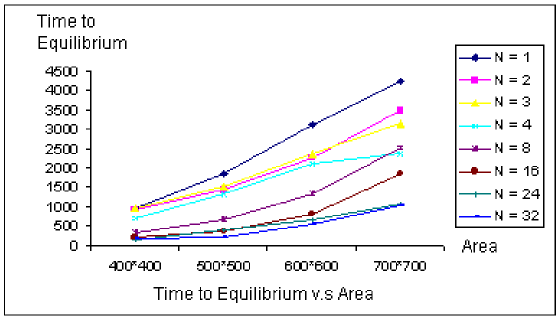


FIGURE 3.17: Relationship between time to equilibrium and temperature with different number of copies of components from insertion sort program

temperature, pressure rises as more components are placed into the pool. However, with the same number of copies and same temperature, pressure in the pool is generally higher than pressure in the pool for insertion sort and bubble sort. This difference becomes more obvious when more copies of components are placed into the pool as is shown in Figure 3.20. Figure 3.21 shows the relation between time to equilibrium and temperature for insertion sort. Generally, the average time required to reach equilibrium decreases as a result of an increase in temperature. Figures 3.22(a) and 3.22(b) show the relation between pressure and area; and time to equilibrium and area respectively. It can be seen that as area increases,



(a)

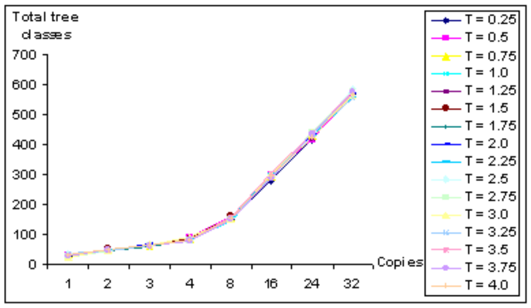


(b)

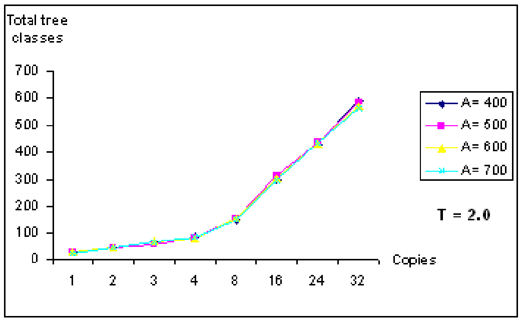
FIGURE 3.18: (a) Relationship between pressure and area with different number of copies of components extracted from insertion sort program. (b) Relationship between time to equilibrium and area with different number of copies of components extracted from insertion sort program

time to equilibrium increases and pressure decreases. However, with the number of copies of components and area of the pool fixed, pressure in the pool for selection sort is slightly higher than that of insertion sort and bubble sort, and this is also because of the larger size of the repository.

Figure 3.23(a) shows that with more copies of components, the total number of different parse tree classes rise when area is set to $360k$ arbitrary area units. Figure 3.23(b) illustrates the relation between diversity of the formed parse trees and the number of copies of components with temperature fixed to 2.0. Similar to the results obtained from bubble sort and insertion sort, diversity is mainly influenced by the number of copies of components.



(a)



(b)

FIGURE 3.19: Using components extracted from insertion sort program: (a) Relationship between the total number of different tree classes and number of copies ($A = 360k$); (b) Relationship between total number different tree classes and the number of copies ($T = 2.0$).

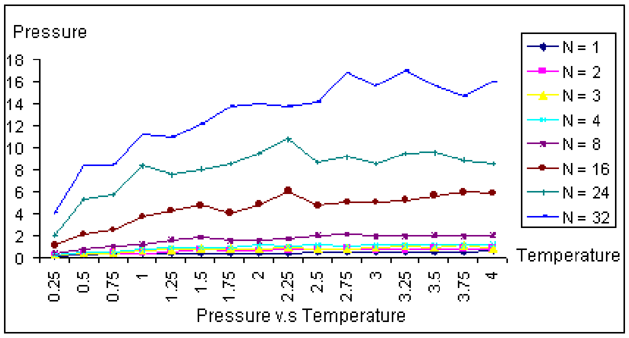


FIGURE 3.20: Relationship between pressure and temperature with different number of copies of components decomposed from selection sort program

Tables 3.5, 3.6, and 3.7 illustrate the R values computed with different environment parameters for bubble sort, insertion sort and selection sort program respectively, i.e. $R = PV/nT$. As has been discussed before in the previous section, programme gas differs from

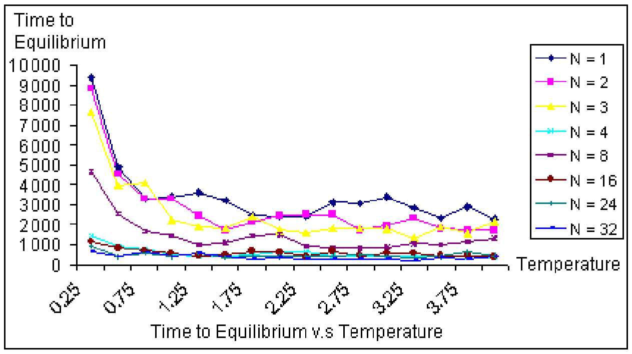
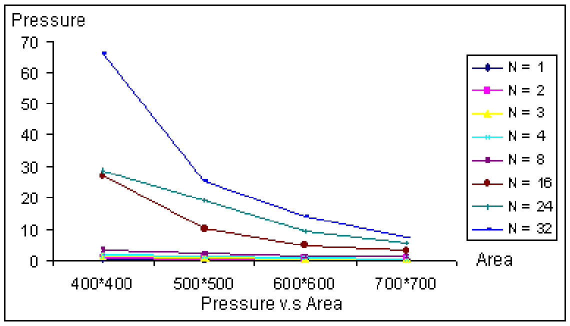
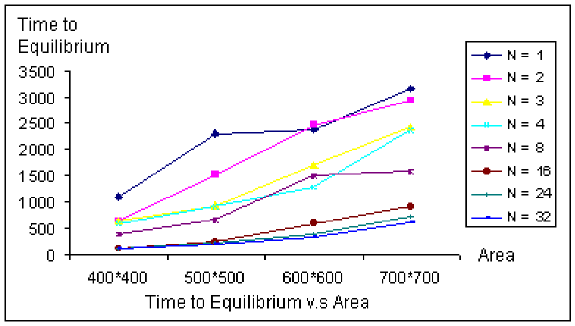


FIGURE 3.21: Relationship between time to equilibrium and temperature with different number of copies of components decomposed from selection sort program

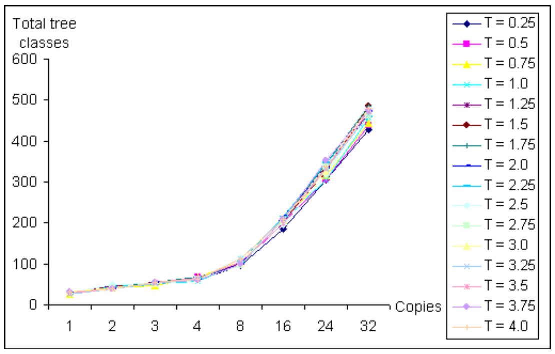


(a)

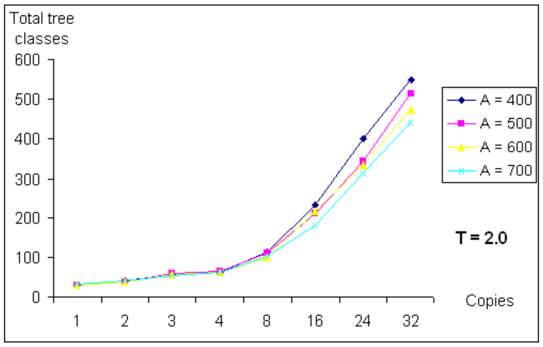


(b)

FIGURE 3.22: (a) Relationship between pressure and area with different number of copies of components decomposed from selection sort program. (b) Relationship between time to equilibrium and area with different number of copies of components decomposed from selection sort program.



(a)



(b)

FIGURE 3.23: Using components decomposed from selection sort program: (a) Relationship between total different tree classes and number of copies ($A = 360k$). (b) Relationship between total number of different tree classes and number of copies ($T = 2.0$).

an ideal gas. In an ideal gas, the R is a constant value. However, it can be seen from Tables 3.5, 3.6, and 3.7 that R values for $ASAP^2$ do not stay constant as opposed to ideal gases. In order to measure how statistically different the R values attained from the three programs are, ANOVA analysis is performed under the same (T, V, n). The computed p-value from the ANOVA test is the probability that the variation between groups may have occurred by chance. The null hypothesis is used to test differences among groups, and the assumption that no difference exists between groups for the variable being compared. By convention, the null hypothesis is rejected if the p-value is smaller than or equal to the significance level 0.05 [13]. Hence, a high p-value therefore indicates the difference between groups

TABLE 3.5: R value computed from different environment parameters for bubble sort program

T	V	n	P	R
0.25	160	1	0.176649802	113.0558736
0.5	160	1	0.311645187	99.72645977
0.75	160	1	0.392366822	83.70492208
1	160	1	0.477072752	76.33164027
1.25	160	1	0.509688654	65.24014767
1.5	160	1	0.613259763	65.41437476
1.75	160	1	0.666353766	60.92377291
2	160	1	0.598151136	47.85209085
2.25	160	1	0.594808557	42.29749742
2.5	160	1	0.620507593	39.71248598
2.75	160	1	0.772877163	44.96739856
3	160	1	0.740508553	39.4937895
3.25	160	1	0.673260397	33.14512722
3.5	160	1	0.706946771	32.31756666
3.75	160	1	0.702513062	29.97389064
4	160	1	0.695453068	27.8181227

TABLE 3.6: R value computed from different environment parameters for insertion sort program

T	V	n	P	R
0.25	160	1	0.159352116	101.9853542
0.5	160	1	0.530052968	169.6169497
0.75	160	1	0.382439856	81.58716922
1	160	1	0.492226003	78.75616051
1.25	160	1	0.49292456	63.09434368
1.5	160	1	0.523136365	55.8012123
1.75	160	1	0.573825845	52.46407729
2	160	1	0.594619024	47.56952194
2.25	160	1	0.626837524	44.57511283
2.5	160	1	0.698775841	44.72165382
2.75	160	1	0.669286865	38.94032669
3	160	1	0.668869555	35.67304293
3.25	160	1	0.648874518	31.94459165
3.5	160	1	0.647836647	29.61538956
3.75	160	1	0.692331722	29.53948682
4	160	1	0.753522834	30.14091334

TABLE 3.7: R value computed from different environment parameters for insertion sort program

T	V	n	P	R
0.25	160	1	0.18346635	117.4184642
0.5	160	1	0.358986948	114.8758232
0.75	160	1	0.413102263	88.1284828
1	160	1	0.499091046	79.85456739
1.25	160	1	0.556439759	71.22428914
1.5	160	1	0.638777477	68.13626421
1.75	160	1	0.654263144	59.81834456
2	160	1	0.691053549	55.2842839
2.25	160	1	0.696325818	49.51650264
2.5	160	1	0.701287535	44.88240226
2.75	160	1	0.817719929	47.57643222
3	160	1	0.77585109	41.37872481
3.25	160	1	0.762574606	37.54213444
3.5	160	1	0.795425819	36.36232316
3.75	160	1	0.780521731	33.30226051
4	160	1	0.812693773	32.50775094

are insignificant. The p-value computed from the ANOVA test for the three groups of R values is 0.90577, which is substantially greater than the acceptance level. This indicates the null hypothesis cannot be rejected, meaning little or no difference exists between the distributions for R as computed from bubble sort, insertion sort and selection sort program. In addition, with the experimental results on the three sorting algorithms, it can be seen that the environment parameters (A, T, N) affect time to equilibrium and diversity of generated programs in a very similar pattern for all the three sorting algorithms.

3.5 Predictive Formulae

We introduced a mathematical model that can be used to predict both t_ε and D_ε for the standard $ASAP^2$ implementation as a function of (A, N, T). This was summarized in [58] with a regression software. Formulae are produced to predict t_ε having one of the three free environment parameters fixed to a pre-specified value ($T = 2.0, N = 8, A = 360$ respectively). These predictive models are presented in Eq. 3.2, 3.3 and 3.4. Figure 3.24 depicts how accurately Eq. 3.2, 3.3 and 3.4 predict t_ε in comparison with the average of the 20 replicas.

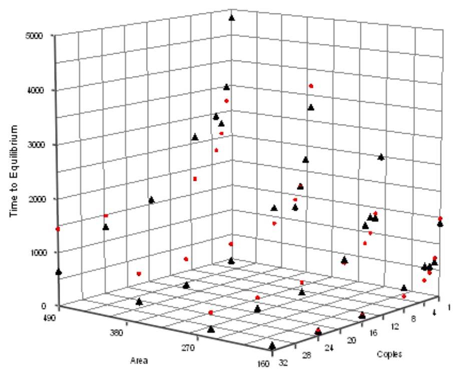
$$t_\varepsilon(A,N) = \frac{(7.05 \times A) + 321.2}{N} + (3.91 \times A) - 593.71 \tag{3.2}$$

$$t_\varepsilon(A,T) = \frac{(5.21672 \times A) - 659.015}{T} + (3.7274 \times A) - 416.114 \tag{3.3}$$

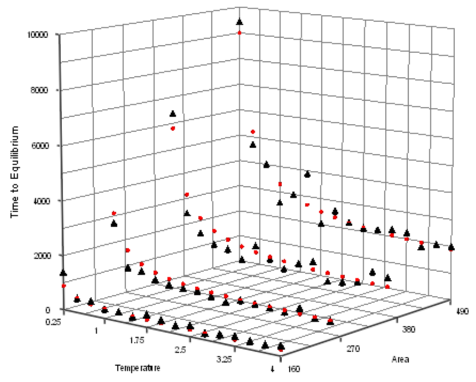
$$t_\varepsilon(T,N) = (\frac{1}{T} + 1) \times (\frac{1726}{N} + 637.325) \tag{3.4}$$

Figure 3.24 shows that the predictions are fairly accurate as experimental data follow the same trend as predicted by the formulae although errors exist between predictions and experimental results. The errors are quantified by calculating the average and standard deviation of errors rate for each formula (Table 3.8). Table 3.8 shows average and standard deviation of errors rate for each equation in different parameter settings. With a lower area setting in Eq. 3.2 and Eq. 3.3, the equations predict more accurately having a smaller average error rate. In additions, Eq. 3.2 and 3.3 predict better than Eq. 3.4 because they report a smaller average error rate.

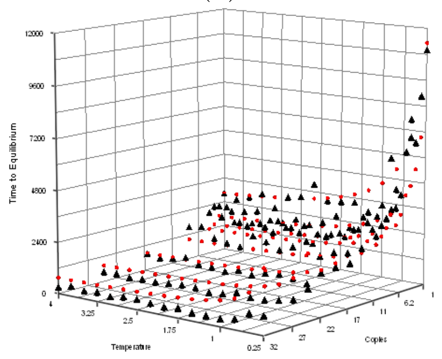
$$D_\varepsilon(N) = 15 * N + 16.6 \tag{3.5}$$



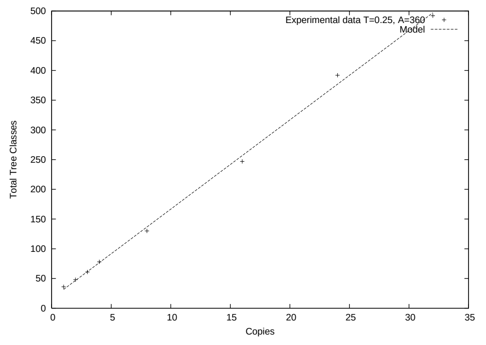
(a)



(b)



(c)



(d)

FIGURE 3.24: Prediction model assessment on (a) time to equilibrium with T=2.0 (b) time to equilibrium with N=8 (c) time to equilibrium with $A = 360k$ (d) diversity of the generated programs with $A = 360k, T = 0.25$. Triangles represent experimental data, and circles represent the corresponding data obtained from our predictive model.

TABLE 3.8: Error statistics for predictive model on $ASAP^2$

equation	category	average error rate	standard deviation
Eq. 3.2	$A = 160$	0.208	0.148
	$A = 250$	0.425	0.525
	$A = 360$	0.389	0.473
	$A = 490$	0.261	0.374
Eq. 3.3	A=160	0.242	0.145
	A=250	0.152	0.146
	A=360	0.192	0.098
	A=490	0.097	0.090
Eq. 3.4	$N = 1$	0.106	0.106
	$N = 2$	0.129	0.089
	$N = 3$	0.181	0.108
	$N = 4$	0.263	0.095
	$N = 8$	0.224	0.144
	$N = 16$	0.219	0.189
	$N = 24$	0.733	0.207
	$N = 32$	1.190	0.521
Eq. 3.5	N/A	0.039	0.010

As has been shown in Fig. 3.14, diversity of self-assembled programs is not greatly affected by temperature nor by area of the pool. Hence, we present Eq. 3.5 to predict program diversity in relation to number of copies of components only. Figure 3.24(d) illustrates predicted and the observed experimental results, and the obtained experimental data agree quite well with the model predictions. This can also be seen from Table 3.8, which shows a small average error rate. This interpolation for (t_ε, D_ε) can already be seen as an advance since similar predictive performance is yet to be achieved with GP.

3.6 Summary

In this chapter, we presented the Automated Self-Assembly Programming Paradigm ($ASAP^2$). The initial experiments are reported with a naive implementation of "program gases". Program gases is a metaphor for a software self-assembly system that takes a set of software components from a given repository and places them into a finite area container where they perform a random walk and sometimes they will bind to each other. While Genetic Programming evolves both the structure and the content of the parse trees simultaneously, $ASAP^2$ is aimed at *reusing the content* of previously built software components by self-assembling a suitable *structure*. A program gas component can be as simple as an assignment statement or arbitrarily large, such as a program that takes a user input and produces some complex output. Hence, software self-assembly concentrates more on exploring and discovering the architecture of programs. The kinetic theory of perfect gases was used as an approximation of the behaviour of the system. The R values computed for program gas suggest that it is not constant, as opposed to ideal gases, and hence prove that program gas behaves in a similar yet different way to ideal gases. We measured the relation between different settings of the environment, i.e. A, T, N, with pressure, time to equilibrium and diversity of assembled structures when equilibrium is reached thus assessing how well this particular implementation of $ASAP^2$ explores the space of all program architectures. Unlike genetic programming that uses an evolutionary metaphor to guide the process of automatic programming, $ASAP^2$ relies on natural metaphors of self-assembly for the exploration of programs space, thus we expect to have very different kinds of limitations and potentials.

In addition, we have presented a predictive model to interpolate software self-assembly efficiency and diversity for a given set of environment parameters (A, T, N). We can predict the time needed for $ASAP^2$ to reach its equilibrium and the resulting diversity of the generated populations, given the available number of copies of components and other environmental conditions. In contrast, there is no GP theory that can predict diversity or time to local optima.

CHAPTER 4

A Particle Swarm Realization of Software Self-Assembly

In the previous chapter, we introduced the Automated Self-Assembly Programming Paradigm ($ASAP^2$), a software self-assembly system inspired by natural and artificial self-assembly systems. We presented $ASAP^2$ using the kinetic theory of ideal gases as a metaphor. In $ASAP^2$, a set of human made components are collected in a software repository and later integrated through self-assembly into a software program. Manually decomposed software components are placed in a confined area with specific temperature and pressure constraints, giving rise to a variety of program architectures. We investigated and discovered different factors that influence the efficiency of software self-assembly and the diversity of the generated programs using *unguided* self-assembly.

Self-assembly systems can be guided using various methodologies. Wang tiles [92] is a prime example of computational self-assembly. A Wang tile system consists of square tiles maneuvering on a two dimensional plane. Each side of the tile has a colour associated with it. A table is used to record the binding strength between every pair of colours. As tiles perform Brownian motion in the plane, they may either self-assemble or continue their random movement depending on the binding strength on their colliding side. In [88], an evolutionary algorithm is used to evolve a family of tiles together with the colour associated with them so that they self-assemble into a target shape.

In [43], a different approach is used to guide the self-assembly process. Simple and identical components autonomously grow into a desired shape on a two-dimensional square lattice by connecting of components around a single initial seed component. The successful assembly lies in the generation of appropriate rules that are executed by the components. In turn, the set of rules for the components to follow is compiled based on the given goal structure. As can be seen, various approaches can be used to construct a self-assembly system. However, the key to the successful assembly of the goal structure lies in the careful design of components and their interactions.

As $ASAP^2$ is inspired by those natural and artificial self-assembly systems, software self-assembly process can also be guided using various metaphors. This chapter introduces a different metaphor to guide the self-assembly process for $ASAP^2$. The extended model is based on particle swarm intelligence, and the concepts of leaders and dynamic neighbourhood is introduced into $ASAP^2$. The extended model is inspired by previous works in PSO [14, 38, 75]. The swarm based approach changes the way software components interact with each other, and we are thus interested in investigating how this affects time to equilibrium, while at the same time observing how diversity of the generated programs is affected. With the extended system, we performed the same set of experiments as presented in the previous chapter [59].

This chapter is organized as follows. In section 1, we briefly introduce artificial life and particle swarm optimization (PSO) in particular. In section 2, we present our software self-assembly system inspired by PSO. Section 3 illustrates experimental results and analysis, and the comparison between the two different approaches for $ASAP^2$. Finally, Section 4 gives a summary of this chapter.

4.1 Particle Swarm Optimization

There is a vast multitude of computational techniques inspired by natural living systems. For example, genetic algorithm is inspired by the evolutionary process of living species. Neural network simulates the central nervous systems and the information processing of neurons. Artificial life [8] is the term used to describe research and man-made systems that mimic the behaviour of living entities and hence capture certain properties of life. Besides the purpose of enriching the knowledge about nature, artificial life helps to find new computational techniques such as swarm intelligence. The artificial life models we describe here are restricted to social learning systems. More specifically, those systems that exhibit high level of emergent behaviour as a result of the interaction between individuals of the population and the environment.

Social learning systems tend to concentrate on how complex natural organisms achieve sophisticated group behaviours, or deal with abstract agents that are difficult to be manually constructed [34]. Swarm intelligence is one of the metaphors that mostly attract interest of the social learning modelling community. Swarm intelligence refers to phenomena where natural creatures behave as a swarm with individuals following simple rules, but leading to complex, adaptive behaviour of the swarm [21]. There are two main variants of swarm inspired methods within the social learning areas: one is to mimic the flocking of birds (particle swarm optimization), and the other is inspired by the co-operative behaviour of ant colonies (ant colony optimization) [17]. In this chapter, we concentrate on the former, as a metaphor to guide software self-assembly.

Particle swarm optimization (PSO) was first introduced by Kennedy and Eberhart [19, 20]. PSO mimics the behaviour of bird flocking and fish schooling. Individuals in a swarm exchange previous experiences whilst the randomness of moving in the embedding space is maintained. In PSO, each individual in the population is called a particle which represents a solution to a given problem. Particles are modelled with their positions and velocities. At each iteration of the algorithm, each particle's velocity is stochastically accelerated towards its previous best position and towards a global best position among the population. Particles then adjust their velocities according to their personal best position and the global best position. The fundamental concept behind PSO algorithm is that individuals exchange previous experience whilst maintaining random movement in the search space.

Let $X[i][j]$ be the current location of particle i in dimension j, $P[i][j]$ be the current best location for particle i, V_i be the velocity of particle i and G be the global best location. The following shows a pseudo code for the PSO algorithm, where Γ_1, Γ_2 are uniform random numbers between 0 and 1. C_1, C_2 and ω are pre-specified parameters for the algorithm. ω describes how much particles' previous velocity affects their current velocity. C_1 and C_2 are constants that reflect how much particles' velocities are influenced by the individual's best and the global best respectively. “Best” in this context is a problem / domain specific measure of preference.

Algorithm 3 PSO algorithm

```
1: Initialise the population of particles with $X$ and $V$
2: At each time step $k$
3: for each particle $i$ do
4:   for each dimension $j$ do
5:     $V[i][j] = \omega \times V[i][j] + C_1 \times \Gamma_1 \times (P[i][j] - X[i][j]) + C_2 \times \Gamma_2 \times (G[j] - X[i][j])$
6:     $X[i][j] = X[i][j] + V[i][j]$
7:   end for
8: end for
```

In [14, 75], fitter particles are recorded into a leader set. Non-leader particles randomly select a leader from the leader set and follows the leader. In [38], particles determine and change their neighbours dynamically according to the distances in each generation. Those previous works on PSO have shown encouraging outcomes on fitness and diversity of the generated results. Recent studies [70, 71] by O’Neil and Brabazon have also demonstrated that it is possible to generate programs by using *grammatical swarm*, a PSO inspired automatic programming methodology which uses grammars to guide the construction of programs for specific task. Here we borrow the idea of using PSO to synthesize programs and we hybridize it with our $ASAP^2$ concept.

4.2 $ASAP^2$ with Dynamic Leaders and Neighbourhood

4.2.1 Model description

Inspired by PSO algorithm and grammatical swarm, we extended the $ASAP^2$ model presented in [59] by introducing *leaders* in the software self-assembly process in an attempt to shorten time to equilibrium. In this extended system [58], neighbourhood structures and movements of software components are influenced by leader components in a similar pattern as in [14, 75]. Figures 4.1 and 4.2 are screenshots of the system. The left panels show the compartment in which software self-assembly takes place. The difference from the original model is that leader components are introduced in the pool, and they can be identified by circles or tree structures with white borders in Fig. 4.1 and 4.2.

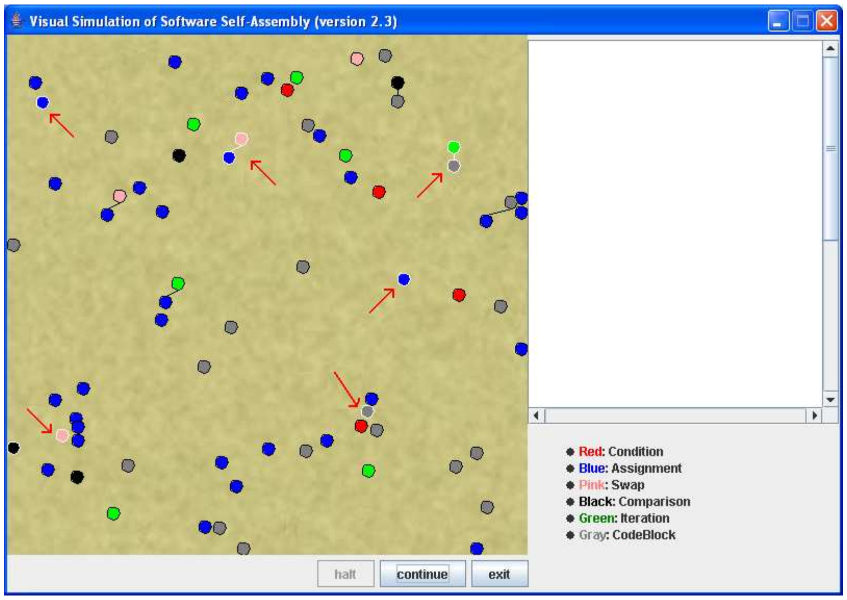
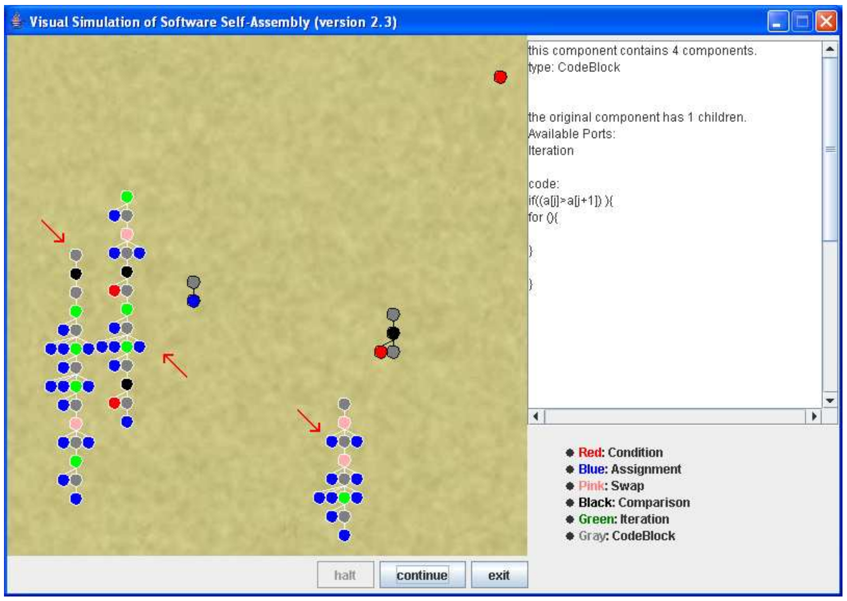


FIGURE 4.1: Early stage of software self-assembly, where circles with white borders are leader components.

Algorithm 4 describes the implementation of the PSO inspired $ASAP^2$ system in pseudo-code. As in the unguided $ASAP^2$ model, software components are retrieved from a software repository and placed into the compartment. An initial proportion of software components are then randomly selected as leaders. Software components perform random walk in the compartment as they do in [59]. However, when a non-leading component is within a distance threshold (D_α) to a leader, it will follow the leader component. If



```
this component contains 4 components.
type: CodeBlock


the original component has 1 children.
Available Ports:
Iteration

code:
if((a[j]>a[j+1]) ){
for (){

}

}
```

- ● Red: Condition
- ● Blue: Assignment
- ● Pink: Swap
- ● Black: Comparison
- ● Green: Iteration
- ● Gray: CodeBlock

FIGURE 4.2: Latter stage of software self-assembly, where circles with white borders are leader components

more than one leader is within the threshold range (D_α) of a non-leading component, there needs to be a way to decide which leader to follow. To do that an attractive force (F) of a leader component is introduced. Non-leader components follow a leader component that exhibits the greatest attractive force. In this way, a non-leader component changes its leader dynamically. The attractive force of a leader component is in direct proportion to the number of available ports it contains and its distance to a following component. Algorithm 5 formally describes this process.

When a certain component binds with a leader component, the self-assembled structure also becomes a leader. In this way, the proportion of leader components in the compartment increases as more components are self-assembled and non-leader components disappear. The motivation behind this is that the self-assembly process could be made more efficient by introducing leaders since components will attract each other and form groups

Algorithm 4 Algorithm for PSO inspired $ASAP^2$

```
1: $C$ = RETRIEVE_ COMPONENTS(number_ of_ copies)
2: $L$ = DETERMINE_ LEADERS($C$)
3: $N$ = non-leading components
4: while (equilibrium not reached) do
5:    for (every component $C_i \in C$) do
6:       if ($C_i \in N$) then
7:          if ( $C_i$.canFindLeader($L, D_\alpha$) ) then
8:             $C_i$.followLeader()
9:          else
10:            $C_i$.move()
11:         end if
12:      else
13:         $C_i$.move()
14:      end if
15:   end for
16:   attemptBinding()
17: end while
```

Algorithm 5 Algorithm to find a leader component within threshold range D_α for a non-leader component N_i

```
1: initialize $current\_leader$
2: initialize $F\_max$
3: for ( every leader component $L_i$ in the component set $C$ ) do
4:    if ( distance ($N_i$, $L_i$) $\leq D_\alpha$ ) then
5:       $F$ = distance($N_i$, $L_i$) $\times$ availablePorts($L_i$)
6:       if ( $F \leq F\_max$ ) then
7:          $F\_max = F$, currentLeader = $L_i$
8:       end if
9:    end if
10: end for
```

rather than getting stuck in distant areas of the pool. Figure 4.1 shows an early stage of software self-assembly process in which a small proportion of leader components are present in the compartment. At a latter stage of self-assembly, Fig. 4.2 shows the proportion of leader components in the pool increases significantly.

The particle swarm based approach for $ASAP^2$ is a simplified version of PSO. Although it mimics PSO algorithm such that components follow leaders and dynamic neighbour will gradually form, it differs with the original PSO algorithm in two main aspects: (1) each particle is "flying" over search space in PSO, whereas software components are moving on a two-dimensional "physical" compartment in $ASAP^2$. (2) Each particle represents a solution in PSO and is evaluated by an objective function. The flying of particles in the problem space is controlled by velocities, which in turn is determined by each individual's experience and the best solution found in the entire population. In swarm based $ASAP^2$, however, the interactions between components are not determined by evaluation function. The interactions between components are determined by leader's attractive force and their relative distance. Components merely follow a leader component which exhibits the greatest attractive force given by ports number (i.e. a leader with more available ports is more attractive) in an attempt to shorten time to equilibrium.

4.3 Experimental Results and Analysis

4.3.1 Methods

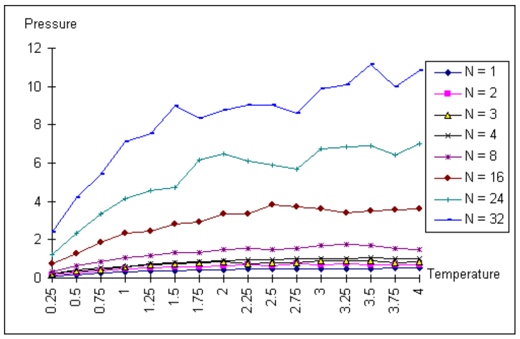
We compared how PSO inspired $ASAP^2$ differs from the unguided approach. Hence, the same set of experiments as we did for the unguided $ASAP^2$ are performed for the swarm based system. The fixed parameters in our experiments are distance threshold (D_α) for attractive force, and proportion of leader components (l_τ) at the start of the simulation. Temperature (T), area (A), number of copies of software components in the compartment (N) are the free parameters for the simulation. l_τ is fixed to 0.1 so that the influence of leaders are unbiased while more components are placed into the compartment. The threshold distance ensures that a component with arbitrarily large attractive force will not attract everything in the compartment. T affects wrappers' moving probability the same way as in unguided $ASAP^2$, i.e. $p(M) = 1 - e^{-T}$.

With the above parameters, the same experiment data as we measured for unguided $ASAP^2$ are observed: pressure (P), time to equilibrium (t_ε) and diversity of the self-assembled trees at equilibrium (D_ε). These observed experiment data are measured in the same way as in [59] for unguided dynamics. That is, P is measured by the number of hits on the borders of compartment caused by wrappers' movements per unit of time. Time to equilibrium (t_ε) measures the number of time units recorded before equilibrium is reached. The diversity of assembled program (D_ε) is measured by the total number of different parse tree classes at equilibrium.

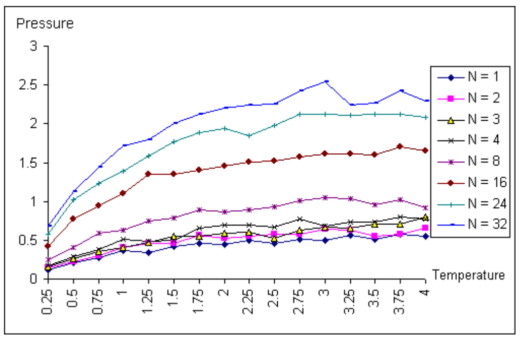
4.3.2 Results and Analysis

Figure 4.3 shows the comparison of how pressure is affected in relation to temperature between unguided $ASAP^2$ and swarm based $ASAP^2$. And Fig. 4.4 illustrates the relationship between pressure and area of the pool. As can be seen, pressure in the compartment is affected in similar patterns as the unguided software self-assembly system. That is, pressure increases with more number of components, higher temperature or smaller area of the pool. In addition, it can be seen (Fig. 4.3(a) v.s Fig. 4.3(b), and Fig.4.4(a) v.s Fig. 4.4(b)) that comparing with unguided software self-assembly, the overall pressure on the wall drops significantly. This is because more components are following leaders instead of moving randomly in the compartment and hitting the walls.

Figures 4.5 and 4.6 show how t_ε is affected by T and A under various N respectively. Fig. 4.5, 4.6 suggest that area of the pool and temperature have a similar influence on time to

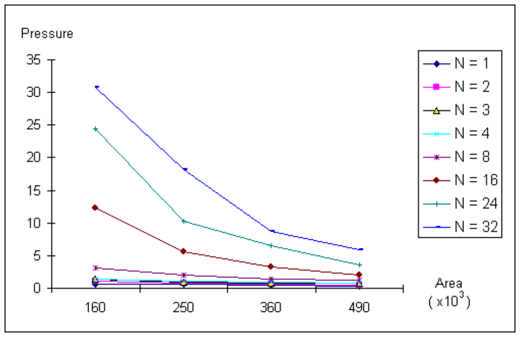


(a)

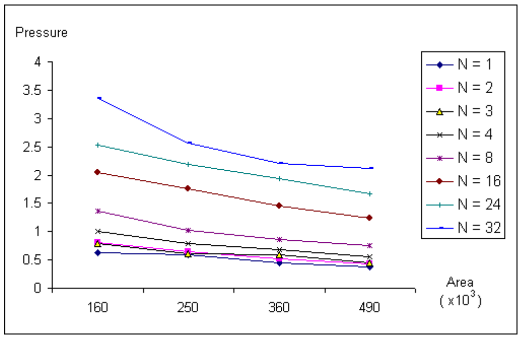


(b)

FIGURE 4.3: Relationship between pressure and temperature with different number of copies of components with (a) unguided $ASAP^2$, (b) swarm based $ASAP^2$



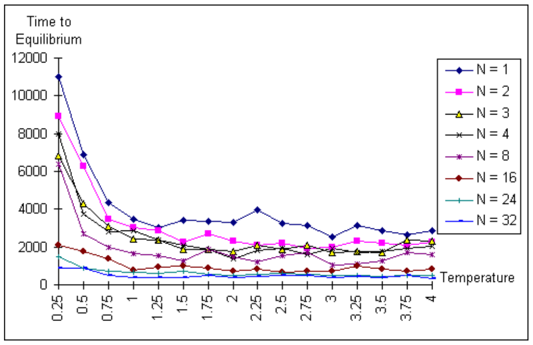
(a)



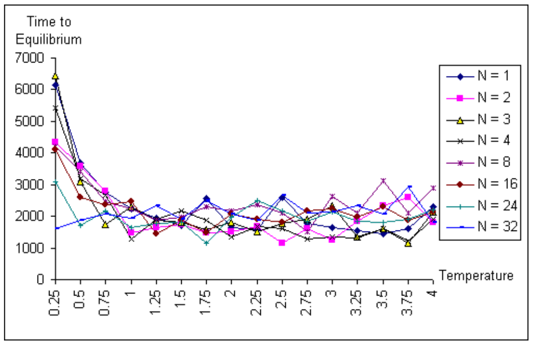
(b)

FIGURE 4.4: Relationship between pressure and area with $T = 2.0$, (a) unguided $ASAP^2$, (b) swarm based $ASAP^2$

equilibrium for unguided and swarm based software self-assembly. Figure 4.5 indicates that temperature only affects time to equilibrium at a low range (from 0.25 to approximately 1.0) such that t_ε decreases as T increases. This is because the moving probability of a component is determined by temperature, and move probability of each component rises significantly while temperature rises from 0.25 to 1.0. Furthermore, Fig. 4.6 shows that it takes longer for the system to reach equilibrium with a larger pool. This is because the average density of components decreases with a larger compartment, and therefore takes longer for components to form bindings.

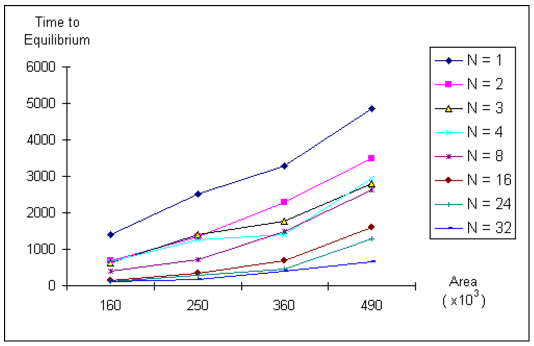


(a)

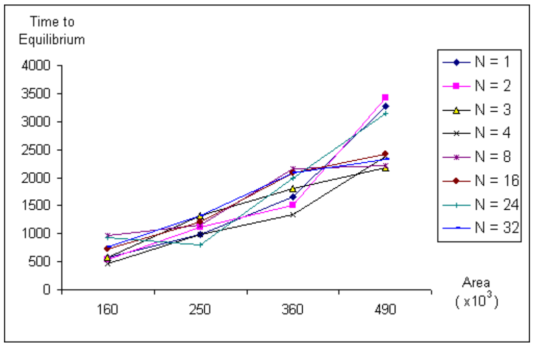


(b)

FIGURE 4.5: Relationship between time to equilibrium and temperature with $A = 360k$, (a) unguided $ASAP^2$, (b) swarm based $ASAP^2$



(a)



(b)

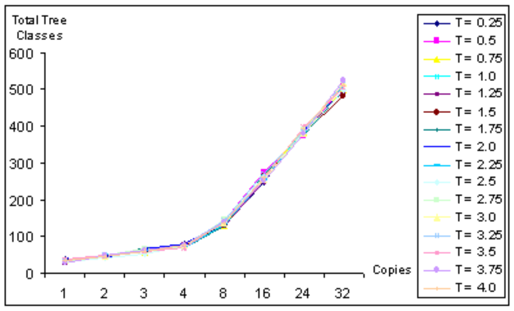
FIGURE 4.6: Relationship between time to equilibrium and area with $T = 2.0$, (a) unguided $ASAP^2$, (b) swarm based $ASAP^2$

Comparing swarm based $ASAP^2$ with unguided $ASAP^2$, time to equilibrium is influenced by N in an utterly different way. It can be seen from Fig. 4.5(b), 4.6(b) that N does not play an important role in t_ε for swarm based $ASAP^2$. As the proportion of leader selected at the beginning of the simulation is a fixed parameter, an increase in N results in an even increase in the number of leader components as well as non-leader components. Hence, leader components are attracting the same amount of non-leader components as N increases. Experimental results (Fig. 4.5(a) v.s Fig. 4.5(b), and Fig.4.6(a) v.s Fig. 4.6(b)) show that time to equilibrium has been significantly reduced from a maximum of 11000 to a maximum of 6500 when leaders are introduced into the system under $N \in \{1, 2, 3, 4\}$. However, unguided $ASAP^2$ reaches time to equilibrium faster than swarm based $ASAP^2$ with a large number of components, i.e. $N \in \{24, 32\}$.

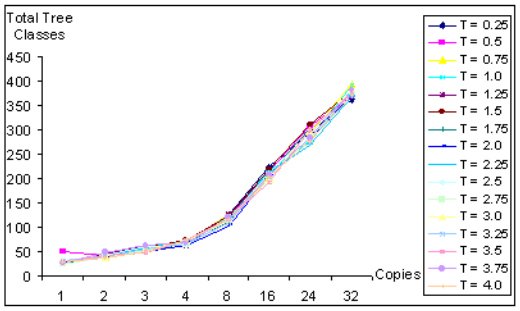
The reason for this is the following: under unguided $ASAP^2$ the more components there are in the compartment, the higher the number of collisions between components. Moreover this collision happens without a preferred position or direction in compartment (i.e, the space is isotropic in what pertains to direction of movements). On the other hand, in the PSO $ASAP^2$ the presence of a large number of leaders, which incidentally change with time, severely distorts the space isotropy and introduces non-linear feedbacks into the dynamics of the simulation. For example, one way in which the above consideration could affect t_ε is this: if there are 2 or more leaders and a non-leader component C, it is possible for C to follow leader L_1 for a long time, because L_1 has more ports available than L_i (with $i \neq 1$), but not being able to bind as C and L_i do not have compatible ports. This pseudo-deadlock would only be broken if L_1 gets self-assembled with another leader or if C falls within the basin of attraction of L_i (with $i \neq 1$).

As can be seen from Fig. 4.7, 4.8, diversity of the generated population is influenced primarily by the number of copies of components. Comparing with unguided $ASAP^2$, the swarm based approach generate fewer parse tree classes than unguided software self-assembly(Fig. 4.7(a) versus Fig. 4.7(b), and Fig. 4.8(a) versus Fig. 4.8(b)).

To confirm unguided $ASAP^2$ and extended $ASAP^2$ exhibit different behaviour in terms of time to equilibrium and diversity of the generated programs, t-tests are performed on the experimental data obtained from the two models. Table 4.1 shows (for $A = 360$), all

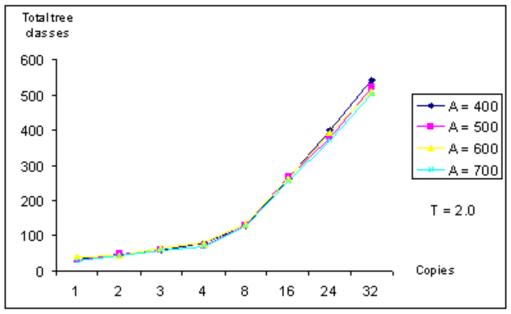


(a)

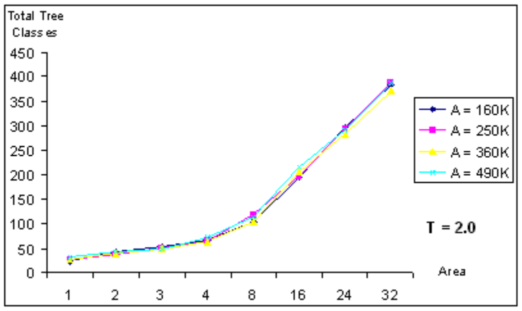


(b)

FIGURE 4.7: Relationship between total tree classes and number of copies in different temperature settings, (a) unguided $ASAP^2$, (b) swarm based $ASAP^2$



(a)



(b)

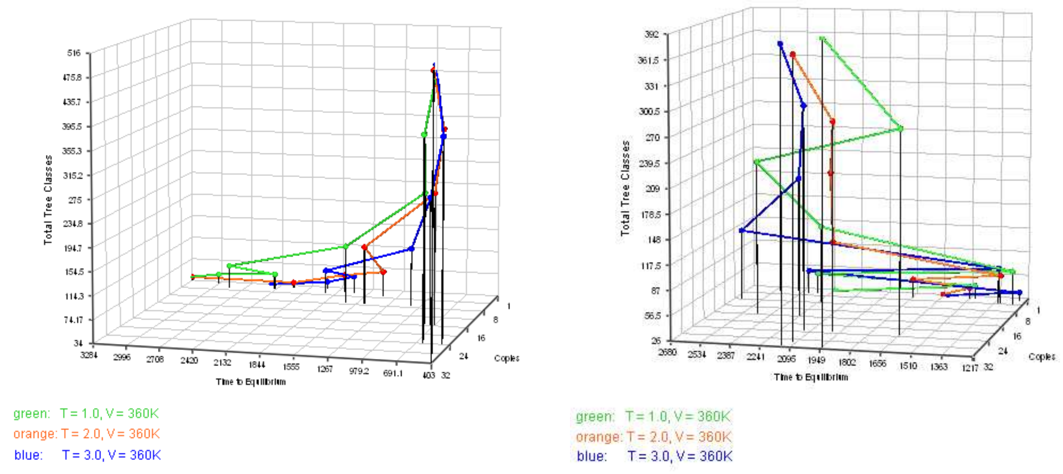
FIGURE 4.8: Relationship between total tree classes and number of copies in different area settings: (a) unguided $ASAP^2$, (b) swarm based $ASAP^2$

p-values generated by t-test are smaller than 0.05. By convention, this is considered strong evidence that the results are statistically different, namely PSO $ASAP^2$ is statistically significantly faster but also statistically significantly poorer in terms of diversity of assembled program structures.

As diversity is mainly affected by the number of copies in both systems, we present three-dimensional charts to show how it can influence the diversity along with time to equilibrium at the same time. Figure 4.9 are three-dimensional charts to show its tendency, each of which is under an identical pre-specified environment setting. As can be seen, software self-assembly with leader reaches equilibrium much quicker but produces a less

TABLE 4.1: t-test performed for $ASAP^2$ v.s. extended $ASAP^2$ with area set to 360 for both models

	time to equilibrium	diversity
N=1	$2.61 * 10^{-5}$	0.0088
N=2	0.0044	$1.76 * 10^{-5}$
N=3	0.0097	0.0017
N=4	0.0086	0.0003
N=8	0.0191	$3.61 * 10^{-5}$
N=16	$4.71 * 10^{-9}$	$3.30 * 10^{-11}$
N=24	$9.33 * 10^{-11}$	$2.54 * 10^{-13}$
N=32	$9.06 * 10^{-11}$	$1.47 * 10^{-15}$



(a) (b)

FIGURE 4.9: 3-d chart indicating the inter-relation between total tree classes, time to equilibrium and copies of components in different temperature settings using (a) unguided self-assembly (b) dynamic leaders and neighbourhood

diverse population.

4.3.3 Assessment of the predictive model on extended $ASAP^2$

In the previous chapter, a predictive model is introduced to interpolate time to equilibrium and diversity of assembled programs with the environment parameters for unguided $ASAP^2$. We examine next this predictive model for experimental data obtained with the swarm intelligence inspired system to show how much the dynamics of the $ASAP^2$ has changed. As can be seen from Fig. 4.10, greater errors occur between the average of the 20 replicas and predicted outcomes. This can also be seen from Table 4.2, where the average error rate figures become greater compared with Table 3.8. Although unguided $ASAP^2$ deviates from the behaviour of perfect gases, its dynamics could still be modelled with a few simple equations. On the other hand, for PSO $ASAP^2$ this cannot be done any more.

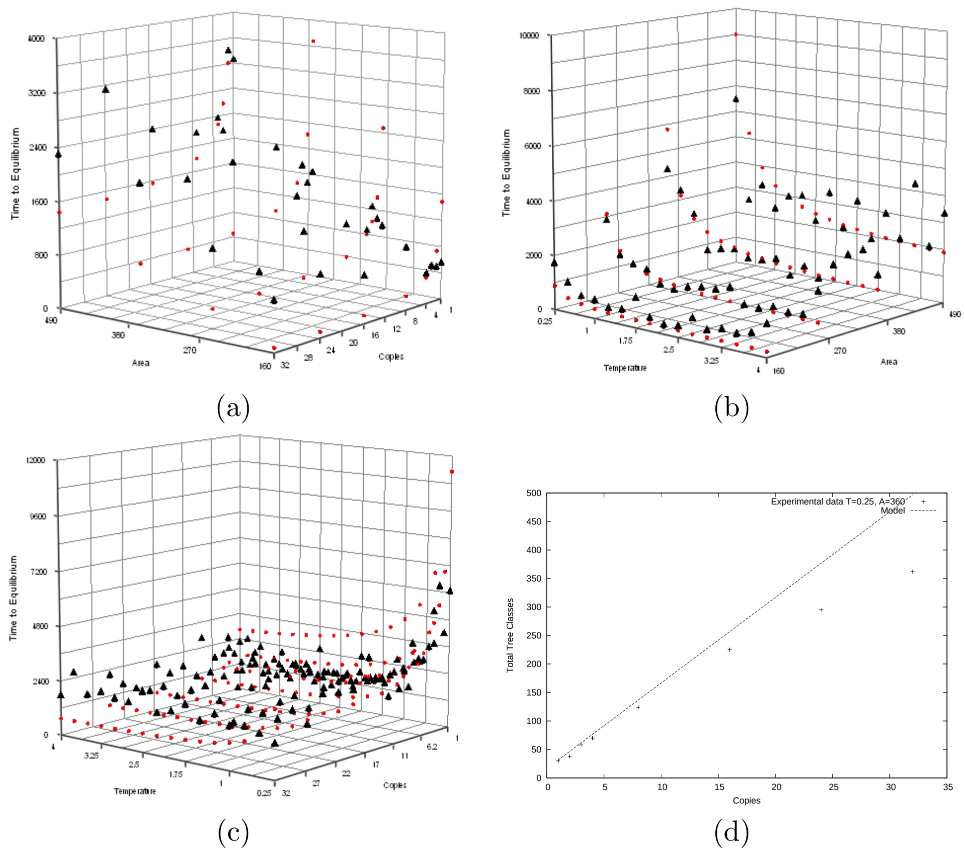


FIGURE 4.10: Prediction model assessment using extended $ASAP^2$ on (a) time to equilibrium with $T = 2.0$ (b) time to equilibrium with $N = 8$ (c) time to equilibrium with $A = 360k$ (d) diversity of the generated programs with $A = 360k, T = 0.25$. Triangles represent experimental data, and circles represent the corresponding data obtained from our predictive model.

TABLE 4.2: Error statistics for predictive model on extended $ASAP^2$

equation	category	average error rate	standard deviation
Eq. 3.2	$A = 160$	0.706	0.475
	$A = 250$	0.518	0.445
	$A = 360$	0.493	0.354
	$A = 490$	0.289	0.197
Eq. 3.3	$A = 160$	0.568	0.135
	$A = 250$	0.196	0.154
	$A = 360$	0.209	0.166
	$A = 490$	0.275	0.247
Eq. 3.4	$N = 1$	0.824	0.299
	$N = 2$	0.363	0.247
	$N = 3$	0.161	0.134
	$N = 4$	0.157	0.136
	$N = 8$	0.442	0.161
	$N = 16$	0.462	0.150
	$N = 24$	0.440	0.208
	$N = 32$	0.561	0.183
Eq. 3.5	N/A	0.167	0.115

4.4 Summary

In the previous chapter, program gas is used as a metaphor for software components. Software components are placed into a pre-specified area container and perform Brownian motions. In this chapter, we present an extended $ASAP^2$ model inspired by particle swarm optimization. The concepts of leaders and dynamic neighbourhood are introduced in software self-assembly. We illustrate how diversity of the population and software self-assembly efficiency is affected under the particle swarm regime under the same set of environment parameters (A, T, N). We report the experiments conducted on the extended system, which have shown a significant improvement in software self-assembly efficiency, i.e. time to equilibrium, at the expense of a decrease in diversity of the generated population.

Furthermore, we have assessed our prediction model based on the extended $ASAP^2$.

Results have shown greater prediction errors as the behaviour of the system has been altered when leaders are introduced.

Chapter 5

The Impact of Network Topology on Software Self-Assembly

In previous chapters, we have introduced the Automated Self-Assembly Programming Paradigm ($ASAP^2$), a software self-assembly system in which software components move and interact with each other and eventually self-assemble into programs. Software self-assembly can be embodied using various metaphors and we introduced program gas and Particle Swarm Optimization inspired approaches for this purpose. We investigated and analyzed how different factors can affect the course of self-assembly and the diversity of the generated systems in both methodologies.

Our previous investigations have focused on software self-assembly in an *unstructured* environment, where components self-assemble in a single confined space and can bind with other components provided that these are of the correct types. This chapter extends our previous studies by addressing the issue of *structured* environments. How a structured network of compartments affects the course and result of software self-assembly is the driving research question here. In this chapter, a diversified compartment approach is introduced to investigate the impact of network structures on software self-assembly. The diversified compartments are embedded in a variety of tree structures, as well as arbitrary graphs, in which each compartment can be seen as a node in the network with each node having similar characteristics to the (unique) compartment of previous chapters. Software self-assembly occurs asynchronously and in parallel within various compartments in the network. We observe how various network properties affect time to equilibrium, diversity and complexity of self-assembled programs under the new settings.

5.1 Motivation

The research we present next has two main motivations. One is the compartmentalistic approach on the origin of life: self-assembly plays an important role in nature not least in its purported role in the origin of life. There are strong arguments claiming that life originated from inanimate matter through a spontaneous and gradual increase of molecular complexity [16, 63, 73]. One such model that seeks to explain the origin of life is called *the compartmentalistic approach*. The compartmentalistic theory on the origin of life argues that compartment structures form spontaneously through self-assembly processes, and perhaps provided the original membrane-bounded environment required for cellular life to begin [18, 67]. The main concept behind the compartmentalistic approach is the fact that known life forms are based on cells, i.e. closed compartments that can keep inside of themselves a running metabolism and information polymers. [1] Compartment structures are believed to have a vital importance during this self-assembly process. The main function of life can be viewed as an interaction between the compartment and the external medium, the

[1] A virus does not have a metabolism.

interaction being realized by the flux of information and material exchanged through the boundaries of the compartments. The compartmentalistic approach is reinforced by the fact that molecules of prebiotic origin are thought to have self-assembled and formed cell-like compartments as the miscelle structures formed by amphiphilic molecules.

Although previous work on $ASAP^2$ [58, 59] resembles this compartmentalistic approach such that self-assembly takes place in a confined area, no information exchange occurs from the inside to the outside of compartments. This important aspect of compartmentalistic approach is captured by introducing a diversified compartments structure, where a network of compartments are embedded in a graph and software self-assembly takes place simultaneously in each compartment with the possibility of migration between compartments.

In the graph structure in which the self-assembly process is embedded, each vertex represents a compartment, and components can freely move to another compartment provided that the two are connected by an edge. This allows software components to hop from compartment to compartment within the network. A large variety of graph topologies are investigated, ranging from simple tree structures to general graph structures covering the transition from random to ordered graphs. We are interested in how different topologies and average inter-connection distances within the network can have an influence (if any) on the software self-assembly process, along with the resulting complexity and diversity of the generated programs.

A second motivation to use a diversified compartment approach, and one that is more closely related to computer science, is that network structures such as those studied here allow to abstract the "Internet" and study $ASAP^2$ as a phenomenon of *mobile* and *adaptive* computer code embedded in the web / grid. The network structure can be seen as a LAN (Local Area Network) or a WAN (Wide Area Network), where each node represents a computer server. While software self-assembly occurs simultaneously in each node, (partially) self-assembled programs are exchanged within the network. Thus this model is a very abstract and coarse approximation to the emergence of software structures in the "infosphere".

The diversified compartment approach is based on graphs. A graph refers to a set of vertices and a set of edges that connect pairs of vertices. The formal definition of a graph is given by Wilson and Watkins [96] as:

"A graph G consists of a nonempty set of elements, called vertices, and a list of unordered pairs of these elements, called edges. The set of vertices of the graph G is called the vertex set of G, denoted by $V(G)$, and the list of edges is called the edge list of G, denoted by E(G). An edge of the form ij is said to join or connect i and j if i and j $\in V(G)$."

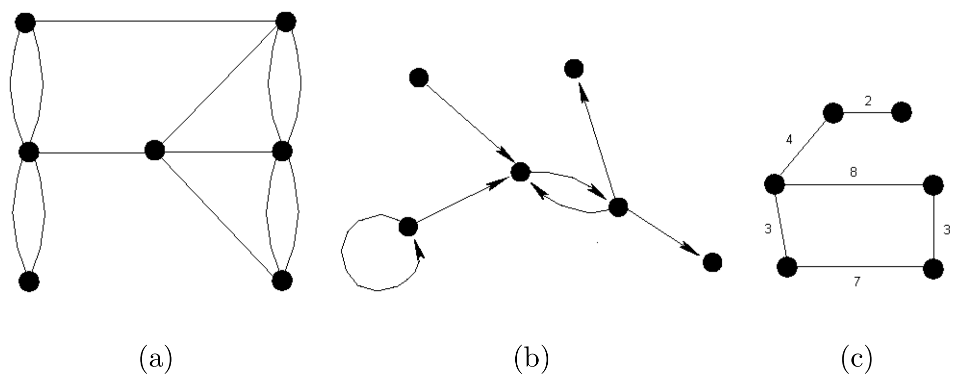


FIGURE 5.1: Example of graph structures: (a) an undirected, unweighted, non-simple (with parallel edges connecting same pair of vertices) graph; (b) a directed, unweighted, non-simple (with vertex connecting to itself) graph; (c) an undirected, weighted, simple graph

Graphs can be categorized in various ways and Fig. 5.1 illustrate examples of different graph structures. For the purpose of this research, the graphs used in the diversified compartment approach are restricted in the following ways:

- The graph is *simple*, that is, multiple edges between the same pair of vertices or edges connecting a vertex to itself are forbidden.
- The graph has to be *connected*, i.e, no isolated compartment exists. Please note that this implies that under the $ASAP^2$ model studied so far, a network-wide global equilibrium will eventually be reached.
- The graph is assumed to be *undirected.* Hence, software components can flow in and out of any vertex in the graph.
- The graph is *unweighted* because we are merely taking into account the relation between the vertices on the graphs and the connections themselves, while ignoring constraints such as distances between vertices or link capacity (for example, bandwidth, noise, etc).

Recent studies by Farley [25] have shown the impact of different population structures on the artificial evolutionary process. In [25], a variety of graph structures are used as population structures for a genetic algorithm. Individuals reside at vertices of the graph and can only choose their mating partners among their neighbours in the graph. It has been shown that the diversity and the fitness of genes can be greatly influenced with a graph embedded genetic algorithm.

5.2 Initial investigations on specific network topologies

Inspired by the studies into the topological impact upon the evolutionary process [25], it is already possible to start our investigations with some special graph structures. The aim is to find out whether software self-assembly can be affected by network structures. To do this, trees are selected as the graph structure for the $ASAP^2$ network because trees are minimally connected graphs, which means the graph is no longer connected if any one edge of the tree is removed. Three tree structures, namely string, star, and binary tree are chosen as an initial investigation on the $ASAP^2$ in a network. Figure 5.2 shows the three topologies where each compartment is represented by a square.

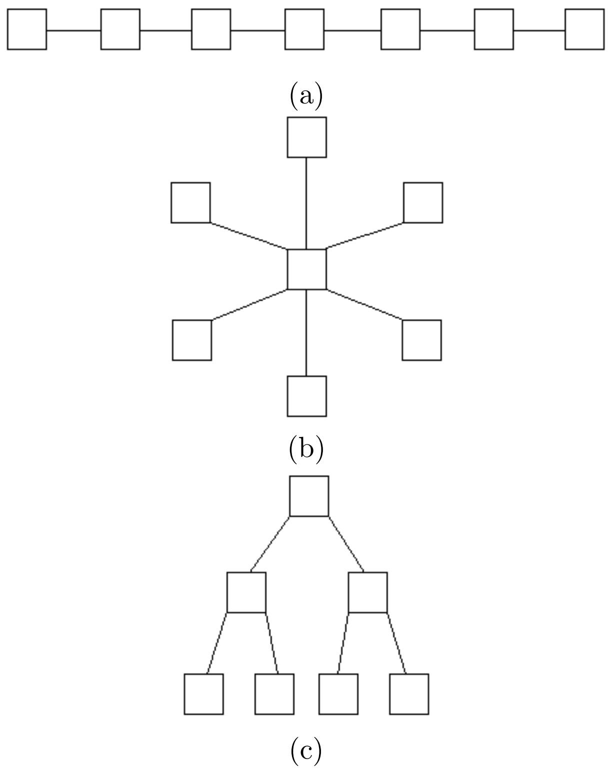


FIGURE 5.2: Trees with 7 vertices, and each vertex represents a compartment: (a) string; (b) star; (c) binary tree

5.2.1 The model

The diversified compartment approach can be explained as follows. Firstly, a graph G of a graph type *g_type* is created with a set of externally provided graph parameters. In the initial investigation where the three specific tree structures are used, the graph parameter is solely the size of the graph, i.e, the number of compartments, which are identified in a one to one fashion with vertices V in G. With the compartment set C embedded into G, software components are distributed evenly in each compartment C_i. If two compartments C_i and C_j are directly connected via an edge, C_i and C_j are identified as *neighbouring* compartments.

Previous work [58, 59] has suggested that the environment has a more obvious impact on software self-assembly with unguided dynamics than with a swarm based methodology. As we seek to find out how the environment (i.e, the network structure at this stage) affects software self-assembly, the diversified compartment approach is based on the unguided $ASAP^2$ presented in [59]. Thus, software components perform a random walk in the compartment where they "live". When a component hits a border, it is randomly transferred to one of its neighbouring compartments with equal probability.

The simulation finishes when there are no more possible binding actions between the remaining self-assembled trees *in the whole graph.* That is, global equilibrium on the network rather than local equilibrium of each compartment is the terminating condition for our simulations. [2] Algorithm 6 presents the pseudo code for the above described process.

[2]Please note that global equilibrium of the complete graph implies equilibrium at local level but not vice versa.

Algorithm 6 Graph Embedded $ASAP^2$

```
1: $G$ = GENERATE_GRAPH(g_type, graph_parameters)
2: $C$ = RETRIEVE_COMPONENTS(number_of_copies)
3: while (GLOBAL_EQUILIBRIUM_NOT_REACHED in $G$) do
4:   for each compartment $V_i$ on $G$ do
5:     for each component $C_j$ in $V_i$ do
6:       MOVE_COMPONENTS($C_j, V_i$)
7:       $V_i$ = ATTEMPT_BINDINGS($V_i$)
8:     end for
9:   end for
10: end while
```

5.2.2 Experiments

There are several fixed parameters throughout the conducted experiments. The total area (A) is fixed to $360k$ arbitrarily squared units (AU^2), and the temperature (T) is fixed to 2.0. We observe time to equilibrium (t_ε) in relation to number of compartments (V) in the graph. In addition, (t_ε) are compared between the three tree topologies versus a complete graph. A complete graph is one in which every pair of distinct vertices is connected by an edge. The complete graph G with V compartments has $\frac{V(V-1)}{2}$ edges.

5.2.3 Results

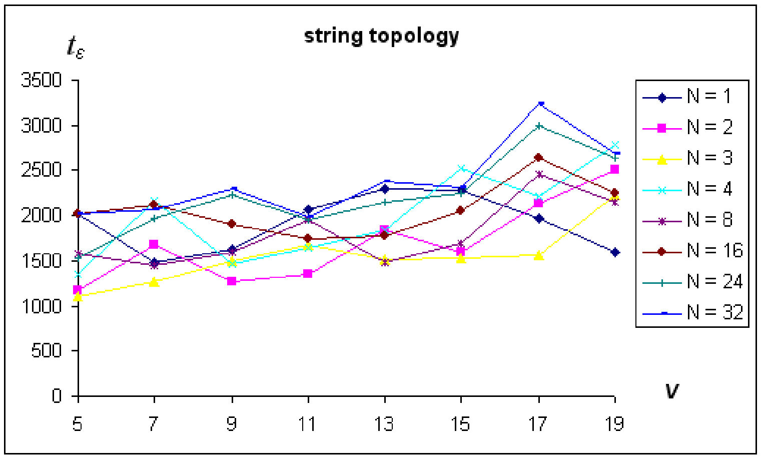
Figure 5.3 summarizes our findings. It can be seen that t_ε is influenced in different ways using different topologies. Figure 5.3(a) shows that t_ε increases with more vertices in the graph using string topology. This is because, in the worst scenario, it takes longer for a component to traverse from one end of the string topology to the other. Similarly, t_ε also increases as V increases with the binary tree topology as Fig. 5.3(c) illustrates. This is because the average length increases with more vertices introduced in the string and binary topology. Obviously the longest path length in a binary tree topology grows logarithmically, while in the string topology it grows linearly. However, Fig. 5.3(b) shows that t_ε is influenced in an opposite manner with star topology, i.e. t_ε decreases while V increases. This is because while more vertices are introduced into the network, the average path length of the graph remains constant, and components can exploit more compartments in the network to find another to bind with.

5.3 Systematic experiments with tunable graphs

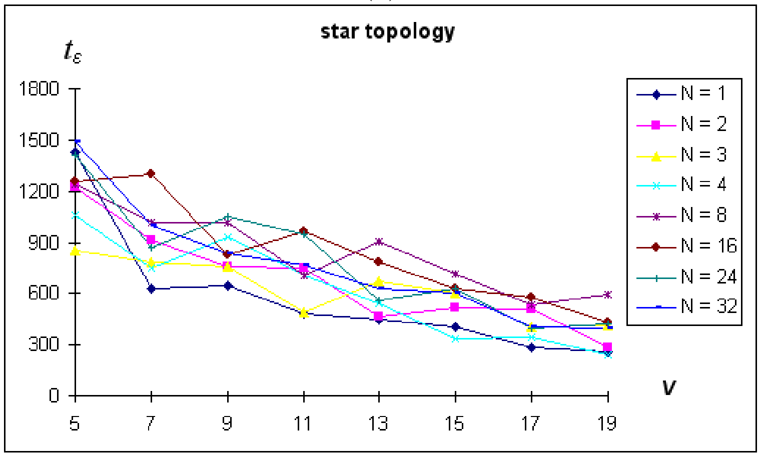
5.3.1 β-graph and graph statistics

As the preliminary experiments with the simple tree topologies suggest, the dynamics of $ASAP^2$ are changed when the process of software self-assembly occurs within a network of compartments. Furthermore, the network structure where $ASAP^2$ is embedded greatly influences its dynamics. As the relationship between V and t_ε indicates, dynamics of $ASAP^2$ is influenced by the number of compartments and the topology of the graph where the compartments are embedded. In what follows, a systematic study is conducted into how the topology of the graph affects time to equilibrium, diversity and complexity of self-assembled programs for a family of graphs.

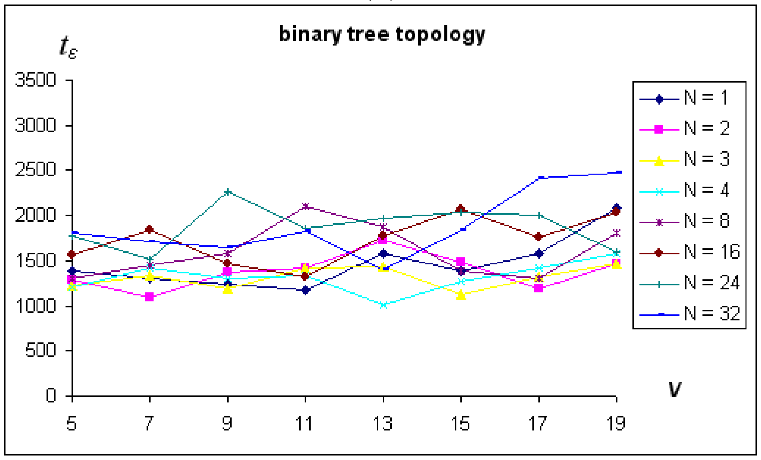
We consider length and clustering properties as a measure of graph characteristics. Studies into the length properties of a graph have been an active research area and have been performed for a number of different problem classes, for example, the performance of computer networks [26], tele-communication network [61], chemical functions, and etc [77]. Characteristic Path Length (CPL) is one of the most important statistics used to measure the *shortest* distance between every pair of vertex (i, j) in a graph, representing



(a)



(b)



(c)

FIGURE 5.3: Time to equilibrium in relation to number of vertices in the graph: (a) string, (b) star, (c) binary tree

the minimum number of edges to traverse in order to reach vertex j from vertex i. The formal definition of CPL based on a graph G is given by Watt [93] as:

"The characteristic path length (CPL) of a graph G is the median of the means of the shortest path lengths connecting each vertex $i \in V(G)$ to all other vertices. That is, calculate $d(i,j)\ \forall i,j \in V(G)$ and find $\bar{d_v}$ for each $v \in V(G)$. Then define CPL as the *median* of $\{\bar{d_v}\}$."

Figure 5.4 shows an example to illustrate how the CPL is computed. Assuming all edges have an equal length of 1, the shortest distance between A and all other vertices are 1, 2, 1, 2. In turn, the average of the shortest distance between node A to every other vertices ($\bar{d_A}$) in the graph is 1.5. This process is repeated for all the nodes i in the graph to calculate $\bar{d_i}$. As is recorded in Table 5.1, the CPL of the graph is the *median* of all $\bar{d_i}$, which is 1.5.

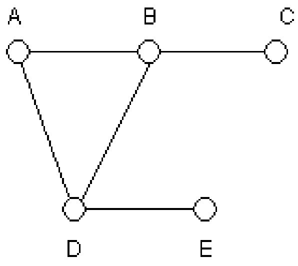


FIGURE 5.4: An example graph to show how CPL and CC are computed

TABLE 5.1: The shortest distance d between every pair of vertex (i,j) and the average shortest distance for every vertex i in the example graph

	A	B	C	D	E	$\bar{d_i}$
A	-	1	2	1	2	1.5
B	1	-	1	1	2	1.25
C	2	1	-	2	3	2
D	1	1	2	-	1	1.25
E	2	2	3	1	-	2

While the CPL is used to characterize (on average) the length of a graph, the "cliquishness" of a graph is measured by its clustering co-efficient (CC). The following terms need to be explained before the CC can be defined. The neighbourhood Γ_v of a vertex is defined as the subgraph that contains its directly connected neighbours. The degree of a vertex k_v describes the number of neighbours Γ_v. Thus, the definition of the CC is given [93] by:

"The CC γ_v of Γ_v characterizes the extent to which vertices adjacent to any vertex v are adjacent to each other." With $|E(\Gamma_v)|$ denoting the number of edges in Γ_v and $\binom{k_v}{2}$ giving the total number of *possible* edges in Γ_v, the clustering coefficient γ_v of a vertex can be formally represented as:

$$\gamma_v = \frac{|E(\Gamma_v)|}{\binom{k_v}{2}}$$

where $k_v \geq 2$. The term γ_v can be defined to be either 0 or 1 [82] when $k_v < 2$. In this thesis, we define $\gamma_v = 1$ when $k_v < 2$. Table 5.2 shows the clustering coefficient for each node in Fig. 5.4. For example, the neighbours of vertex B are A,C,D. The total number of possible connections in Γ_B is 3 and number of edges in Γ_B is 1. Therefore, γ_A equals to $\frac{1}{3}$. The clustering coefficient γ_v of a vertex describes the likelihood of the neighbours of v being connected. γ_v is 1 if every neighbour connected to v is also connected to every other vertex within Γ_v, and 0 if no vertex that is connected to v connects to any other vertex within Γ_v. The CC of G is $\gamma = \gamma_v$ averaged over all $v \in V(G)$. The computed CC for the example graph in Fig. 5.4 is 0.7333.

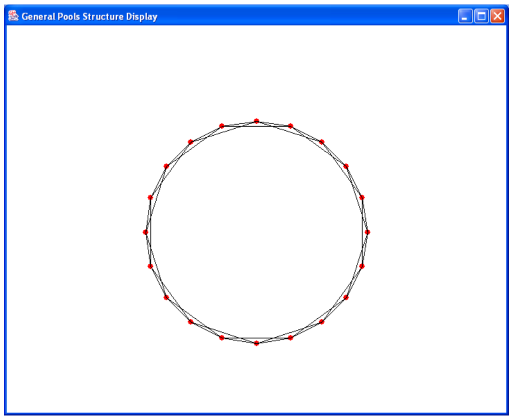
TABLE 5.2: Number of edges in Γ_v, number of possible connections in Γ_v, and the computed clustering coefficient for node v.

node	neighbours	$\|E(\Gamma_v)\|$	$\binom{k_v}{2}$	γ_v
A	{B,D}	1	1	1
B	{A,C,D}	1	3	$\frac{1}{3}$
C	{B}	-	-	1
D	{A,B,E}	1	3	$\frac{1}{3}$
E	{D}	-	-	1

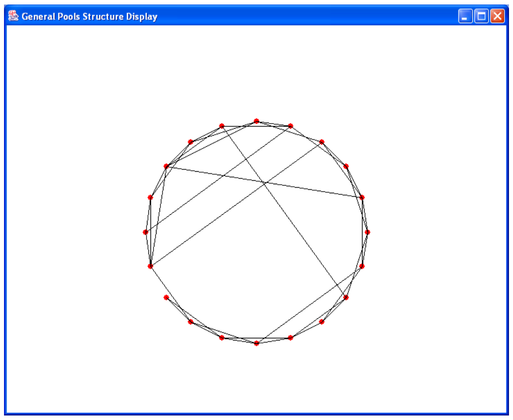
In order to systematically generate a large number of graph instances with a range of CPL and CC values, a graph model called β-graphs is used. This allows us to produce, for example, sparse or dense, highly ordered or random graphs. β-graph originates from Watts's proposal to analyze small world phenomena [93]. The question Watts tries to answer can be briefly explained as: What are the most general conditions under which the elements of a large, sparsely connect network will be "close" to each other? Based on β-graphs, we seek to answer what impact, if any, the closeness between compartments and the neighbourhood structure of the graph have on the process and results of software self-assembly.

A β-graph is represented as a ring that according to certain parameters can change from a highly ordered to a completely random graph. Three parameters are used to define the properties of graphs generated by the β-graph model.

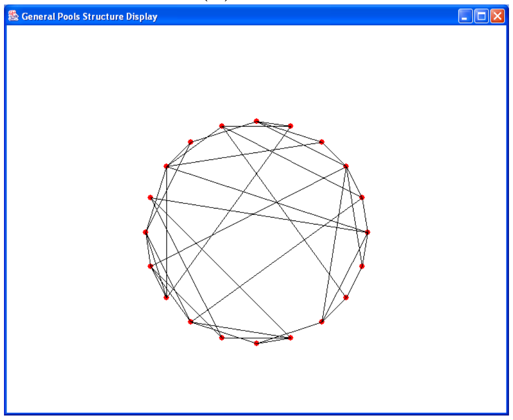
- V: the number of vertices in the graph. Each vertex represents a compartment.
- k: determines the initial number of nearest neighbours each vertex has.
- β: a probability value that determines the rewiring rate of the model, and hence randomness of the graph. In our case, this represents a connection between two neighbouring compartments.



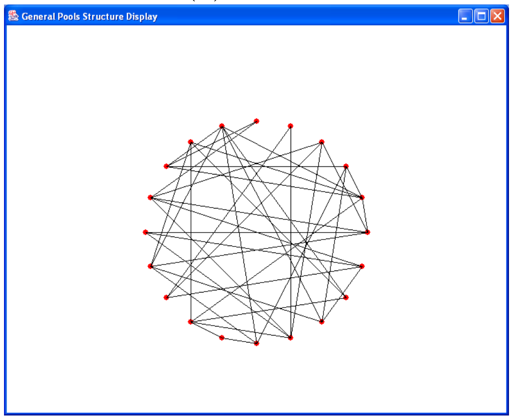
(a) $\beta = 0$



(b) $\beta = 0.2$



(c) $\beta = 0.6$

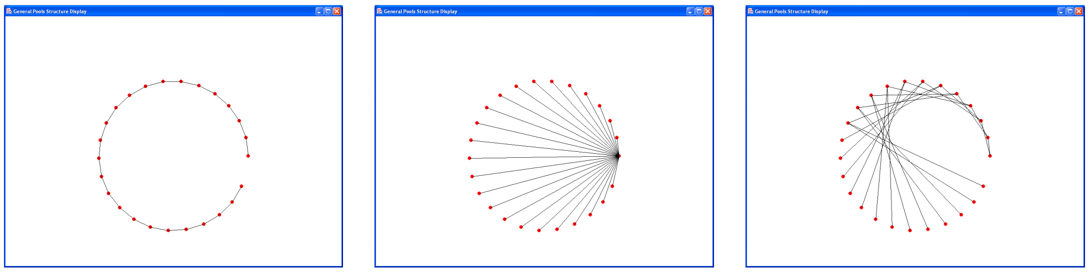


(d) $\beta = 1.0$

FIGURE 5.5: Exemplar β-graphs with $V = 20, k = 4, \beta \in \{0.0, 0.2, 0.6, 1.0\}$

With the parameters mentioned above defining the topology of β-graphs, the model starts with a perfect ring structure, where each node has precisely k-neighbours ($\frac{k}{2}$ on either side, and hence we restrict k to be an even number). Then with a given probability β, the graph is randomly rewired as follows: (1) Each vertex i and the edge which connects it to its nearest neighbour is chosen in turn in a clockwise fashion. (2) A random deviator γ is generated. If $\gamma < \beta$, then the edge connects vertex i and $i + 1$ is removed and *rewired* as (i, j), where vertex j is *randomly* chosen from the entire graph (excluding self-connection and repeated connections). Otherwise, the connection is unaltered. (3) This process is repeated for all vertices in the graph (4) After all vertices have been considered, the above procedure is repeated for edges that connect each vertex to its *next closest neighbour*, i.e. $(i, i+2)$, until all edges have been considered to be rewired once.

Figure 5.5 shows some β-graphs constructed in this fashion with $V = 20, k = 4$, and $\beta \in \{0.0, 0.2, 0.6, 1.0\}$. On the one extreme, β is set to 0 and the original structure is unaltered, and a highly ordered structure is obtained (Fig. 5.5(a)). At the other extreme, with $\beta = 1.0$, a stochastic graph with all edges randomly rewired is produced (Fig. 5.5(d)). Fig. 5.5(b) and Fig. 5.5(c) show intermediate structures. Thus, β-graphs can be gradually transformed from ordered graphs to random graphs with increasing β value. As Fig. 5.5 suggests, the properties of randomness and order are controlled by this single parameter β. β-graph captures a variety of general graph structures, and the simple population structures investigated in the previous section (i.e. string, star and binary tree topology) can be seen as specific instances of the β-graph model (Fig. 5.6).

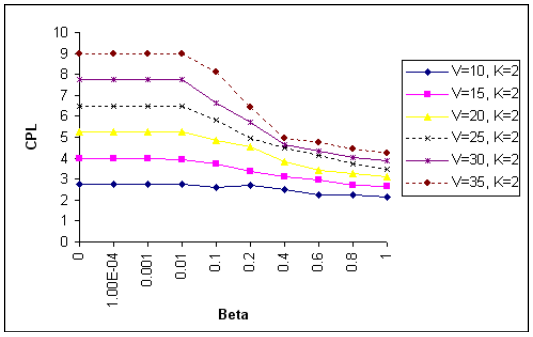

General Pools Structure Display
General Pools Structure Display
General Pools Structure Display

(a) (b) (c)

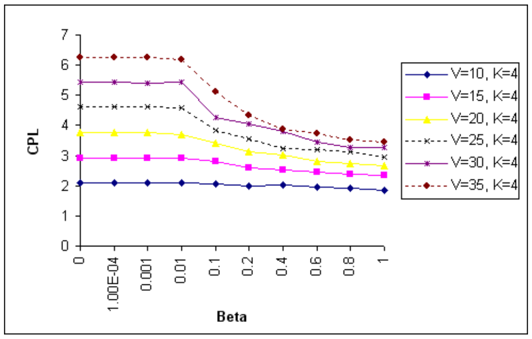
FIGURE 5.6: β-graph for specific topologies: (a) string, (b) star, (c) binary tree

5.3.2 Graph properties

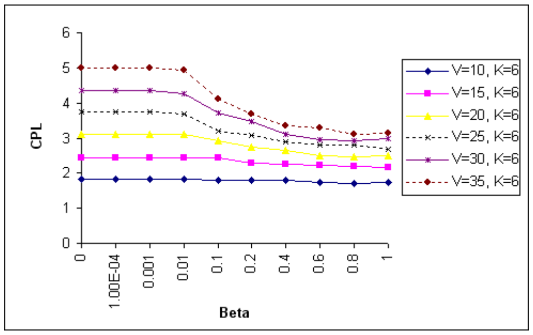
Watt investigated in [93] how length and clustering properties, i.e. the CPL and the CC, are affected by β under fixed k and V values. Figure 5.7 illustrates the relationship that exists between the CPL of a graph and parameters (β, k, V) used to construct it. It can be seen from the figures that the CPL of a graph decreases with a higher β value. And this effect is more pronounced for sparsely connected graphs (lower k). Hence, a graph with a more ordered structure results in a greater average length than a random graph does. The transition of the CPL value occurs when β is greater than 0.01 regardless of k. In addition, the decrease in CPL value against β becomes more obvious with a larger graph, i.e. a greater V value. Moreover, the CPL increases with a larger graph as Fig. 5.7 suggests. Finally, the CPL decreases when k value increases.



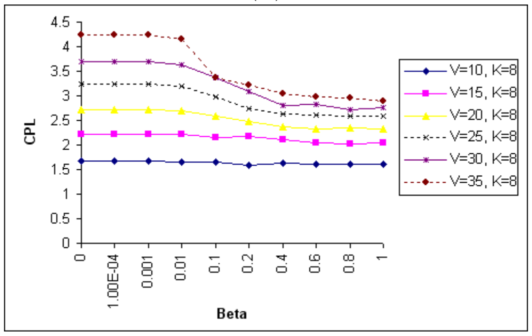
(a)



(b)



(c)



(d)

FIGURE 5.7: Relationship between CPL and β, k with: (a) $k = 2$; (b) $k = 4$; (c) $k = 6$; (d) $k = 8$

Hence, it can be concluded that, in general, a β-graph starting with higher k, β and lower V value will result in a graph with a lower average graph length. On the contrary,

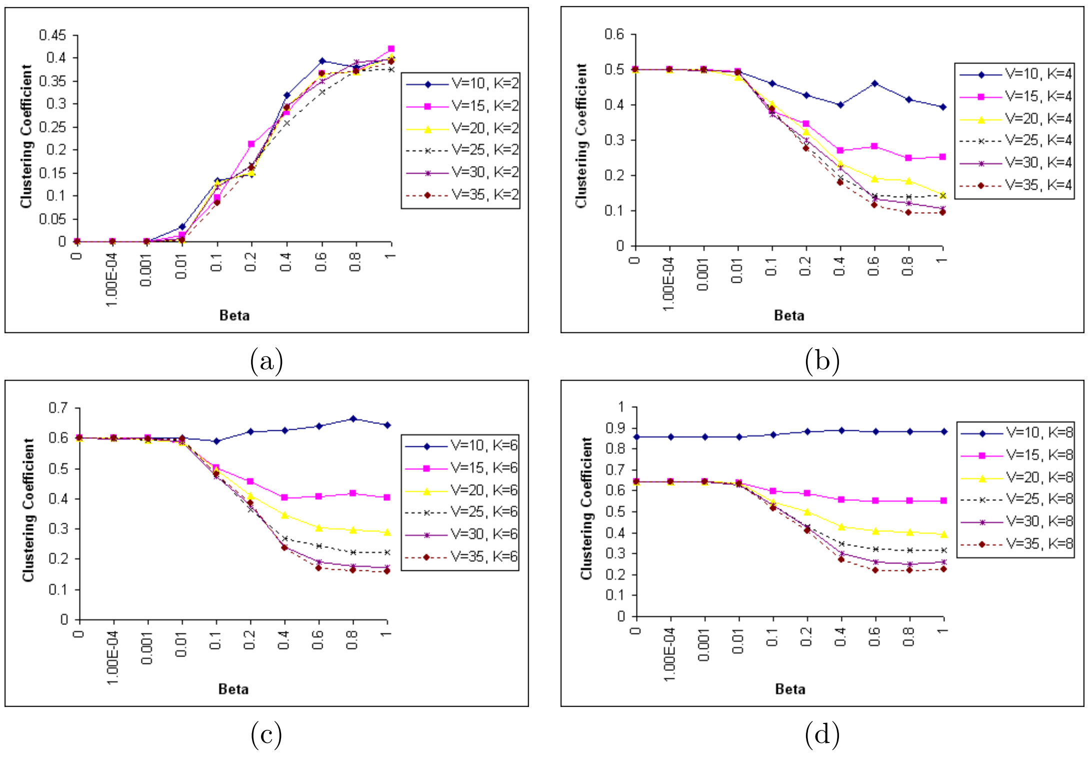


FIGURE 5.8: Relationship between CC and β, k with: (a) $k = 2$; (b) $k = 4$; (c) $k = 6$; (d) $k = 8$

a graph with high CPL value has a greater probability of being the result of a low k, β, or high V value.

Figure 5.8 illustrates the relationship between the CC and the three graph parameters (β, k, V). It can be seen that the CC behaves dramatically different with different k. When $k \in \{4, 6, 8\}$, the CC decreases as β increases. The decrease on CC becomes more significant and obvious with a higher V. In addition, CC increases as k increases while the opposite is true for $k = 2$. For all k and β, increasing V decreases the clustering co-efficiency.

5.4 Experiment

We describe next the experiments we have conducted on embodying $ASAP^2$ within β-graphs.

5.4.1 Methods

Based on the knowledge of how (β, k, V) influences the characteristic path length and clustering properties of a graph, we aim to find out the relationship between the CPL, the CC, the number of components (N), time to equilibrium (t_ϵ), the diversity (D_ϵ) and the complexity of the generated programs at equilibrium.

We fix some of the model parameters to: Temperature (T) to be 2.25, as this is the median temperature value we used in our previous work [58, 59]. The total area (A) is 5000×5000 AU^2, which is 10 times the size we used in [58, 59]. The total area (A) in the simulations is constant regardless of the number of compartments. The size of each compartment (A_i) is hence inversely proportional to the total number of compartments involved in the graph, i.e. $A_i = A/V$. Hence, the software components concentration in the compartment could increase with more compartments added to the network. However, note that for the V values used in our experiments the size of A_i is never smaller than the area of the single compartment used in the previous chapters.

Since a particular β-graph is constructed by rewiring edges in a stochastic fashion, each (β, k, V) triplet represents a *family* of graphs for which an average CPL and CC must be computed. We construct 5 graphs in each (β, k, V) triplet, with different number of copies (N) placed into each graph and run 20 replicas of the self-assembly process. The ranges for k, V, N are: $k \in \{2, 4, 6, 8\}$, $V \in \{10, 15, 20, 25, 30, 35\}$, and $N \in \{10, 25, 40, 55, 70, 85, 100\}$. As [93] suggests, a significant transition in CPL and CC occurs when β ranges from 0.0001 to 0.1. Hence, β is set in a detailed full scale, i.e. $\beta \in \{0, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.4, 0.6, 0.8, 1.0\}$. That is, a total of 151200 experiments are conducted. The average of time to equilibrium, the diversity and the complexity are calculated for each run. The complexity of a program is measured by the height and number of nodes contained in its tree representation.

5.4.2 Results

Time to equilibrium analysis

Figures 5.9 and 5.10 shows for a different number of copies of components how time to equilibrium (t_ε) is influenced by CPL and CC respectively. Most of the β-graphs constructed have CPL values less than 4 and CC values ranging from 0.2 to 0.6 as indicated in Fig. 5.9, 5.10. We wish to see how software self-assembly with multiple compartment structures differs from software self-assembly with single compartment. In order to do that, we compared the experimental results with predicted t_ε using Eqs. 3.2 and 3.3 for a single compartment structure. Each prediction data uses the same parameter settings $\{\beta, k, V\}$ as the corresponding experimental data. We take two complementary view points. First a "local" viewpoint where we would like to compare $ASAP^2$ in structured networks against the prediction obtained by a "single" compartment of equivalent size to the ones within a network. From this local view point, other compartments in the network act as the external environment. The global view point takes into account the total area of the network.

For the global view case, we set the area (A) in Eqs. 3.2 and 3.3 to be the *total* area of the compartments in the network. The predicted results are shown as $\Box$ and $+$ sign for Eqs. 3.2 and 3.3 respectively. Figure 5.9 illustrates the predictive outcome differ slightly between Eqs. 3.2 and 3.3. Moreover, as the CPL increases, t_ε gradually becomes lower than the corresponding predicted results.

For the local view point, we use as area (A) for Eqs. 3.2 and 3.3 the size of each sub-compartments in the network, that is $A_i = A/V$. The predicted results from Eqs. 3.2 and 3.3 are indicated in Fig. 5.9 as $\triangle$ and $\times$ sign respectively. It can be seen that the predicted results of Eqs. 3.2 and 3.3 are similar. The corresponding scattered points are situated at the bottom of the figures, showing a faster time to equilibrium is expected using a single compartment structure with ($A_i = A/V$), thus grossly underestimating t_ε for structured compartments, precisely because it ignores the existence of other sources of components in the network. On the other hand, Eqs 3.2 and 3.3 with $A = \Sigma A_i$ estimates, albeit with a large deviation, the median t_ε.

Comparing across panels in Fig. 5.9, it can be seen that t_ε decreases significantly from $1.2 \cdot 10^6$ to $3.5 \cdot 10^5$ as more components are placed into the network. This behaviour is similar to what we observed in the single compartment approach [59].

Figures 5.9(a), (b) also show that for $N = 10, N = 25$, t_ϵ increases to maximum when the CPL approaches 4, and then decreases for values greater than 4. On the other hand, Fig. 5.9 (c), (d), (e), (f), (g) indicate that t_ϵ on average decreases with larger CPL values, with the transition somewhere between CPL=4 and 5. The decrease in t_ε is counter-intuitive as it suggests that a network with longer average path length results in

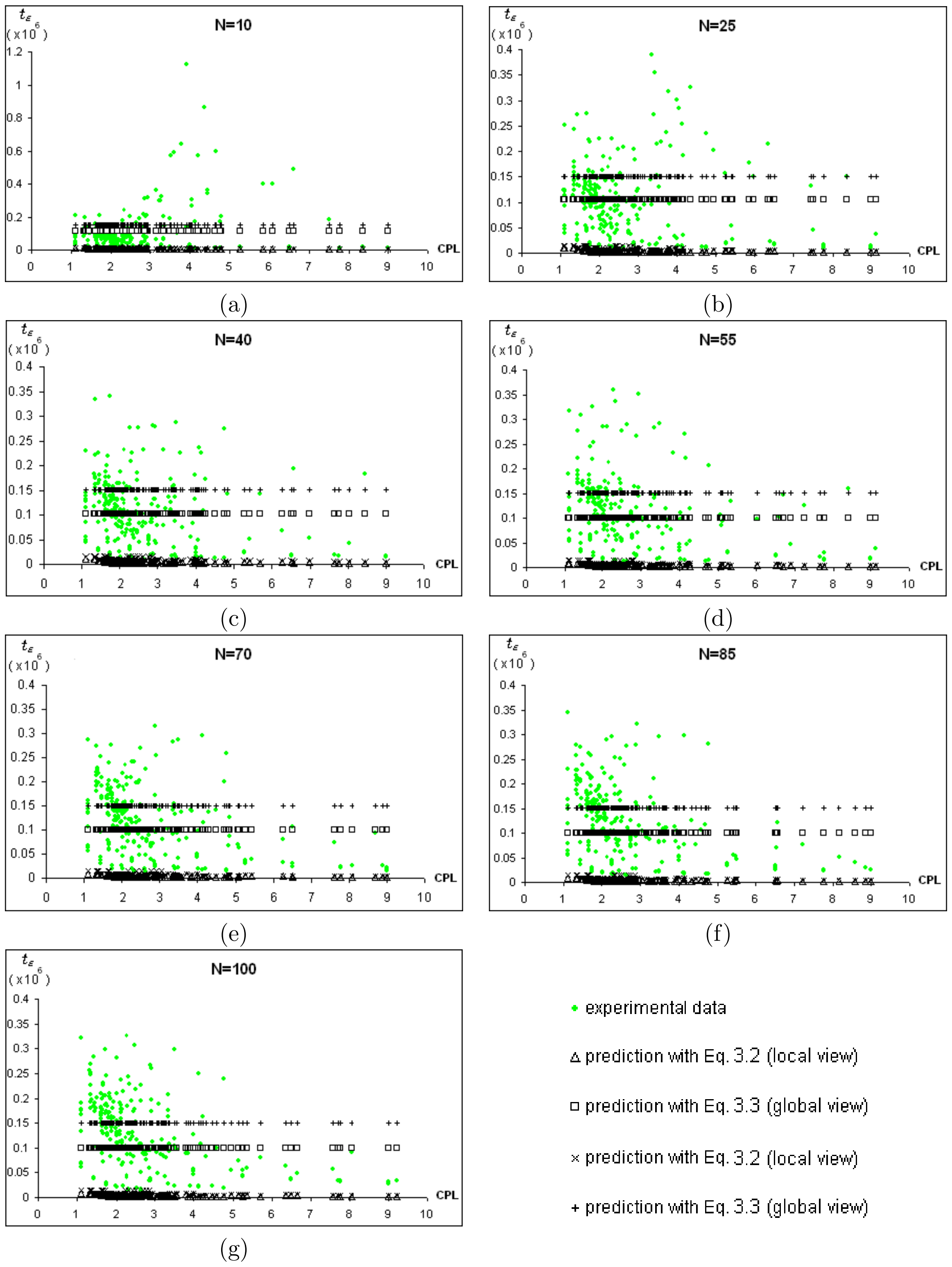


FIGURE 5.9: Relationship between time to equilibrium and CPL for runs with: (a) $N = 10$ (b) $N = 25$ (c) $N = 40$ (d) $N = 55$ (e) $N = 70$ (f) $N = 85$ (g) $N = 100$. Comparisons are made between the experimental data and predicted time to equilibrium using Eq. 3.2 ($\triangle$ and $\times$) and Eq. 3.3 ($\square$ and $+$) under the same environment parameters of the experimental data, i.e. using the same temperature ($T = 2.25$) & Area ($A = 5000 \times 5000, Ai = 5000 \times 5000/V$).

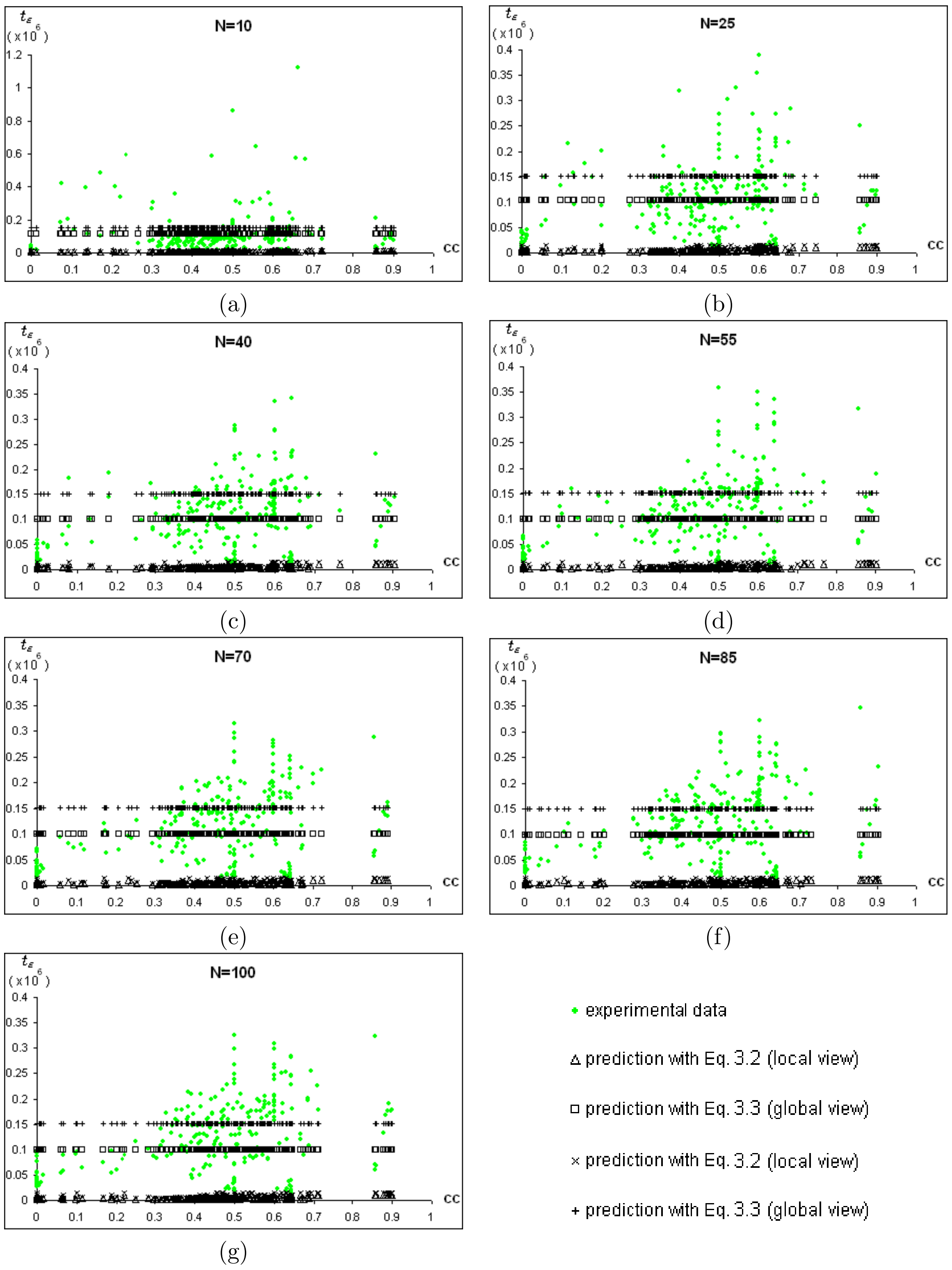
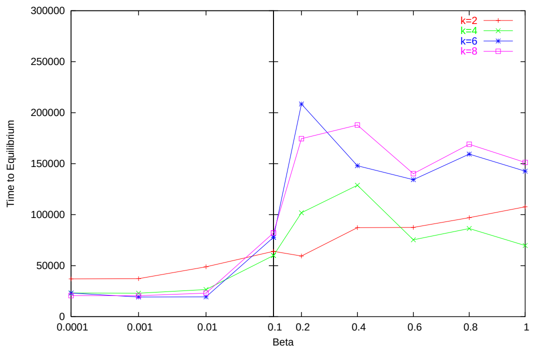


FIGURE 5.10: Relationship between time to equilibrium and CC for runs with: (a) $N = 10$ (b) $N = 25$ (c) $N = 40$ (d) $N = 55$ (e) $N = 70$ (f) $N = 85$ (g) $N = 100$. Comparisons are made between the experimental data and predicted time to equilibrium using Eq. 3.2 ($\triangle$ and $\times$) and Eq. 3.3 ($\square$ and $+$) under the same environment parameters of the experimental data as before.

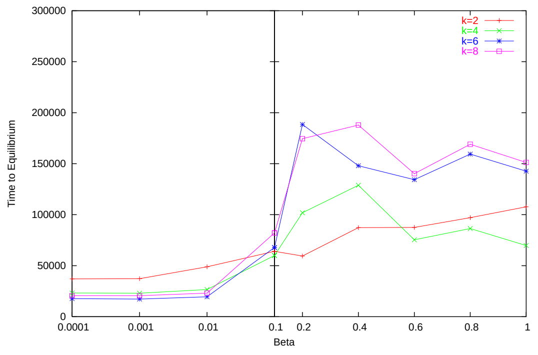
faster software self-assembly. This phenomenon is possibly due to the decrease in the size of the sub-compartments in the network. As the total area is fixed to $5000 \times 5000 AU^2$ in our experiments, a larger network with greater V will have smaller individual compartment size and hence have a higher pressure and concentration. As the results in [59] suggests, higher pressure leads to a smaller t_ε. Hence, software components find possible binding components faster in local compartments. Furthermore, taking into account Fig. 5.8, a network with more compartments tends to have smaller CC (for sufficiently large k). And as shown in Fig. 5.10, smaller CC results in faster t_ε (with a peak around 0.6).

Figure 5.10 shows how t_ε is influenced by the clustering features of the graphs. It can be seen that t_ε increases to maximum while CC approaches 0.5 - 0.6 regardless of N. In addition, Fig. 5.10 suggests that t_ε is higher than the predicted results with area equals to the size of sub-compartments, i.e. $A_i = A/V$. Also, t_ε gradually increases and eventually exceeds the predicted data as CC increases from 0 to 0.6 even when using the total area.

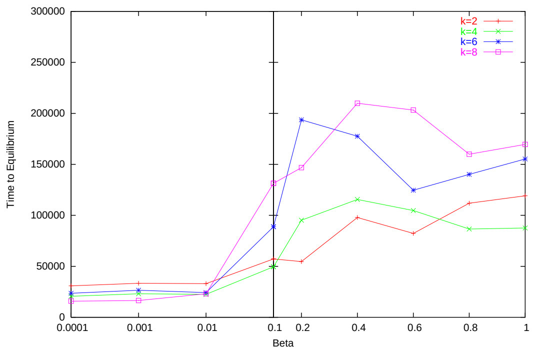
In order to have a clearer idea of how t_ε is affected by the free experiment parameters (V, β, k), Fig. 5.11 shows t_ε in relation to β with different k values $(N = 100)$. It can be seen that t_ε increases as β grows and the increase on t_ε becomes more obvious as β rises to 0.1. This suggests that an ordered graph results in a faster time to equilibrium than a random one. Figure 5.11 is split into two panels, with $\beta \in \{0.0001, 0.001, 0.01, 0.1\}$ on the left panel and $\beta \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ on the right. Comparing across panels in Fig. 5.11, it can be seen that under the same β value, t_ε is lower with a higher k with $\beta \in \{0.0001, 0.001, 0.01, 0.1\}$. However, a higher k manifests a lower t_ε with $\beta \in 0.2, 0.4, 0.6, 0.8, 1.0$. Therefore, it can be concluded that given two ordered graphs,



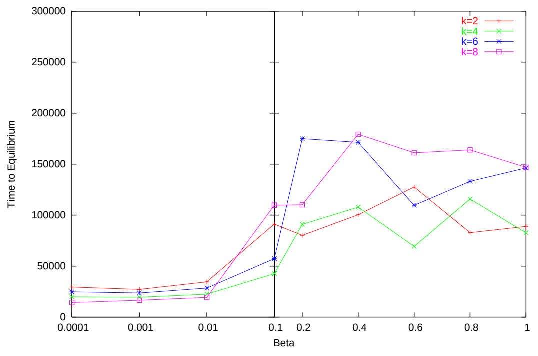
(a)



(b)



(c)



(d)

FIGURE 5.11: Relationship between t_ε and β, k with $N = 100$ and: (a) $V = 20$ (b) $V = 25$ (c) $V = 30$ (d) $V = 35$

the system will reach time to equilibrium faster in a densely connected one than a sparsely connected one. On the contrary, given two random graphs, software self-assembly running in the sparse graph will reach time to equilibrium faster than the one in the dense graph.

Diversity analysis

Figure 5.12 illustrates how the CPL and the number of copies of components affect the diversity of the generated programs. As can be seen, the total number of assembled tree classes rises with more copies of components placed into the system. In addition, fluctuations occur in the number of total distinct assembled tree classes when the CPL value ranges from 2 to 4, and the fluctuation is more obvious with a greater N.

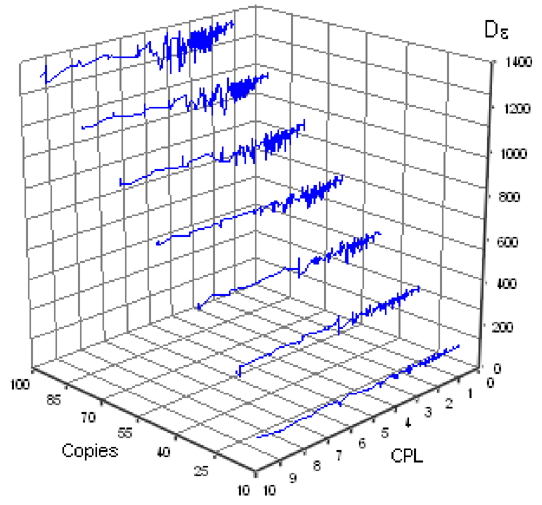


FIGURE 5.12: Relationship between D_ε and number of copies and CPL

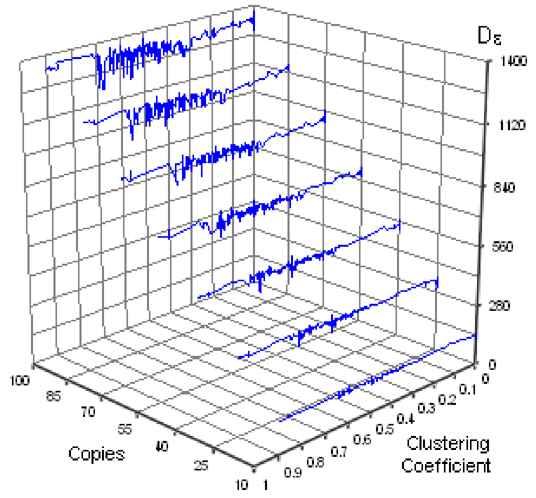
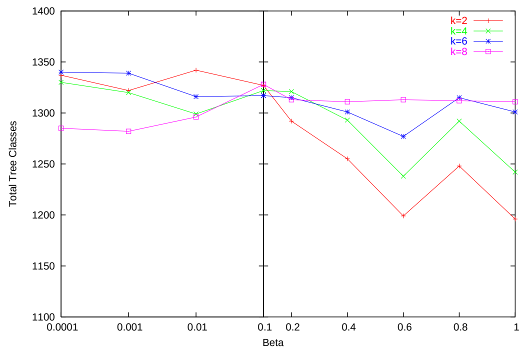


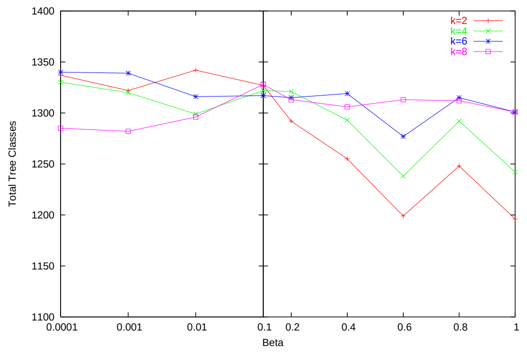
FIGURE 5.13: Relationship between D_ε and number of copies and CC

Figure 5.13 shows how CC and number of copies of components affect the diversity of the generated programs. The fluctuations in the number of total assembled trees occur under all number of copies of components. However, under a greater number of copies, the fluctuations becomes more obvious (when CC value ranges from 0.5 to 0.8) than under a small number of copies.

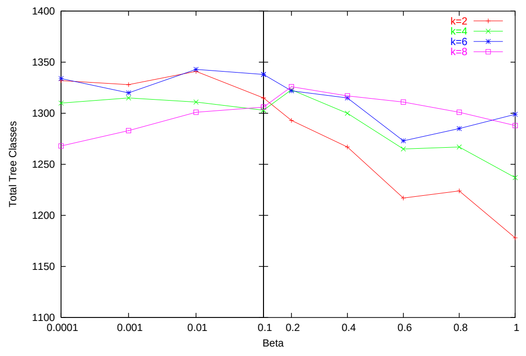
While N is fixed to 100, Fig. 5.14 illustrates how D_ε is affected by β, k with $V \in \{25, 30, 35\}$. It can be seen that diversity of generated programs decreases with an increase in β. When β ranges from 0.2 to 1.0, a higher k value results in a greater D_ε. However, under a given β value, D_ε is higher with a lower k value when $\beta \in \{0.0001, 0.001, 0.01, 0.1\}$, again showing the switching between low / high k values for the 0.01 - 0.1 transition.



(a)



(b)



(c)

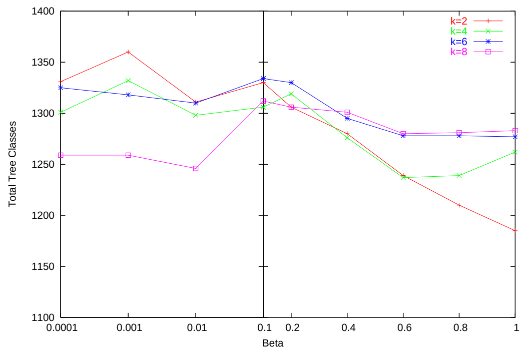


FIGURE 5.14: Relationship between D_ε and β, k with $N = 100$: (a) $V = 20$ (b) $V = 25$ (c) $V = 30$ (d) $V = 35$

Analysis of emergent complexity

As has been mentioned, the complexity of a program tree is measured by its height, h_ε, and the number of nodes it contains, n_ε, at equilibrium. Figure 5.15 presents the histograms of the number of assembled trees versus the complexity bins for different number of copies. For each copy number we differentiate graphs based on their CPL and CC. In Fig. 5.15 we can see that there is an exponentially larger number of simple, small trees than complex and large ones. Figure 5.15 shows that in general, a smaller CPL value results in a greater n_ε. Therefore, a sparsely connected network structure yields less complex programs than a densely connected network. Moreover, with a larger number of copies placed into the system, the growth rate on n_ε increases; n_ε grows linearly with the increase in the number of copies. This result matches the diversity analysis, where the total number of distinct assembled trees increases with the number of copies in a linear fashion. Figure 5.16 illustrates that h_ε behaves similarly yet not identical to n_ε. The only difference is that the growth rate of the number of assembled trees is greater with h_ε than with n_ε.

Figures 5.17 and 5.18 show the histograms for the generated programs as a function of the network's CC along with number of copies of components. The figures suggest that the number of simple and small trees is exponentially larger than the number of complex and large ones. In addition, it can be seen that the smallest and greatest CC value results in the greatest complexity (n_ε and h_ε).

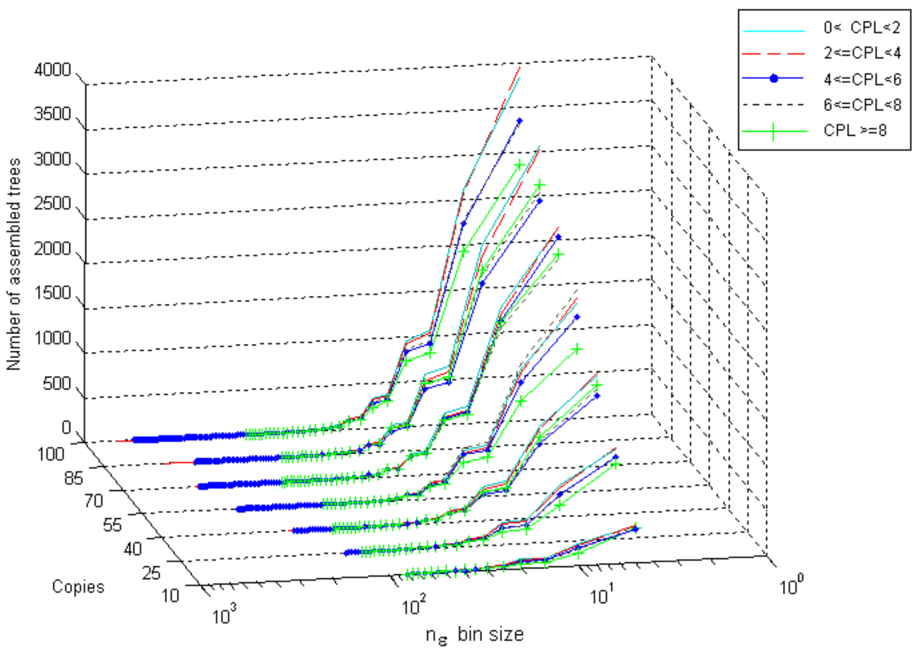


FIGURE 5.15: Histogram of average number of trees assembled in different nodes bin size and in different CPL categories

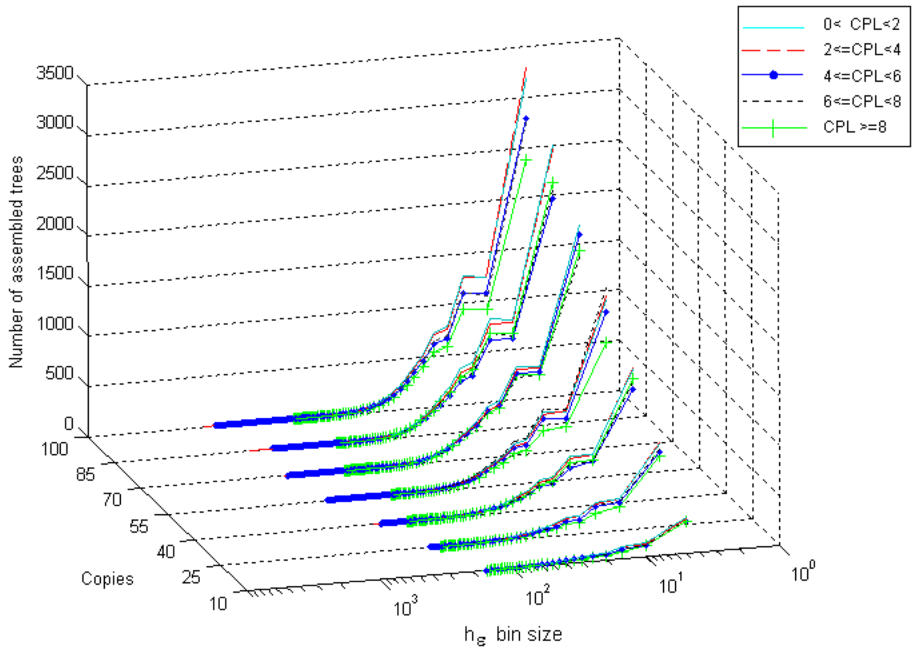


FIGURE 5.16: Histogram of average number of trees assembled in different tree height bin size and in different CPL categories

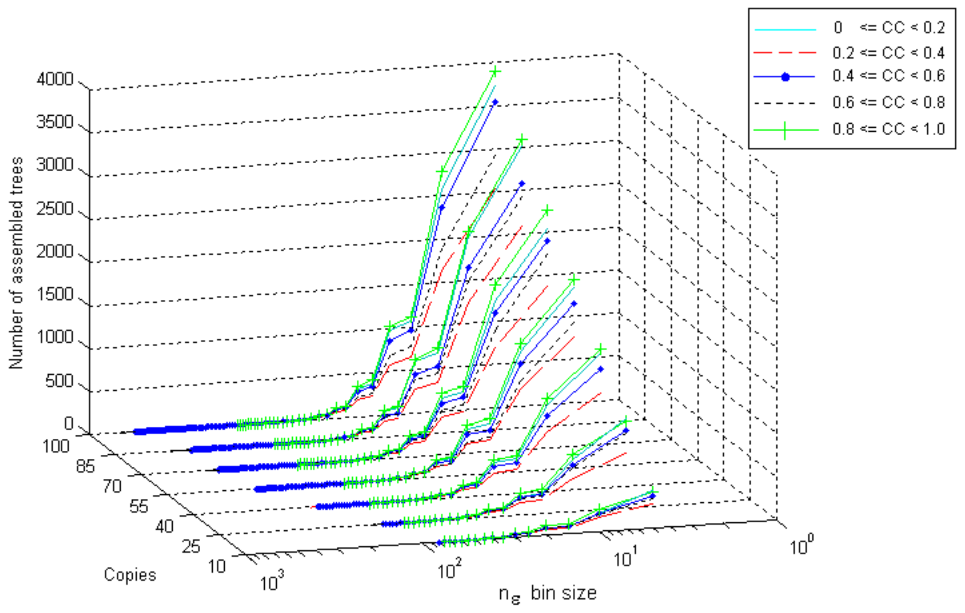


FIGURE 5.17: Histogram of average number of trees assembled in different nodes bin size and in different CC range

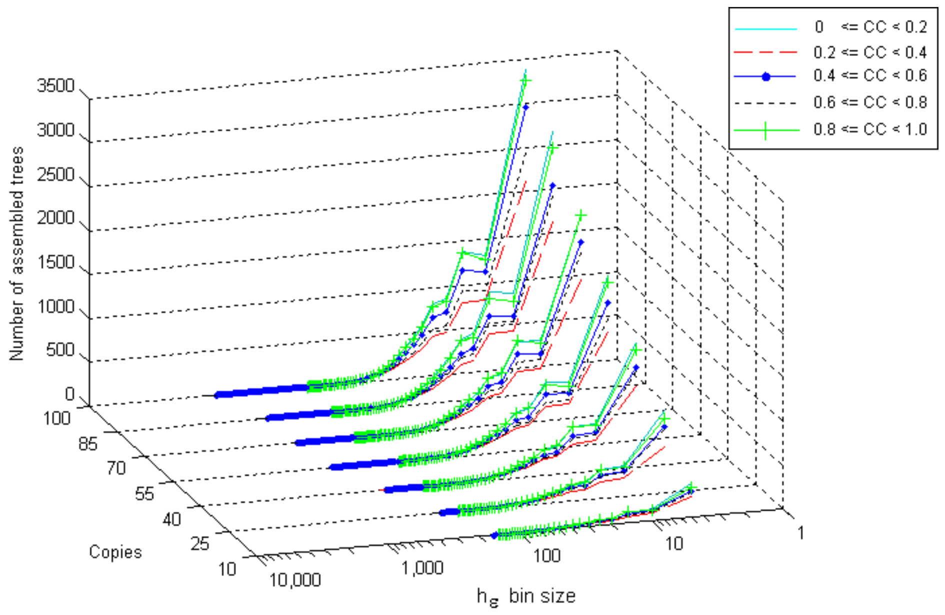


FIGURE 5.18: Histogram of average number of trees assembled in different tree height bin size and in different CC range

ANOVA analysis

In order to assess whether the number of assembled trees varies significantly for different CC and CPL, ANOVA analysis is performed on the experimental data for each copy number. The computed p-value from the ANOVA test is the probability that the variation between groups may have occurred by chance, hence a smaller p-value indicate a more significant difference. By convention, p-values below 0.05 are considered to be statistically significant. Table 5.3 illustrates the p-values grouped by CPL and CC under different number of copies of components (N), where the figures in bold indicate p-values below the conventional threshold 0.05. As can be seen, p-values are close to 1 for a small number of copies, and as N increases, p-value decreases. This means that the difference between n_ε (h_ε) in different CPL or CC groups becomes more significant with larger N. Moreover, Table 5.3 shows that CC p-values become significant earlier, i.e. with smaller N, than CPL p-values. This indicates that CC is perhaps a more important factor for software self-assembly in terms of the complexity of the self-assembled programs.

TABLE 5.3: Anova analysis for n_ε and h_ε under different number of copies. The figures in bold are those p-values close to or smaller than the threshold (0.05).

	$N = 10$	$N = 25$	$N = 40$	$N = 55$	$N = 70$	$N = 85$	$N = 100$
n_ε grouped by CPL	0.99287	0.98592	0.80157	0.3715	0.32458	0.24854	0.05604
h_ε grouped by CPL	0.98344	0.9581	0.59761	0.14386	0.13001	0.05601	**0.00697**
n_ε grouped by CC	0.97513	0.82167	0.35649	0.0576	**0.04857**	**0.02995**	**0.0139**
h_ε grouped by CC	0.93468	0.651	0.12889	**0.0451**	**0.03301**	**0.00116**	$\mathbf{3.2 \cdot 10^{-5}}$

5.5 Summary

In the previous chapters, unguided dynamics and the PSO driven approach were presented for software self-assembly simulations within unstructured compartments. In this chapter, we study the impacts of network topologies on software self-assembly.

The investigations begin with examining the use of three specific tree topologies, i.e. string, star and binary tree. It can be concluded from the experimental results that network topologies change the software self-assembly dynamics significantly. Time to equilibrium is influenced by various graph topologies in different ways. With the pre-specified *ordered* graph structures, time to equilibrium increases while the average length of the graph increases.

Thus, a diversified compartment approach for $ASAP^2$ was introduced [57, 60] to study the topological impacts on software self-assembly in greater details. The diversified compartments are based on β-graphs, a graph theoretic model that uses one parameter, β, to control the complexity of the generated graphs ranging from highly ordered to totally random ones. The graph constructed by the β-model represents a software self-assembly network system, in which each vertex in the network can be seen as a compartment where software components self-assemble locally. Components can migrate into another compartment by a connecting edge in the graph.

We overviewed how the length and the clustering property of a graph, measured by the characteristic path length (CPL) and clustering coefficient (CC) of the graph, are affected by the β-graph parameters (β, k, V). Based on the knowledge of the relationship between (β, k, V), the CPL, and the CC, we investigated how the length and clustering properties of a graph and the number of software components involved can affect the process of software self-assembly within a network. We measured: time to equilibrium, the diversity and the complexity of the generated programs. We reported the experiments conducted on the extended system, which have shown that the complexity of self-assembled programs rises with more copies of components.

Experimental results also show a counter intuitive behaviour: on average a higher CPL of the graph leads to a lower time to equilibrium. Furthermore, time to equilibrium decreases as the number of components increases, which matches our previous results based on $ASAP^2$ with single compartment structure. Moreover, regardless the number of components placed into the system, a fluctuation in diversity of the generated programs is observed when the CPL ranges from 2 to 4. The fluctuation in the total number of distinct assembled trees is also observed when the CC ranges from 0.5 to 0.8. A possible explanation for the unexpected experimental results in the analysis of time to equilibrium and diversity of generated programs is that the concentration in each compartment is changed because of the change in the CPL and the CC.

In addition, as the experimental results suggest, given two *ordered* graphs, sparsely connected graphs yield more diversed programs, also taking longer to reach time to equilibrium, than densely connected graphs. On the limit a densely connected graph would behave as a simple large compartment. On the contrary, given two *random* graphs, $ASAP^2$ produces less diversed programs and takes shorter time to reach equilibrium in sparsely connected graphs than in densely connected graphs. Together these two observations would seem to suggest that posing the network at the "edge of chaos" [55], i.e. neither too random nor too ordered, might be the optimal choice for achieving maximum diversity.

Experimental results have also indicated that a sparsely connected network structure (larger CPL values) yields less complex programs than a densely connected network (small CPL values). In addition, a network structure with CC ranging from either 0 to 0.2 or from 0.8 to 1.0 manifests greatest complexity of the generated programs.

The $ASAP^2$ systems introduced so far are based on static self-assembly in which an equilibrium state will eventually be reached. In the next chapter, we investigate software self-assembly *far* from equilibrium. The dynamic $ASAP^2$ system introduces a permanent state of dis-equilibrium by imposing a flow of software components by means of source and sink compartments, thus resulting in a different concentration of components in different parts of the network.

Chapter 6

The Impact of *Open* Network Topology on Software Self-Assembly

Self-assembly systems can be categorized in various ways. Whitesides and Grzybowski [94] categorize self-assembly systems into static self-assembly and dynamic self-assembly. Dynamic self-assembly systems involve dissipation of energies. A final structure is self-assembled while the minimum energy form is obtained. Examples of these self-assembly systems can be found in weather and solar systems. In static self-assembly, an equilibrium will eventually be reached, and the final assembled structure is stable at equilibrium.

Previous chapters focus on a *static* software self-assembly system in which an equilibrium will eventually be reached when no further binding actions between software components is possible. To complete this preliminary stage of research for software self-assembly, we introduce *dynamic* software self-assembly in this chapter. The extended $ASAP^2$ is based on the diversified population structure introduced in the last chapter. Software components are added from the influx pool and removed from the outflux pool based on a pre-defined probability during the self-assembly process. We observe how the dynamics of $ASAP^2$

far from equilibrium have changed. We do this by studying how diversity and complexity of self-assembled program structures are affected by flux rate as well as a range of graph parameters used in the previous chapter to make the comparisons.

6.1 Motivation

Prior to the emergence of evolutionary processes, it has been widely argued that elementary and replicative objects relied upon a basic mechanism of interaction in order to self-assemble to form highly sophisticated entities for evolution to begin [76]. The transition between raw interactions of rudimentary components to evolutionary change has been a widely discussed research topic. The compartmentalistic theory on the origin of life argues that compartment structures form spontaneously through self-assembly processes, and perhaps provided the original membrane-bounded environment required for cellular life to begin [18, 67].

In addition, recent studies on genomes from various species have suggested that the functional characteristics of organisms are not directly related to the number of genes encoded within the organism [29, 64]. In [29], Görnerup and Crutchfield presented finitary process soup to study the hypothesis that diversity and complexity of genomes arises from a hierarchy of interactions between genes and between interacting gene complexes. The finitary process soup involves a set of elementary objects called σ-machines that perform local and elementary operations. A new σ-machine can be generated as a result of interactions between two smaller σ-machines. The finitary process soup model seeks to answer how the transition between raw interactions of rudimentary components to evolutionary change occurs in terms of structural complexity. When dynamic influx and outflux is introduced, it has been shown that the global complexity in the finitary process soup is mainly related to the emergence of higher levels of organization.

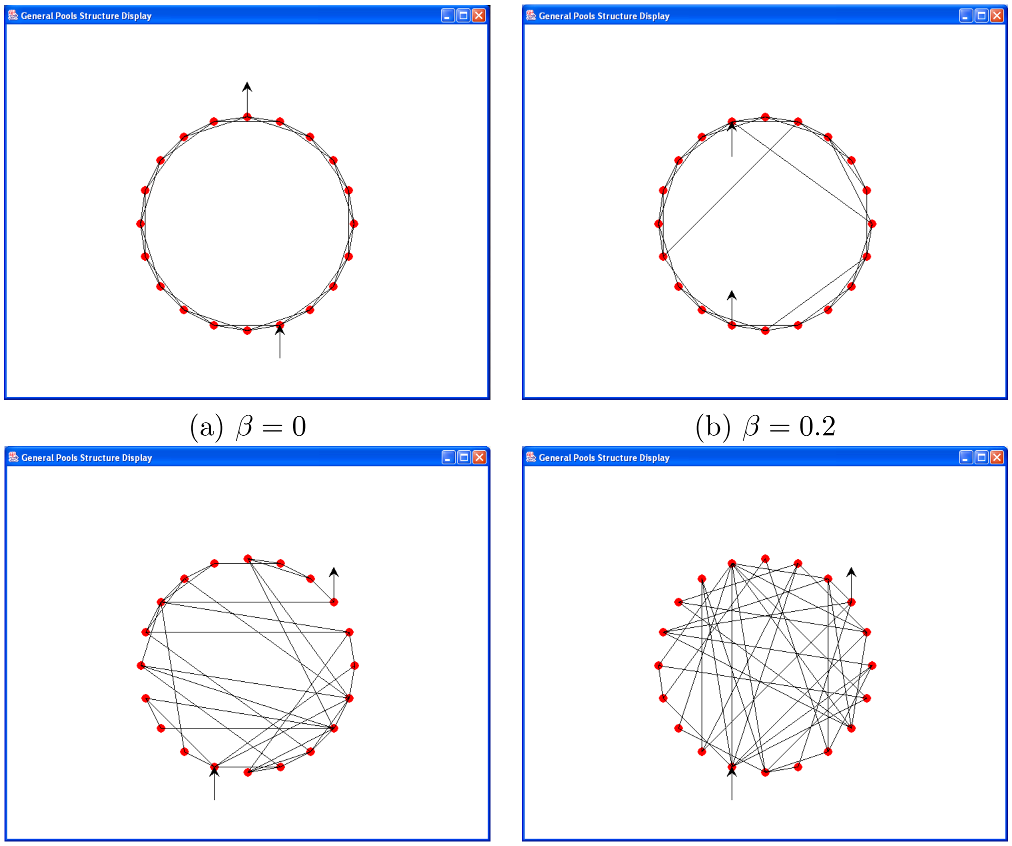
In the previous chapter, we have presented automated self-assembly programming paradigm ($ASAP^2$) based on a network of compartments. Software components can be exchanged between compartments in this network via edges. An equilibrium will eventually be reached in this *close* system when no more binding is possible. Inspired by Görnerup and Crutchfield's work and the compartmentalistic approach on the origin of life, we extend $ASAP^2$ and investigate the impact of *open* network topology on complexity and diversity of self-assembled programs.

6.2 Model Descriptions

6.2.1 β-graph model

The extended system is based on the diversified compartment approach on $ASAP^2$, where β-graphs [93] are used to produce a family of network topologies, ranging from completely random graphs to highly ordered graphs. The β parameter is a probability value that determines the random rewiring rate of edges from an initially completely ordered ring structure.

Figure 6.1 shows examples of β-graphs constructed with $\beta \in \{0.0, 0.2, 0.6, 1.0\}$. As can be seen from Fig. 6.1, β-graphs can be gradually transformed from ordered graphs to random graphs with increasing β value. A highly ordered structure is shown in Fig. 6.1(a) where β is set to 0. On the other extreme, Fig. 6.1(d) illustrates a complete stochastic graph with all edges rewired when β is set to 1.0.


General Pools Structure Display
General Pools Structure Display
General Pools Structure Display
General Pools Structure Display

(a) $\beta = 0$ (b) $\beta = 0.2$

(c) $\beta = 0.6$ (d) $\beta = 1.0$

FIGURE 6.1: Exemplar β-graphs with $\beta \in \{0.0, 0.2, 0.6, 1.0\}$, where influx and outflux compartments are indicated by arrows pointing to and coming out from them.

6.2.2 Implementation

We introduce a dynamic flow of software components in the network such that the resulting software self-assembly system is far from equilibrium. When a β-graph is constructed, a random node in the graph is selected as an *influx* compartment. The *outflux* compartment is selected such that it has the greatest distance to the influx compartment in the network. As can be seen in Fig. 6.1, an influx compartment is represented by a node with an arrow pointing at it, whereas an outflux compartment is indicated by a node with an arrow pointing to the opposite direction. Influx and outflux of components introduce local gradient of concentration of components in each compartment. The concentration level in each compartment is dependent on its distance to the source.

The simulation starts by distributing software components retrieved from the software repository evenly into each compartment in the network. The dynamics of the population is iteratively ruled by introducing new components at the influx compartment and removing old components at the outflux compartment as follows.

At each time step t, a randomly generated software component C_r is added to the system from the influx compartment (V_{in}) with an externally defined probability value Φ_{in}. The size of C_r is determined by a roulette wheel selection from 1 to the size of the largest component in the compartment. The roulette wheel is constructed such that a smaller component size has a significantly greater chance to be selected than a large component size. When C_r is added in the influx compartment, a component of the same size (i.e. same number of nodes) will be taken out from the outflux pool (V_{out}). However, if there are no components of the same size as C_r in the outflux compartment, no software components will be removed (i.e. the influx and outflux of components are not always balanced). Hence, the system is far from equilibrium and does not follow a dynamic self-assembly process [1]. Algorithm 7 formally describes the process in pseudo-code.

[1]Hence it is impossible to define t_ε for D_ε as for static $ASAP^2$

Algorithm 7 Graph embedded $ASAP^2$ with dynamic influx and outflux of components, where V_{in} and V_{out} indicate influx and outflux compartment respectively

```
1:  $G$ = GENERATE_GRAPH(graph_type, graph_parameters)
2:  $V_{in}$ = random compartment in $G$
3:  $V_{out}$ = furthest node from $V_{in}$ in $G$
4:  while (termination condition not satisfied) do
5:      $C$ = RETRIEVE_COMPONENTS(number_of_copies)
6:      $\Phi_{in}$ = components influx rate
7:      At each time step $t$
8:      $r$ = uniform random number between 0 and 1
9:      if ($r < \Phi_{in}$) then
10:         add a random component $C_r$ in $V_{in}$
11:         remove component of the same size as $C_r$ in $V_{out}$
12:     end if
13:     for (each compartment $V_i$ on $G$) do
14:         for (each component $C_j$ in $V_i$) do
15:             MOVE_COMPONENTS($C_j, V_i$)
16:             $V_i$ = ATTEMPT_BINDINGS($V_i$)
17:         end for
18:     end for
19: end while
```

6.3 Experiment

6.3.1 Methods

The extended $ASAP^2$ is based on the diversified compartment approach presented in [57]. In order to compare the difference between $ASAP^2$ close to equilibrium and far from equilibrium, we use the same fixed environment settings as in [57]; that is, $T = 2.25, A = 5000 \times 5000 AU^2$. In addition, the number of copies (N) and the number of compartments in the network (V) are also fixed parameters for our simulation.

The free model parameters of the simulation are β, k, and influx rate Φ_{in}. β and k determine the graph properties (i.e, CPL, CC), Φ_{in} controls the likelihood of a new component being introduced into the system. The ranges for β, k, ϕ_{in} are: $\beta \in$

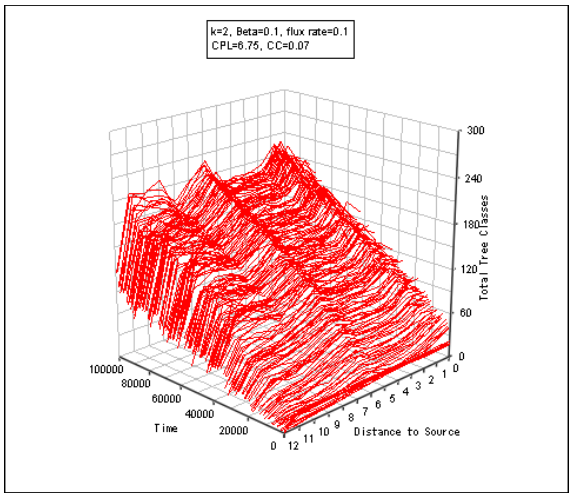
$\{0.1, 0.4, 0.6, 1.0\}$, $k \in \{2, 4, 6\}$, $\phi_{in} \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$.

With $N = 100$ and $V = 30$, we measured: the complexity and diversity of generated programs over time (t) in each compartment, which is characterized by its *shortest* distance to the influx compartment (d_α). Then we observe how complexity and diversity vary under a range of β-graphs, parameterized by the set of β and k settings. Diversity of self-assembled programs (D_φ) is assessed by the total number of different tree classes. The diversity measure is used to keep consistent with the investigations presented in the previous chapters. That is, two programs are considered to belong to the same parse tree class if their *structures* and *content* are both identical [11]. The complexity of a program tree is measured by its height and the number of nodes it contains as in [57].

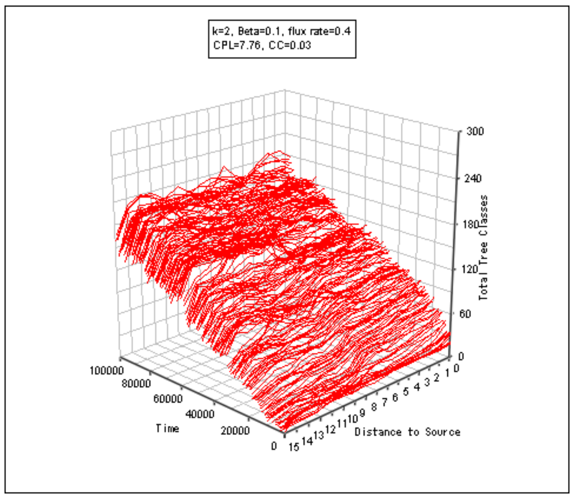
6.3.2 Results and analysis

Diversity analysis

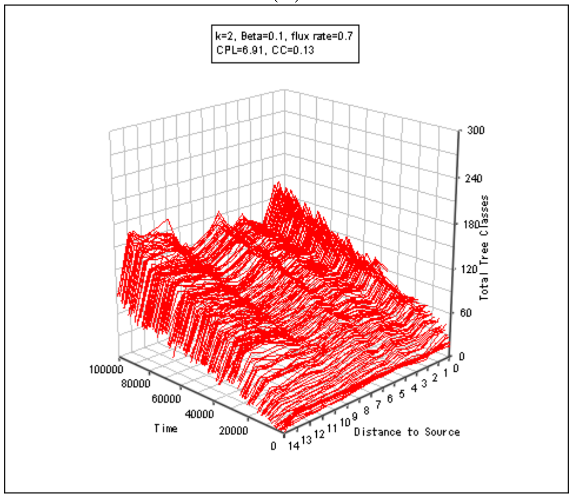
The following figures from Fig. 6.2 to Fig. 6.13 illustrate how D_φ is influenced by t and d_α under each external parameters settings: $\{\phi_{in}, k, \beta\}$. β values are grouped by different coloured sub-charts for Fig. 6.2 to Fig. 6.13. In general, it can be seen that the diversity of the self-assembled programs increases as software self-assembly progresses over time. In addition, compartment locations in relation to the source in the network plays an important role in the diversity of assembled programs.



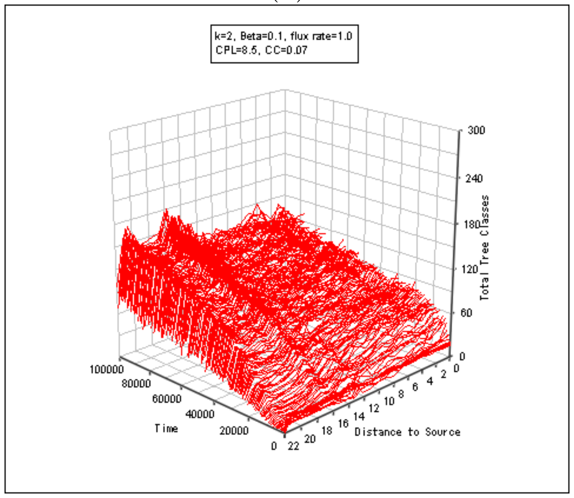
(a)



(b)

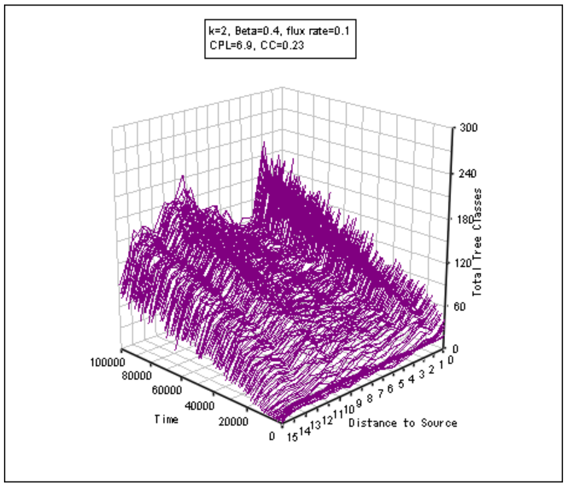


(c)

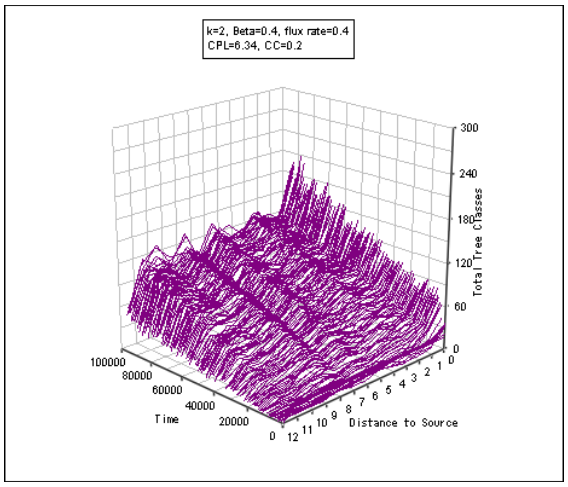


(d)

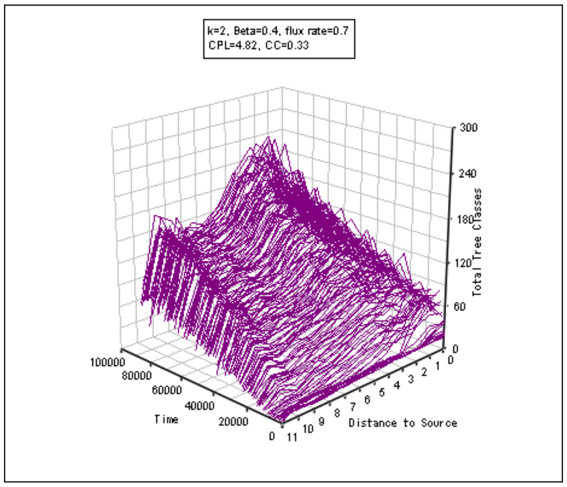
FIGURE 6.2: D_φ in relation to t and d_α, with $k = 2, \beta = 0.1, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



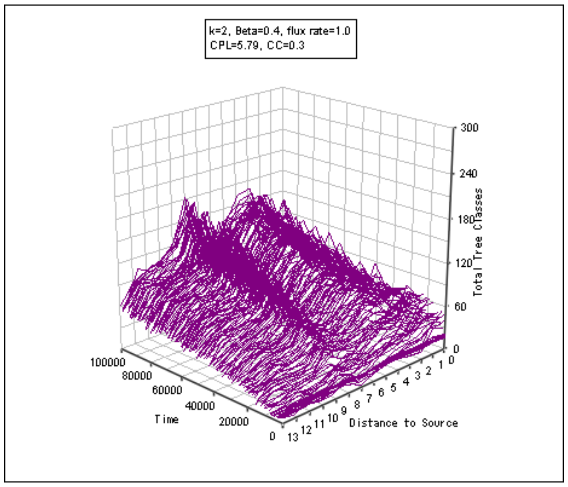
(a)



(b)

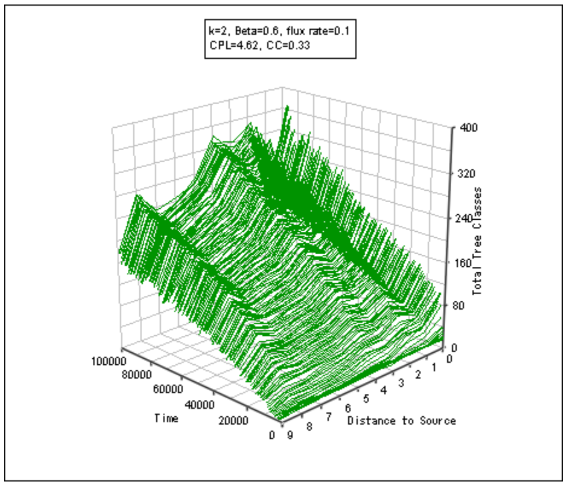


(c)

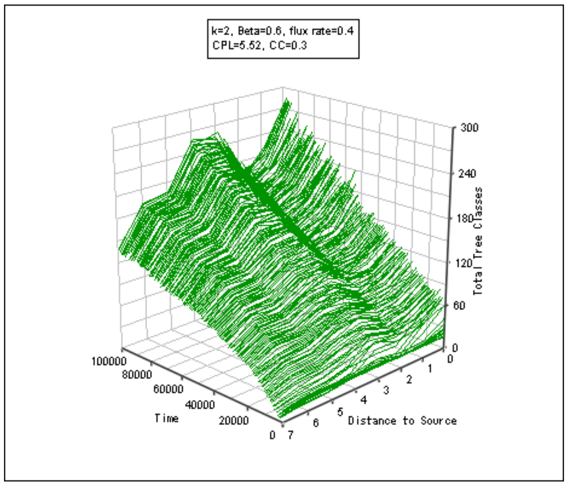


(d)

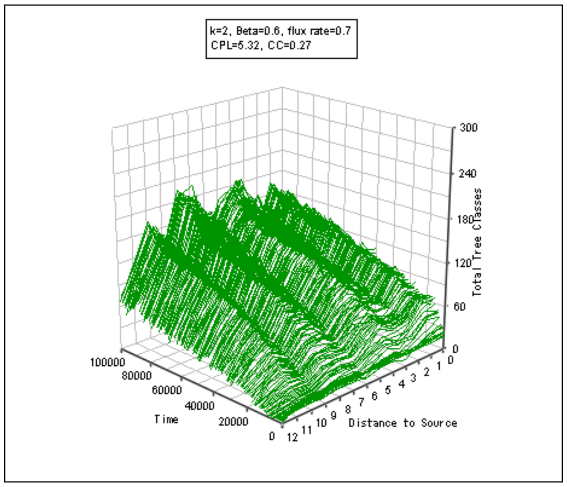
FIGURE 6.3: D_φ in relation to t and d_α, with $k = 2, \beta = 0.4, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



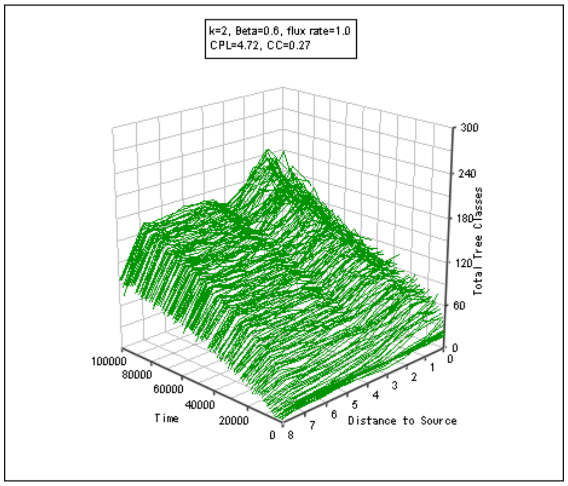
(a)



(b)

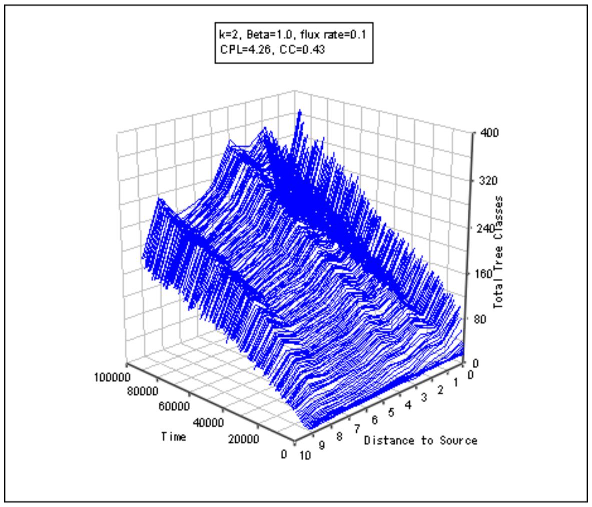


(c)

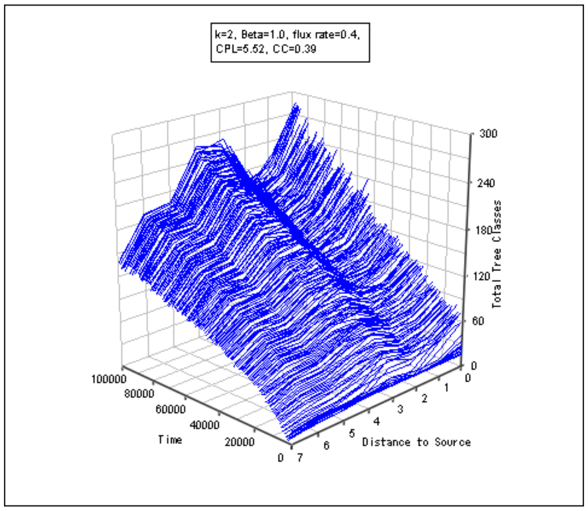


(d)

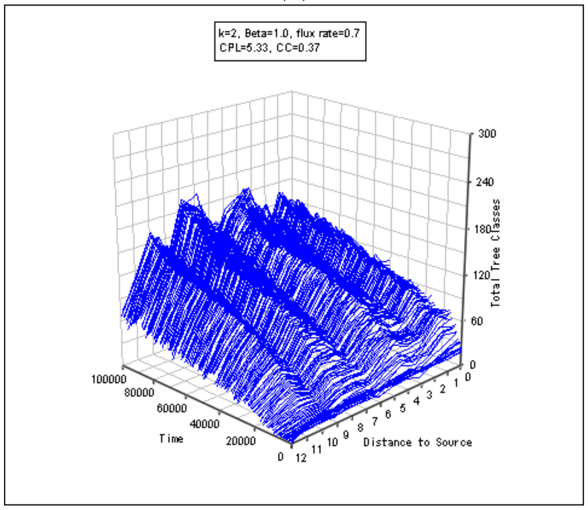
FIGURE 6.4: D_φ in relation to t and d_α, with $k = 2, \beta = 0.6, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



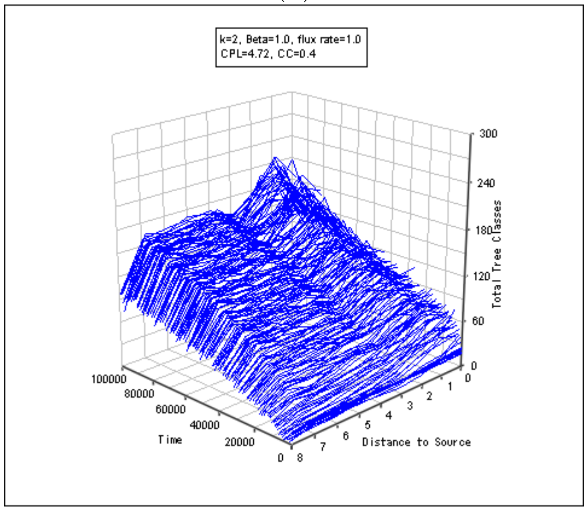
(a)



(b)

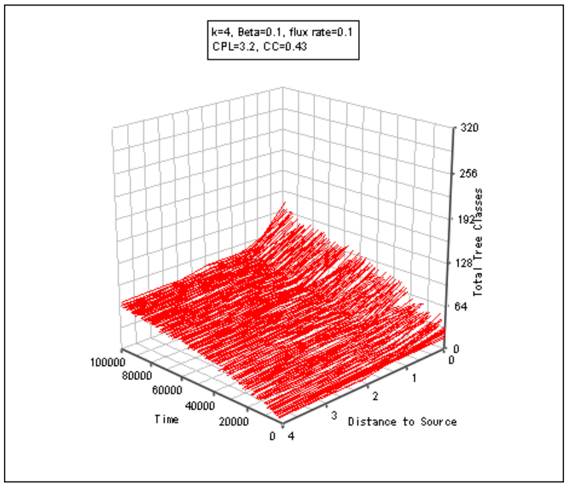


(c)

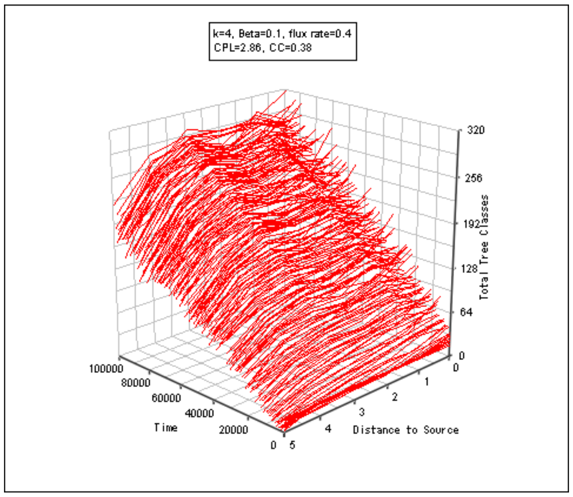


(d)

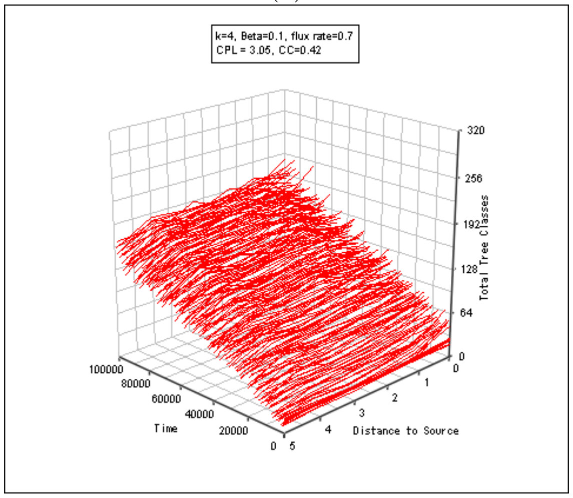
FIGURE 6.5: D_φ in relation to t and d_α, with $k = 2, \beta = 1.0, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



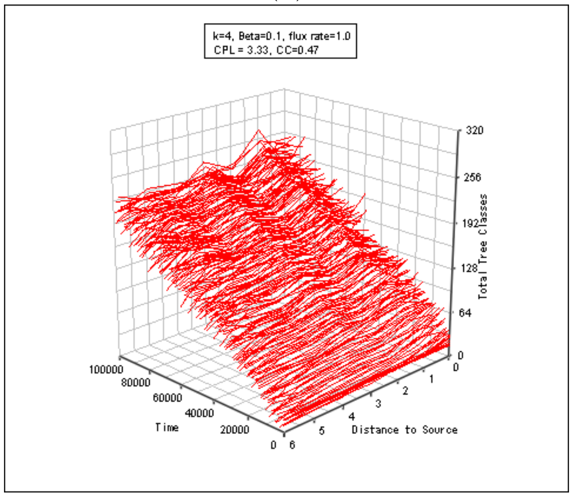
(a)



(b)

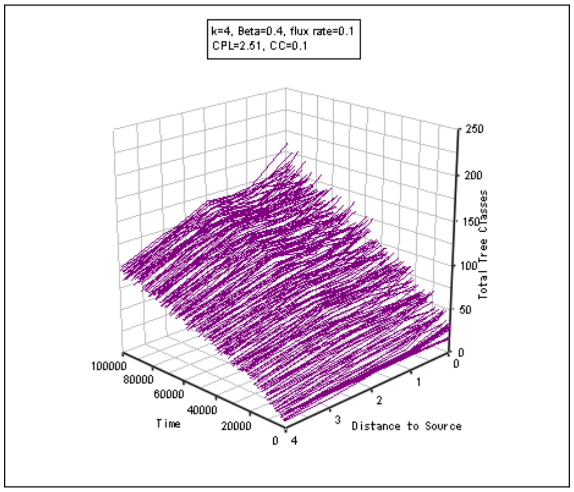


(c)

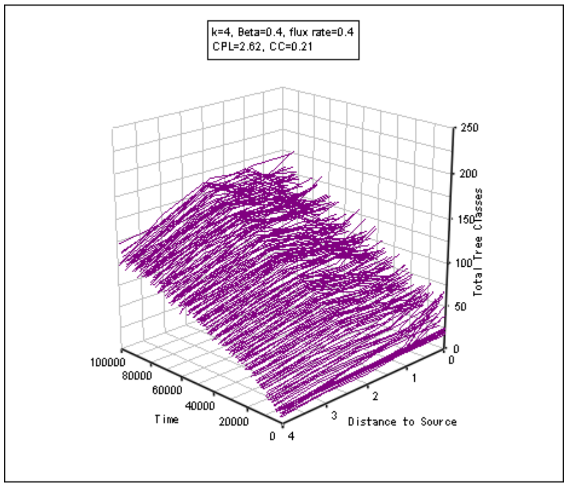


(d)

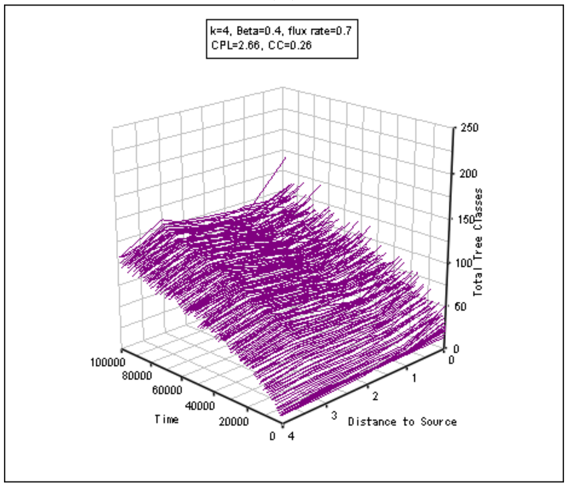
FIGURE 6.6: D_φ in relation to t and d_α, with $k = 4, \beta = 0.1, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



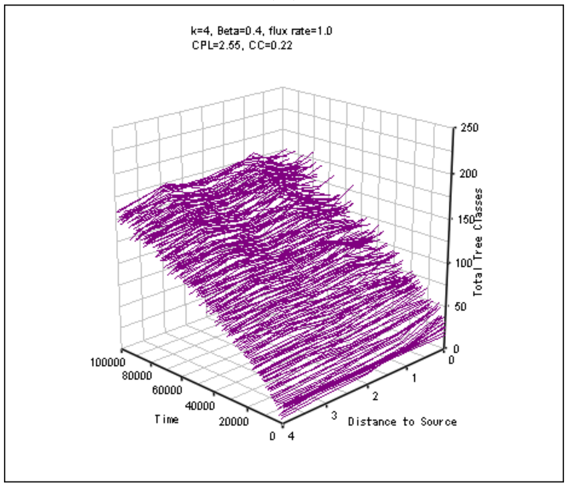
(a)



(b)

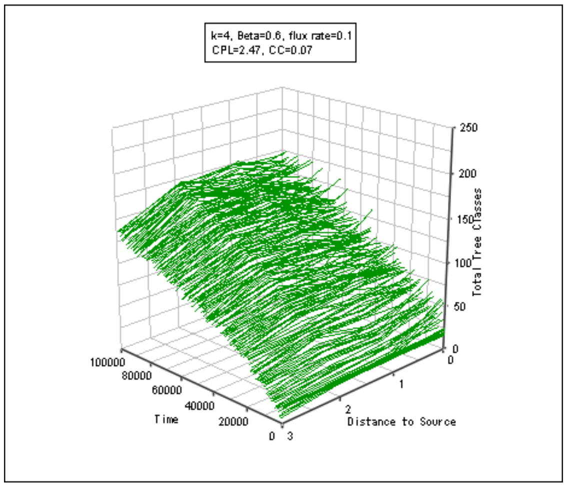


(c)

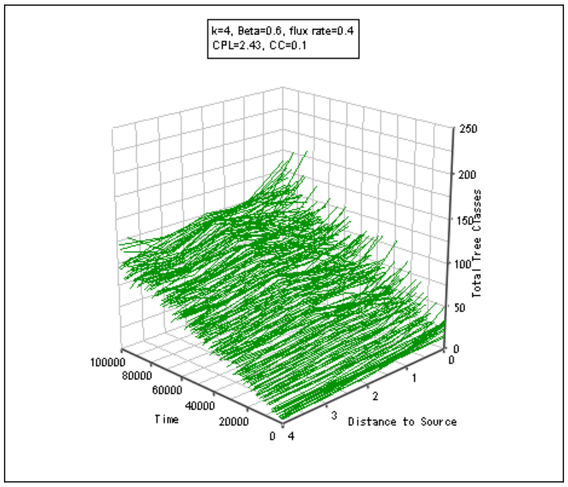


(d)

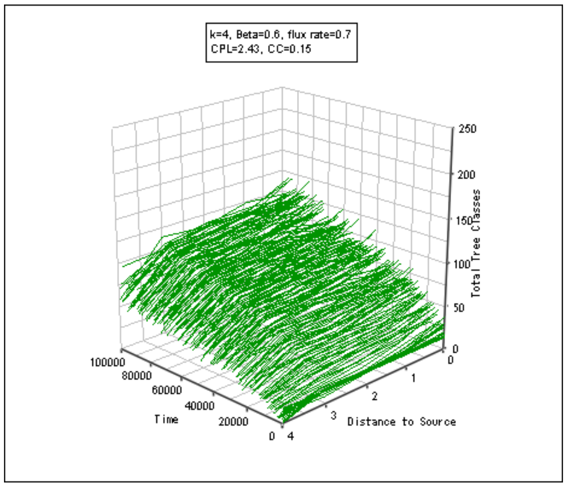
FIGURE 6.7: D_φ in relation to t and d_α, with $k = 4, \beta = 0.4, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



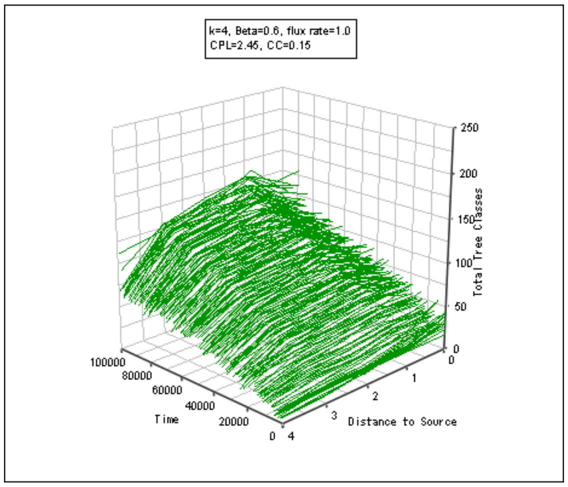
(a)



(b)

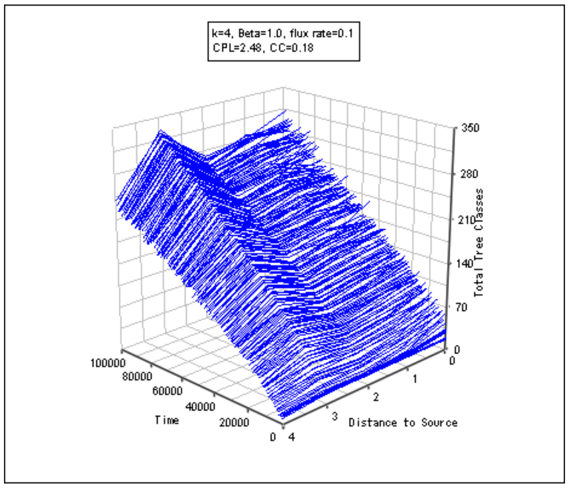


(c)

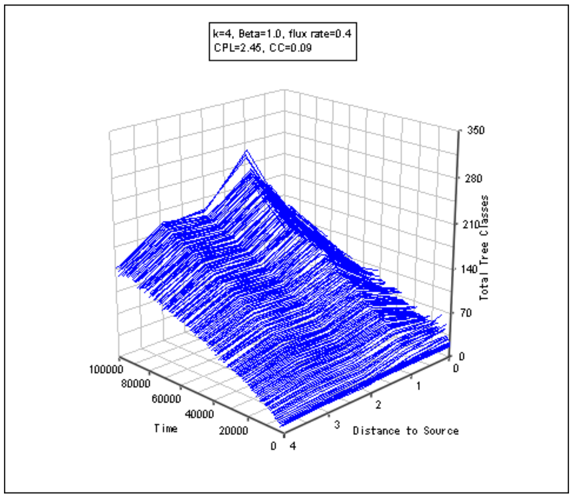


(d)

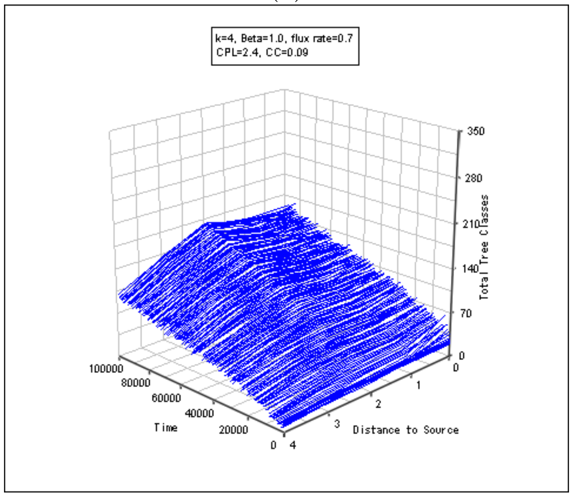
FIGURE 6.8: D_φ in relation to t and d_α, with $k = 4, \beta = 0.6, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



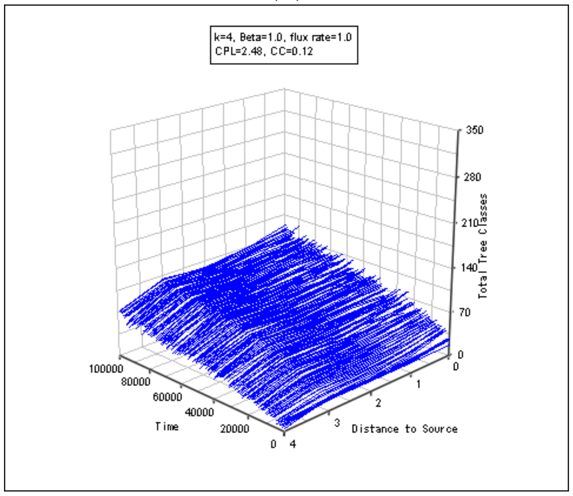
(a)



(b)

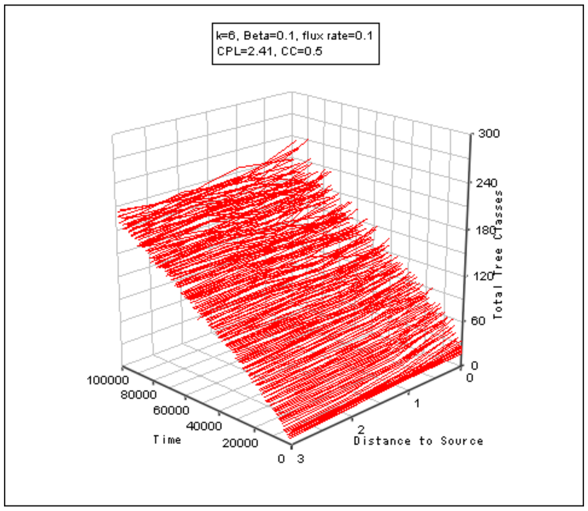


(c)

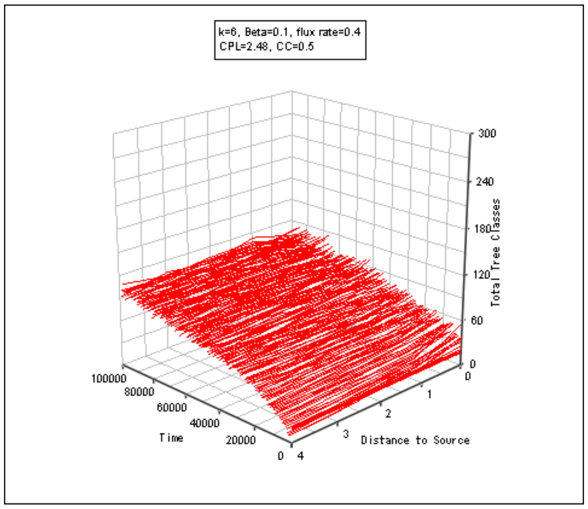


(d)

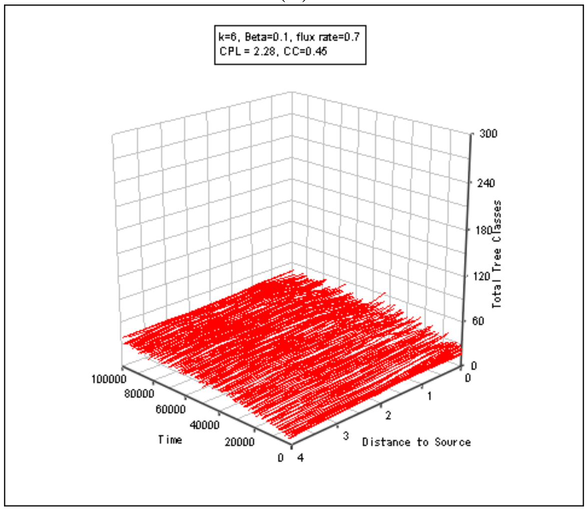
FIGURE 6.9: D_φ in relation to t and d_α, with $k = 4, \beta = 1.0, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



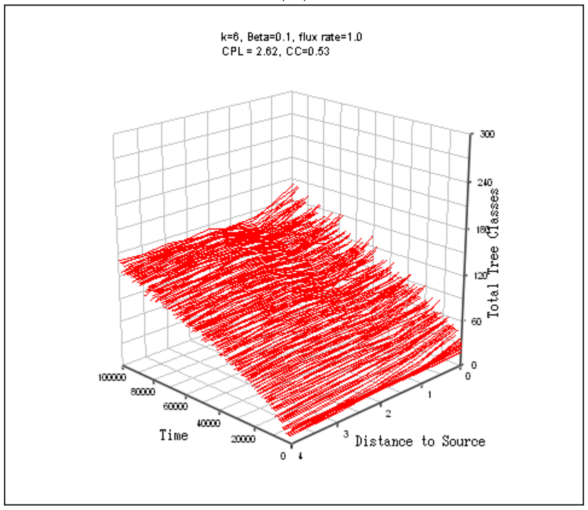
(a)



(b)

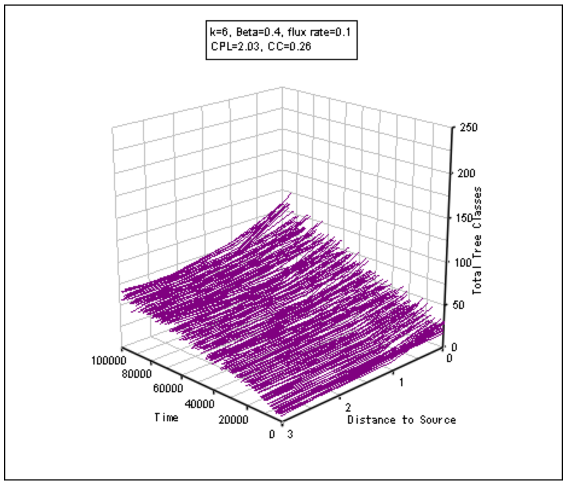


(c)

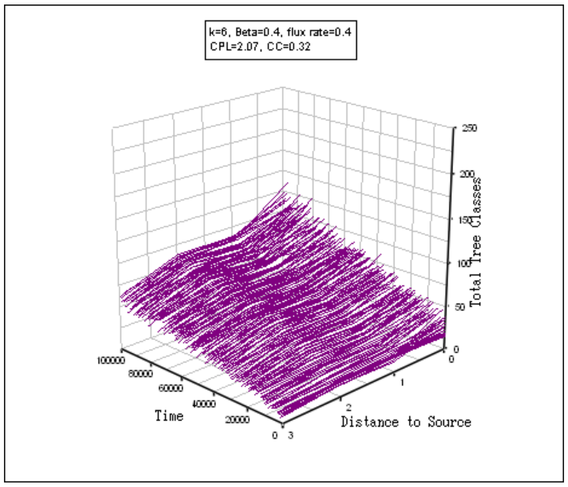


(d)

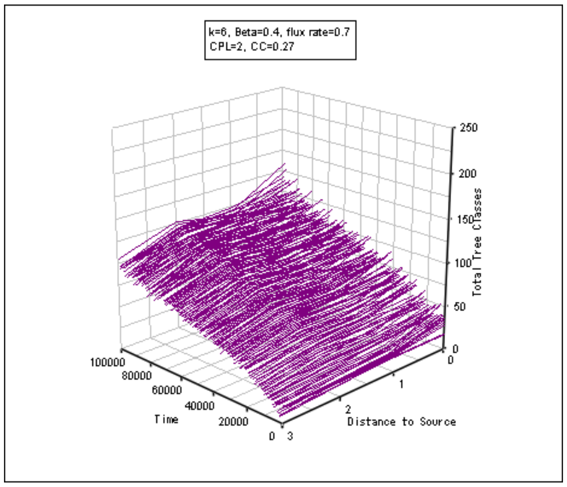
FIGURE 6.10: D_φ in relation to t and d_α, with $k = 6, \beta = 0.1, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



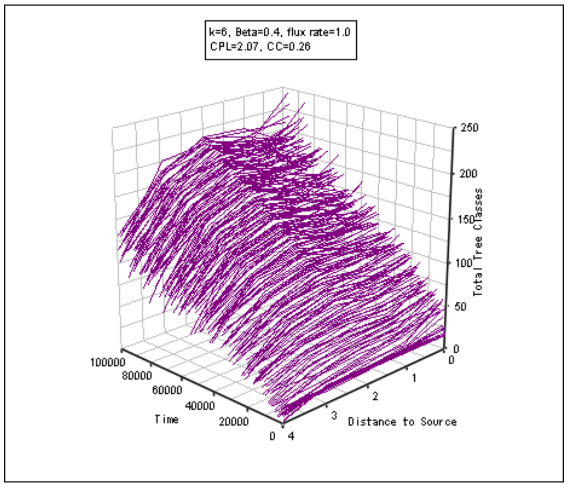
(a)



(b)

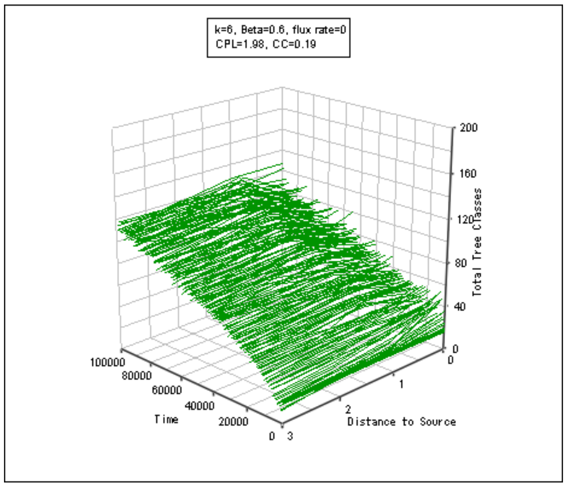


(c)

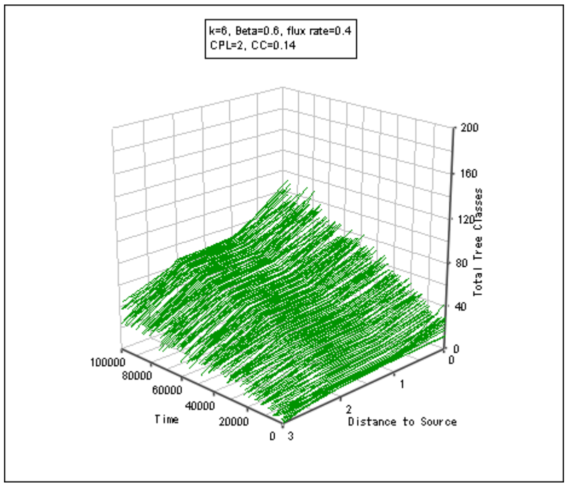


(d)

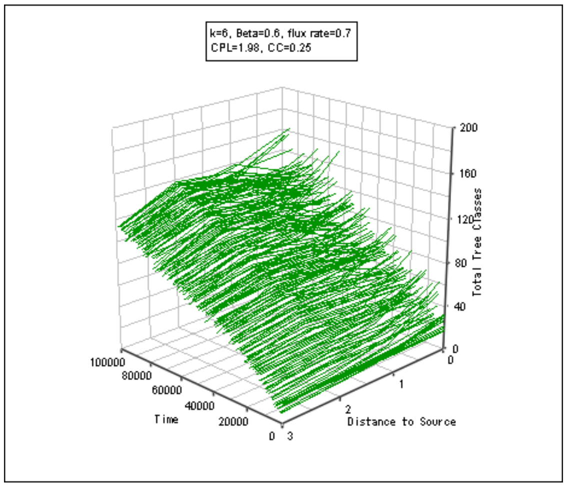
FIGURE 6.11: D_φ in relation to t and d_α, with $k = 6, \beta = 0.4, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



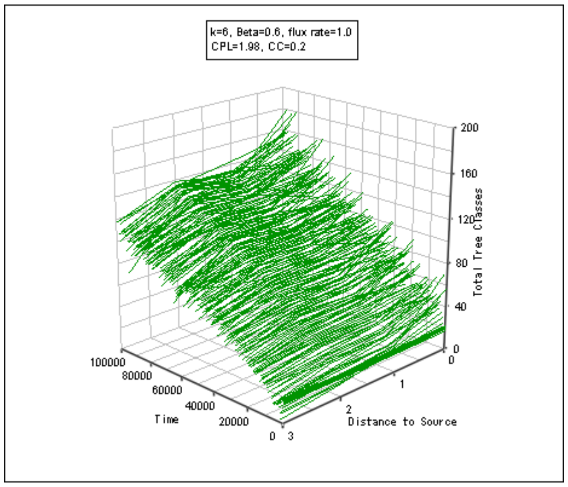
(a)



(b)

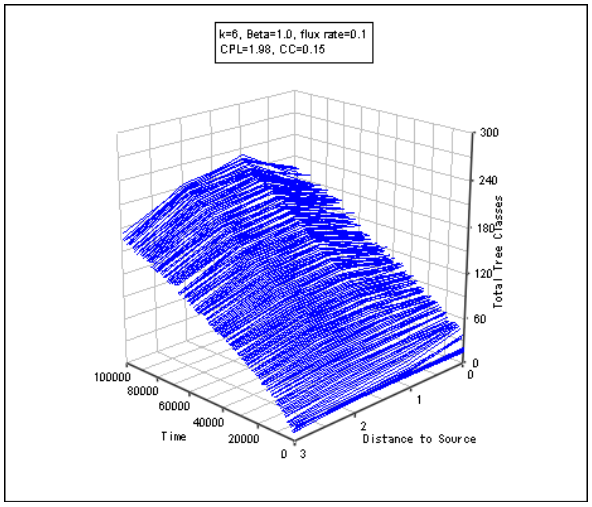


(c)

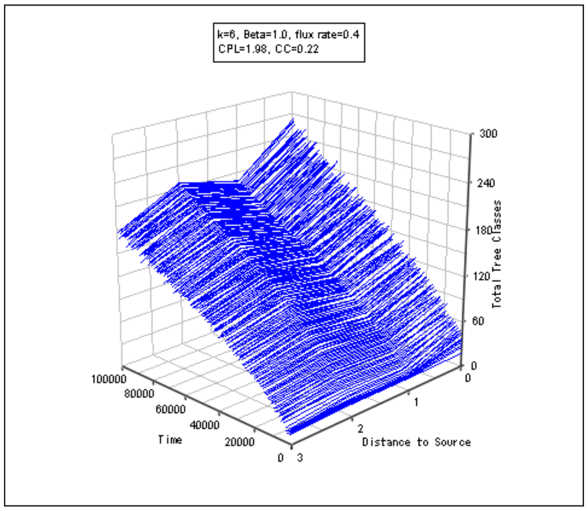


(d)

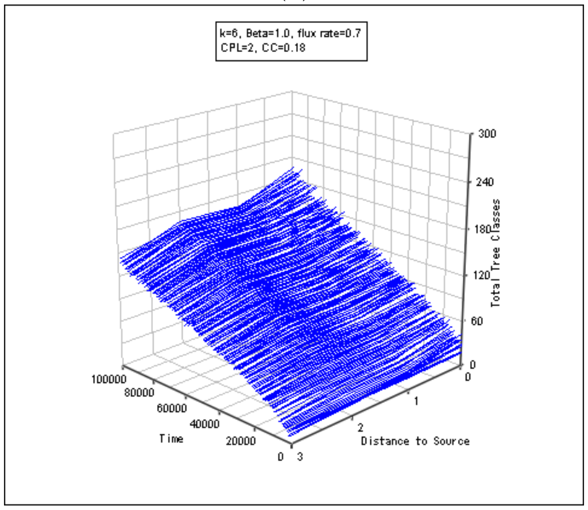
FIGURE 6.12: D_φ in relation to t and d_α, with $k = 6, \beta = 0.6, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$



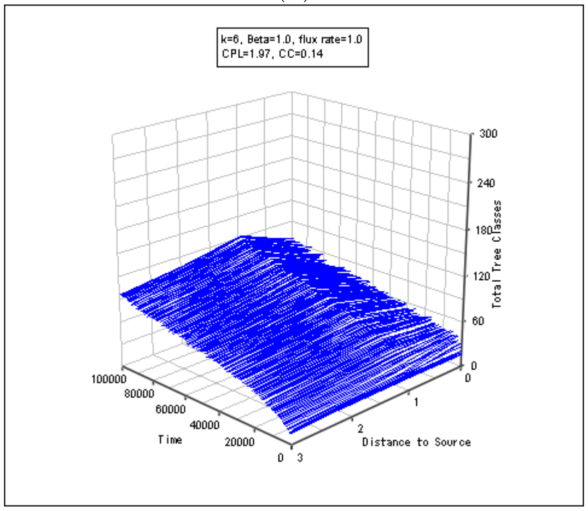
(a)



(b)



(c)



(d)

FIGURE 6.13: D_{φ} in relation to t and d_{α}, with $k = 6, \beta = 1.0, \Phi_{in} \in \{0.1, 0.4, 0.7, 1.0\}$

Comparing across figures from Fig. 6.2 to Fig. 6.13, it can be seen that d_α has a more obvious impact on D_φ with a lower k value. This is because a longer distance between the source and the sink leads to a greater transition of concentration of software components in the network. As has been shown in the previous chapter, a higher k value directly results in a lower CPL value. While the CPL value represents the average path length of the graph, a decrease in the CPL value results in the decrease in distance between the source and the sink.

In addition, d_α influences D_φ in two different patterns. In one case, diversity is higher at the source ($d_\alpha = 0$) and decreases as d_α increases (for example, Fig. 6.3(a),(b), 6.4(a),(b), 6.5(a),(b)). In another case, D_φ increases as the observed compartment is further away from the source until it reaches a local peak and decreases again (for example, Fig. 6.3(c),(d), 6.4(c),(d), 6.5(c),(d)).

Despite the different ways in which D_φ is affected by d_α, some common patterns can be observed. It can be seen that the diversity of assembled trees reaches a minimum at the sink. Moreover, a local peak of D_φ can be observed at d_α which roughly equals the CPL value, and this means the diversity reaches a peak at the compartments located at the middle between the source and the sink in the network. This phenomena is more obvious with a lower k value as the distance between source and sink is greater with a lower k.

Table 6.1 illustrates the number of distinct trees assembled averaged over each compartment for dynamic and static $ASAP^2$ categorized by k [2]. Table 6.1 also shows the average time at which dynamic $ASAP^2$ reaches the level of diversity attained for static

[2]The reason we categorize the average number by k is that k has a more obvious impact than β and Φ_{in} on D_ε, as shown in the previous analysis

$ASAP^2$. It can be seen that while components are continuously added into the system, the total number of distinct program trees generated by dynamic $ASAP^2$ quickly surpasses the static model. And it takes the dynamic system less time to surpass D_ε with a lower k value. As the termination time for the simulation is fixed to 100000 in the open system, the average diversity of assembled programs is higher with a lower k value as shown in Table 6.1.

TABLE 6.1: Open $ASAP^2$ and close $ASAP^2$: comparison of assembled programs diversity

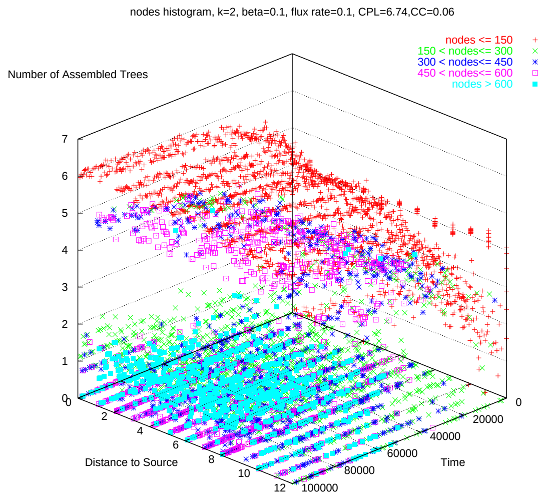
	close system		open system	
	D_ε	t_ε	time that D_φ surpasses D_ε	D_φ at termination
$k = 2$	41.475	89145.75	2350	148.97
$k = 4$	42.541	89347.75	2775	145.28
$k = 6$	43.542	136554	3675	122.26

Analysis of emergent complexity

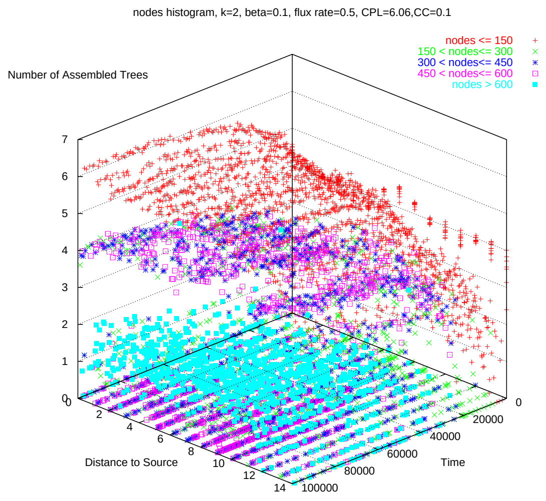
The complexity of a self-assembled program is measured by the number of nodes contained in its tree representation and the height of the tree. We produce a large set of node histograms and height histograms, each showing the relationship between the number of assembled trees versus time and distance to source, grouped by node / height complexity bins under the experimental parameters in $\{\beta, k, \Phi_{in}\}$. Due to the large number of histogram figures with the experimental settings ($4 \times 3 \times 10 \times 2 = 240$), the following figures are a careful selection of histogram figures that we believe best reflect the phenomenon of how the complexity of generated programs is affected by $\{k, \beta, \Phi_{in}\}$. As the height of a tree is logarithmically related to the number of nodes it contains, we present here only the node histogram for the analysis of emergent complexity of self-assembled trees. The experimental figures for the complete range of $\{k, \beta, \Phi_{in}\}$ parameters settings, as well as the tree height histograms can be referenced at: `http://www.cs.nott.ac.uk/~lxl/complexity_figures/home.html`.

Figures 6.14 and 6.15 indicate that the emergent complexity of self-assembled programs is influenced by flux probability in opposite ways under different CPL value of the graph. In a graph (Fig. 6.14) with longer path length, the number of assembled programs increase with a higher influx rate. More complex tree structures are formed as a result. On the other hand, the complexity of generated programs decreases as influx rate increases in a graph with shorter path length. Figure 6.15 indicates that the CPL values of the constructed graphs are significantly lower than those in Fig. 6.14. As a result, it can be seen that n_φ decreases as Φ_{in} increases.

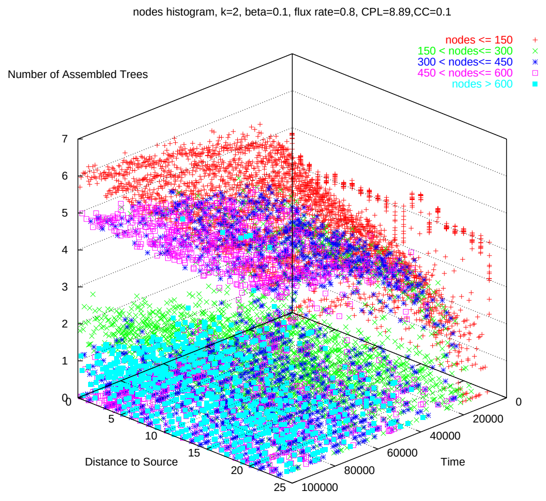
The reason behind the above phenomena is due to the design of the dynamic flow of components. While the probability of introducing a new component in the influx compartment increases, the probability of removing a component from the outflux compartment increases at the same time. Nevertheless, no components will be removed if there are no components of the same size in the outflux compartment, thus the distance between sink and source plays an important role in in the outflux rate of components. As the sink compartment is chosen to have the longest path length to the source compartment, the distance between sink and source compartments are directly proportional to the average path length of the graph. Thus, under the same influx rate, the outflux rate is inversely proportional to the CPL of the graph. The outflux rate is not significantly influenced by the influx rate in a graph with greater average length than in a graph with a shorter average path length.



(a) $\beta = 0.1, \Phi_{in} = 0.1$

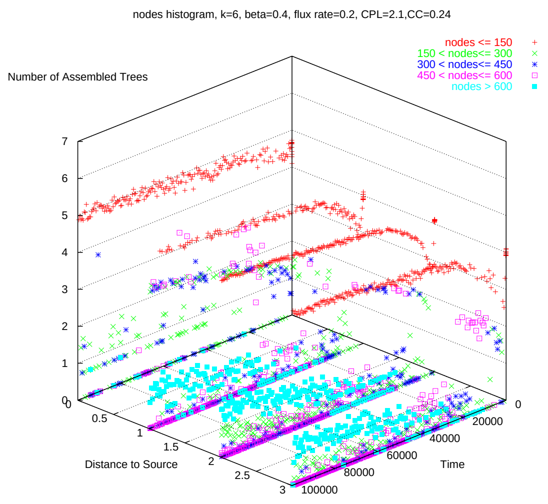


(b) $\beta = 0.1, \Phi_{in} = 0.5$

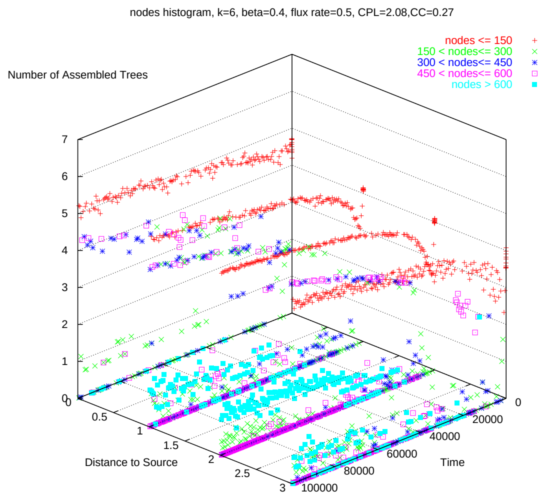


(c) $\beta = 0.1, \Phi_{in} = 0.8$

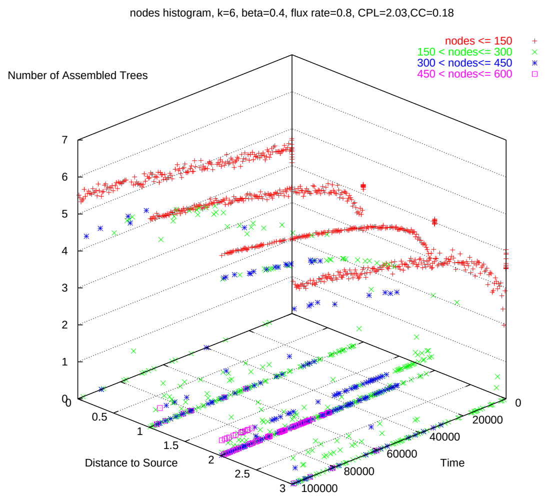
FIGURE 6.14: Node histogram for $ASAP^2$ far from equilibrium



(a) $\beta = 0.4, \Phi_{in} = 0.2$

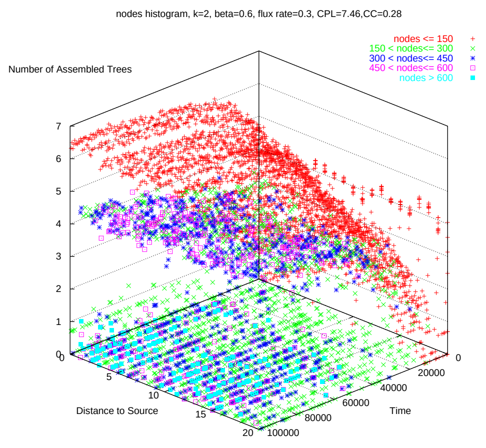


(b) $\beta = 0.4, \Phi_{in} = 0.5$

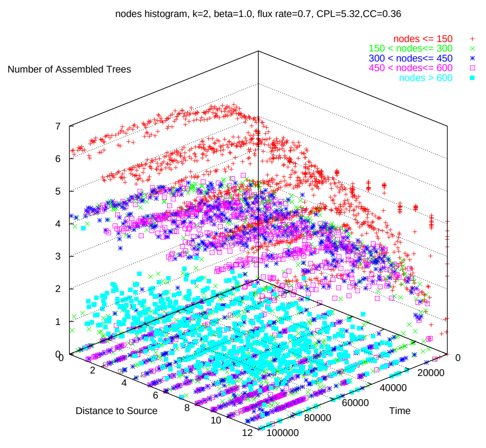


(c) $\beta = 0.1, \Phi_{in} = 0.8$

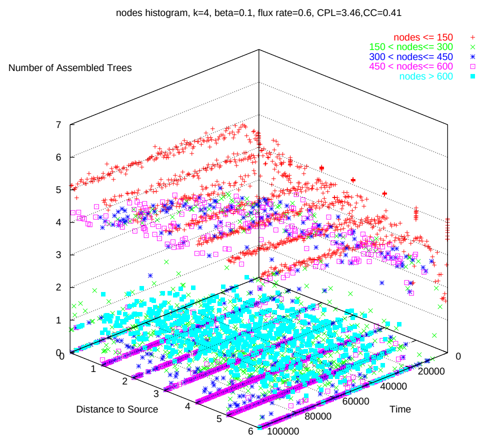
FIGURE 6.15: Node histogram for $ASAP^2$ far from equilibrium



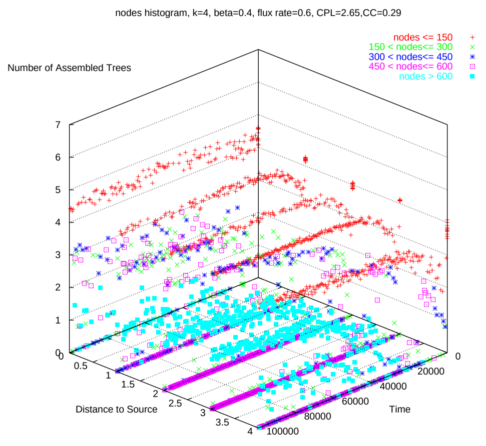
(a) CPL=7.46



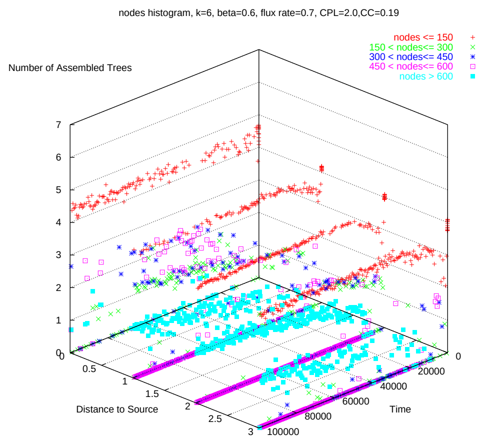
(b) CPL=5.32



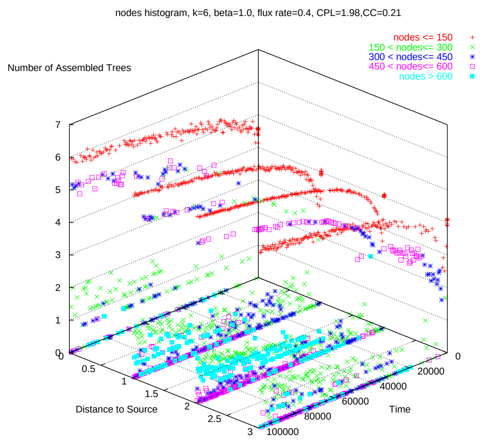
(c) CPL=3.46



(d) CPL=2.65



(e) CPL=2.43



CPL =1.98

FIGURE 6.16: Node histogram for $ASAP^2$ far from equilibrium

Hence, the complexity of self-assembled programs increase while the influx rate increases in a graph with greater path length (CPL>6). On the contrary, the complexity of generated programs decreases with an increase in influx rate in a graph with shorter path length (CPL<3).

Figure 6.16 shows a general phenomenon that the length property of the graph plays a significant role in the complexity of the generated programs. As can be seen, more variety of trees can be observed with a larger CPL. Table 6.2 concludes our major findings on diversity and complexity of assembled programs for open $ASAP^2$.

Table 6.2: Conclusions of findings on open $ASAP^2$

	diversity	**complexity**
time	increases over time	increases over time
CPL	increases with larger CPL	increases with larger CPL
Φ_{in}	no obvious impacts	increases with higher influx rate if CPL>6; decreases with higher influx rate if CPL<3
d_α	a local peak of diversity can be observed at d_α=CPL	no obvious impacts

6.4 Summary

In previous chapters, we systematically and intensively studied software self-assembly close to equilibrium using various embodiments and a wide range of external environment parameters. The static $ASAP^2$ we presented in previous chapters can be seen as a close software self-assembly system. In this chapter, we studied the impact of open network topology on software self-assembly. The extended $ASAP^2$ model imposes a constant flow of software components by introducing source and sink compartments on the network $ASAP^2$ [57]. Based on an external specified influx rate, software components are added into the source compartment and are taken out from the sink compartment, thus introducing a dynamic flow of components and different concentration across the network. As a design choice, the influx rate is always higher or equal to the outflux rate, hence the extended $ASAP^2$ is *far from* equilibrium.

We investigated how graph properties as well as influx rate affect the diversity and complexity of generated programs. In general, the diversity and complexity of generated structures increases as software self-assembly progresses over time. Moreover, it has been shown that the average diversity and complexity of assembled programs is higher with a lower k value.

Experimental results also indicate that compartment locations in relation to source in the network plays an important role in diversity of assembled programs. One interesting finding is that a local peak of program diversity can be observed at d_α roughly equaling that in CPL, which suggests it is more likely to find greater diversity of assembled programs in those compartments that are located in the middle between the source and the sink.

In addition, the influx rate affects the complexity of assembled programs in different ways depending on the length property of the graph. In one case with longer graph path length (CPL>6), the emergent complexity of assembled programs (both simple and complex) increases as the influx rate rises. On the contrary, in a graph with shorter path length (CPL<3), the emergent complexity decreases when the influx rate rises.

Chapter 7

Conclusions and Future Work

This chapter summarizes this thesis by reviewing the main contributions and conclusions that can be learned, as well as a selection of future work that can be extended from this work.

7.1 Conclusions

In this thesis, I presented software self-assembly as a new approach to automatic programming. This first research attempt on software self-assembly gives a new vision on automatic program generation. In this preliminary stage of research on software self-assembly, I investigated the development of software self-assembly. I presented the Automated Self-Assembly Programming Paradigm ($ASAP^2$), a software self-assembly model in which software components autonomously integrate into programs. I examined how various embodiments affect the process and results of software self-assembly under what condition software self-assembly yields more diversified programs.

In chapter 2, we analyzed various self-assembling system features, and reviewed a selection of self-assembly systems applied as an engineering discipline in various domains, thus giving an overview of features, advantages and limitations of self-assembly as a design methodology.

In addition, we briefly reviewed popular current automatic programming approaches, i.e. Genetic Programming and Grammatical Evolution. While Genetic Programming evolves both the structure and the content of the parse trees simultaneously, software self-assembly is aimed at reusing the *content* of previously built software components by self-assembling a suitable *structure*. Unlike GP or GE, that uses an evolutionary metaphor to guide the process of automatic programming, software self-assembly relies on natural metaphors of self-assembly for the exploration of programs space, thus we expect to have very different kinds of limitations and potentials. The literature review concluded by a detailed explanation on our research motivation and research scope on this thesis.

In chapter 3, we introduced "program gases", a metaphor for software self-assembly system in which software components perform a random walk. We observed the similarities between an unguided software self-assembly process and an ideal gas environment. However, unlike pure collisions in ideal gas, in program gas software components will sometimes self-assemble to one aggregate structure as a result of their interactions.

We observed the dynamics of $ASAP^2$ using the kinetic theory of ideal gas as a crude approximation on program gas. The same set of external environment parameters as in ideal gas (i.e. Area, Temperature, Number of components) were introduced to assess how program gas differs from ideal gas. In order to investigate how this particular implementation of $ASAP^2$ explores the space of all program architectures, we measured how pressure, time to equilibrium and diversity of assembled structures is affected by the environment parameters at equilibrium. Experimental results indicate an interesting finding that with unguided dynamics, an increase in the number of copies of components lead to a shorter time to equilibrium with program gas. Temperature and area only affect time to equilibrium and not diversity of self-assembled programs.

In addition, mathematical formulae were presented to predict time to equilibrium and diversity of assembled programs. That is, we can interpolate the time needed for $ASAP^2$ to reach its equilibrium and the resulting diversity of the assembled populations for a given set of environment parameter (A, T, N).

To investigate software self-assembly using a different embodiment, a different $ASAP^2$ model inspired from particle swarm optimization is presented in Chapter 4. Leader components were introduced in software self-assembly and software components follow leaders that exhibit the greatest attractive force. As a result of these changed dynamics, an emergent dynamic neighbourhood is formed. We compared the program gas and PSO inspired implementation and experimental results by illustrating time to equilibrium and diversity of generated programs under the same set of environment parameters (A, T, N). Although an improvement can be made on the average time to equilibrium, a decrease in diversity of the generated population is observed as a result of deploying the PSO inspired approach. The comparisons between the two implementations is also assessed by the prediction model presented in Chapter 3. Results have shown greater prediction errors as the behaviour of the system has been altered when leaders are introduced, which again proves the extended $ASAP^2$ system behaves differently from the original program gas implementation.

In chapter 5, we examined the impact of network topology on software self-assembly by presenting a diversified compartment approach for $ASAP^2$. This approach introduces a network of compartments embedded in a graph and software self-assembly takes place simultaneously in each compartment. Components can migrate into another compartment via a connecting edge in the graph.

We examined how length and clustering property of a graph (measured by CPL and CC) and the amount of software components involved can affect software self-assembly within the network in terms of time to equilibrium, diversity and complexity of the generated programs. Experimental results have indicated that on average a higher CPL of the graph leads to a lower time to equilibrium. It can also be concluded from experimental results that given two *ordered* graphs, a sparsely connected graph will take longer to reach equilibrium and result in greater diversity of assembled programs than a densely connected graph. On the contrary, given two *random* graphs, a sparsely connected network results in shorter time to equilibrium and lower diversity of assembled programs than a densely connected graph. Last but not least, a sparsely connected graph yields less complex programs than a densely one.

Chapter 3, 4 and 5 present a complete study on *static* software self-assembly, i.e. a closed system that will eventually reach equilibrium. Chapter 6 focuses on the investigation of open software self-assembly system *far* from equilibrium. The dynamic $ASAP^2$ is based on the static network $ASAP^2$model with a possibility of adding and removing software components from the sink and source compartment. Thus, this extended system imposes a constant flow of software components resulting in a different concentration of components in different parts of the network.

It can be concluded that the length property of the graph plays the most important role in diversity and the emergent complexity of the assembled structures in dynamic $ASAP^2$. First of all, the influx rate affects the complexity of assembled programs in different ways depending on the length property of the graph. In one case with longer graph path length (CPL>6), the emergent complexity of assembled programs (both simple and complex) increases as the influx rate rises. On the contrary, in a graph with shorter path length (CPL<3), the emergent complexity decreases when the influx rate rises. Secondly, a local peak can be observed at the compartments located in the middle between the sink and source compartments. Last but not least, a lower k value (which leads to a higher CPL value) results in a greater diversity of assembled programs.

This first research on software self-assembly studied how its behaviour and results are affected by using a range of approaches and embodiments. This covers unguided to PSO inspired approach, software self-assembly in a single compartment to a network of compartments, and finally software self-assembly in an open environment.

In the long run, software self-assembly is expected to complement current automatic programming methodologies such as Genetic Programming and Grammatical Evolution, as well as aid the understanding on advantageous properties such as self-healing or self-reconfiguring exhibited in many natural and artificial self-assembly systems.

7.2 Future work

There are several directions of future work that can be followed from this dissertation.

The diversified compartment approach introduced in Chapter 5 is used to represent the "Internet" and investigates the network topological impacts on $ASAP^2$, in our experiments with the diversified compartment approach, only unweighted graphs are considered, meaning the connection probability and speed are unbiased between any pair of compartments connected by an edge. One possible future work for this research is to investigate in greater detail how computer network factors impact the system and the generated programs. One network factor worth investigating on is the bandwidth. This can be realized by implementing a queue on each edge holding software components migrating from one compartment to another. Instead of using an unweighted graph as in Chapter 5 and 6, a weighted graph can be used to represent the bandwidth of connections between two nodes (computers). Therefore, the speed of removing components from the queue can be proportional to the weight on its edge.

In our experiments with $ASAP^2$, we covered software self-assembly in an open and closed environment (i.e. dynamic and static software self-assembly) with the network approach. It has been shown in this thesis that the behaviour and results are dramatically different between the open and close environments. To study how the transition occurs, another possible future avenue is to study an intermediate software self-assembly system, i.e. a "semi"-dynamic system which can be close to or far from equilibrium depending on the requirement. This can be realized by introducing mutation rates. At any time step, a mutation of software component can occur during a transfer between compartments. This mutation may or may not change the component type. Thus, the system can be easily tuned towards far from or close to equilibrium with this mutation parameter. That is, if the mutation rate is high and if it constantly changes the type of the software component, the system is far from equilibrium. However, if the mutation rate is low and the mutation of the component does not involve a type change, the system will eventually reach its equilibrium.

In addition, further research can be proposed to investigate how gradient of concentrations of software components affects software self-assembly. One possible approach is to introduce nested compartments in which information between compartments can flow to the inner/outer world through its border. Each compartment can have a different migration probability based on its size.

Although the swarm based approach introduced in chapter 4 may greatly improve the performance of the self-assembly process, it is unlikely that this alone will be able to overcome the combinatorial nature of programs' space exploration. Instead, more "intelligent" self-assembly should be investigated. Possible solutions may be based on graph grammars for directed self-assembly [46, 47] or the use of a tabu list [28] to ban the bindings between certain components if the binding product is unlikely to be of use.

Finally, future research on software self-assembly could aid the understanding of self-healing [27] and self-reconfigurable [86] software. Software self-assembly is expected to inherit some, if not all, of the advantages of the self-assembly systems reviewed in the previous section.

References

[1] R. J. Abbott. Object-oriented genetic programming, an initial implementation. In *Proceedings of the Sixth International Conference on Computational Intelligence and Natural Computing*, Embassy Suites Hotel and Conference Center, Cary, North Carolina USA, Sep 26-30 2003.

[2] M. Abramson and L. Hunter. Classification using cultural co-evolution and genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 249–254, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[3] L. Adleman, Q. Cheng, A. Goel, M. D. Huang, D. Kempe, P. Moisset de Espans, and P. Rothemund. Combinatorial optimization problems in self-assembly. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on theory of computing*, pages 23–32, New York, NY, USA, 2002. ACM Press.

[4] G. Adorni, S. Cagnoni, and M. Mordonini. Genetic programming of a goalkeeper control strategy for the robocup middle size competition. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming,*

Proceedings of EuroGP'99, volume 1598 of *LNCS*, pages 109–119, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[5] N. Badr, D. Reilly, and A. Taleb-Bendiab. A conflict resolution control architecture for self-adaptive software. In *Proceedings of the International Workshop on Architecting Dependable Systems*, Florida, USA, 2002.

[6] E. Boczko and C. Brooks. First-principles calculation of the folding free energy of a three-helix bundle protein. *Science*, 222:393–396, 1995.

[7] H. Bojinov, A. Casal, and T. Hogg. Emergent structures in modular self-reconfigurable robots. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 1734–1741, San Francisco, California, USA, 2000.

[8] S. Bornhofen and C. Lattaud. Outlines of artificial life: A brief history of evolutionary individual based models. In *Artificial Evolution*, pages 226–237, 2005.

[9] S. G. Brush and N. S. Hall. *The Kinetic Theory of Gases: An Anthology of Classic Papers with Historical Commentary (History of Modern Physical Sciences)*. Imperial College Press, 2003.

[10] E. Burke, S. Gustafson, and G. Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[11] E. Burke, S. Gustafson, G. Kendall, and N. Krasnogor. Advanced population diversity measures in genetic programming. In *7th International Conference Parallel Problem*

Solving from Nature, volume 2439 of *Springer Lecture Notes in Computer Science*, pages 341–350, Granada, Spain, September 2002. PPSN, Springer Berlin / Heidelberg. ISBN 3-540-44139-5.

[12] E. Burke, S. Gustafson, G. Kendall, and N. Krasnogor. Is increased diversity beneficial in genetic programming: An analysis of the effects on fitness. In *IEEE Congress on Evolutionary Computation*, pages 1398–1405. CEC, IEEE, December 2003.

[13] G. M. Clarke and D. Cooke. *A Basic Course in Statistics.* Arnold, 1998.

[14] C. A. Coello and M. S. Lechuga. A proposal for multiple objective particle swarm optimization. In *Proceedings of the Congress on Evolutionary Computation (CEC'2002)*, pages 1051–1056, Piscataway, New Jersey, 2002.

[15] National Research Council Committee to Review the National Nanotechnology Initiative. *A Matter of Size: Triennial Review of the National Nanotechnology Initiative.* Washington, DC: National Academies Press, 2006.

[16] C. de Duve. *Blueprint for a Cell: The Nature and the Origin of Life.* Neil Patterson Publishers, 1991.

[17] M. Dorigo, A. Coloni, and V. Maniezzo. Distributed optimisation by ant colonies. In *Proceedings of First European Conference on Artificial Life*, pages 134–142, 1991.

[18] F. J. Dyson. *Origins of Life.* Cambridge University Press, 1985.

[19] R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Pro-*

ceedings of the Sixth International Symposium on Micro Machine and Human Science, pages 39–43, Nagoya, Japan, IEEE Service Center, Piscataway, NJ, 1995.

[20] R. C. Eberhart and J. Kennedy. Particle swarm optimisation. In *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[21] R. C. Eberhart, J. Kennedy, and Y. Shi. *Swarm Intelligence.* Morgan Kaufmann Publishers, San Francisco, 2001.

[22] A. Ekárt and S. Z. Németh. Maintaining the diversity of genetic programs. In *EuroGP '02: Proceedings of the 5th European Conference on Genetic Programming*, pages 162–171, London, UK, 2002. Springer-Verlag.

[23] RJ Ellis and SM Vandervies. Molecular chaperones. *Annual Review of Biochemistry*, pages 321–347, 1991.

[24] Y. A. Eracar and M. M. Kokar. An architecture for software that adapts to changes in requirements. *J. Syst. Softw.*, 50(3):209–219, 2000.

[25] A. M. Farley. Population structure and artificial evolution. In *Proceedings of the Seventh International Conference on Artificial Evolution*, pages 213–225. Berlin:Springer-Verlag, LNCS 3871, 2006.

[26] H. Frank and W. Chou. Topological optimization of computer networks. *Proceedings of the IEEE*, 60(11):1385–1397, 1972.

[27] S. George, D. Evans, and L. Davidson. A biologically inspired programming model

for self-healing systems. In *Workshop on Self-Healing Systems, Proceedings of the first workshop on Self-Healing systems*, pages 102–104. ACM Press, 2002.

[28] F. Glover, E. Taillard, and D. de Werra. A user's guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.

[29] O. Görnerup and J. P. Crutchfield. Hierarchical self-organization in the finitary process soup. In *Tenth International Conference on the Simulation and synthesis of Living Systems (ALIFE X)*, pages 75–80, 2006.

[30] S. Gustafson. *An Analysis of Diversity in Genetic Programming.* PhD thesis, University of Nottingham, 2004.

[31] S. Gustafson, E. Burke, and N. Krasnogor. The tree-string problem: An artificial domain for structure and content search. In *8th European Conference on Genetic Programming*, volume 3447 of *Springer Lecture Notes in Computer Science*, Lausanne, Switzerland, March, April 2005. EuroGP, Springer Berlin Heidelberg. ISBN 978-3-540-25436-2.

[32] S. Gustafson, A. Ekart, E. Burke, and G. Kendall. Problem difficulty and code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 5(3):271–290, September 2004.

[33] J. B. Pollack H. Lipson. Automatic design and manufacture of artificial life forms. *Nature*, 406:974–978, 2000.

[34] S. Hackwood and S. Beni. Self-organization of sensors for swarm intelligence. In *IEEE*

International Conference on Robotics and Automation, pages 819–829, Nice, France, 1992.

[35] W. R. Hargreaves and D. W. Deamer. Liposomes from ionic, single-chain amphiphiles. In *Biochemistry*, volume 17, pages 3759–3768, 1978.

[36] W.R. Hargreaves, S. Mulvihill, and D.W. Deamer. Synthesis of phospholipids and membranes in prebiotic conditions. In *Nature*, volume 266, pages 78–80, 1977.

[37] T. Hogg. Robust self-assembly using highly designable structures. *Papers from the Sixth Foresight Conference on Nanotechnology*, 10:300–307, 1999.

[38] X. Hu and R. C. Eberhart. Multi-objective optimization using dynamic neighbourhood particle swarm optimization. In *Proceeding of the 2002 Congress on Evolutionary Computation*, Honolulu, Hawaii, USA, 2002.

[39] M. Hutter. Fitness uniform selection to preserve genetic diversity. In *Proceedings of the Congress on Evolutionary Computation*, pages 783–788, Manno(Lugano), CH, 2001.

[40] H. Huzita and B. Scimemi. The algebra of paper-folding. *First International Meeting of Origami Science and Technology*, 1989.

[41] C. Igel and K. Chellapilla. Investigating the influence of depth and degree of genotypic change on fitness in genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1061–1068, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[42] T. Imae, O. Mori, K. Takagi, M. Itoh, and Y. Sawaki. Self-assembly formation of amphiphilic molecules mixed with photoreactive, aromatic unsaturated-acids: examination by light scattering. In *Colloid & Polymer Science*. Springer Berlin / Heidelberg, 2004.

[43] C. Jones and M. J. Mataric. From local to global behavior in intelligent self-assembly. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 721–726, 2003.

[44] A. V. Kabanov, P. L. Felgner, and L. W. Seymour. *Self-assembling Complexes for Gene Delivery: From Laboratory to Clinical Trial*. Willey, 1998.

[45] M. Keijzer. Efficiently representing populations in genetic programming. *Advances in genetic programming*, 2:259–278, 1996.

[46] E. Klavins. Directed self-assembly using graph grammars. In *Foundations of Nanoscience: Self Assembled Architectures and Devices*, Snowbird, UT, 2004. Invited Paper. This online version has some corrections.

[47] E. Klavins, R. Ghrist, and D. Lipsky. Graph grammars for self-assembling robotic systems. In *Proceedings of the International Conference on Robotics and Automation*, 2004.

[48] M. M. Kokar, K. Baclawski, and Y. A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems and their Applications*, pages 37–45, 1999.

[49] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[50] J. R. Koza, L. W. Jones, M. A. Keane, and M. J. Streeter. Towards industrial strength automated design of analog electrical circuits by means of genetic programming. *In U.-M. O'Reilly et al., editors, Genetic Programming Theory and Practice II*, 13-15 May 2004.

[51] N. Krasnogor, W. E. Hart, J. Smith, and D. Pelta. Protein structure prediction with evolutionary algorithms. In *International Genetic and Evolutionary Computation Conference (GECCO99)*, pages 1569–1601. Morgan Kaufmann, 1999.

[52] N. Krasnogor, D. H. Marcos, D. Pelta, and W. A. Risi. Protein structure prediction as a complex adaptive system. In Cezary Janikow, editor, *Frontiers in Evolutionary Algorithms*, pages 441–447, 1998.

[53] N. Krasnogor, G. Terrazas, D. A. Pelta, and G. Ochoa. A critical view of the evolutionary design of self-assembling systems. *Proceedings of the 7th International Conference on Artificial Evolution*, Oct 2005.

[54] T. H. LaBean, E. Winfree, and J. H. Reif. Experimental progress in computation by self-assembly of DNA tilings. In Erik Winfree and David K. Gifford, editors, *Proceedings 5th DIMACS Workshop on DNA Based Computers, held at the Massachusetts Institute of Technology, Cambridge, MA, USA June 14 - June 15, 1999*, pages 123–140. American Mathematical Society, 1999.

[55] C. G. Langton. Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D: Nonlinear Phenomena*, 42(1-3):12–37, 1990.

[56] H. Li, R. Helling, C. Tang, and N. Wingreen. Emergence of preferred structures in a simple model of protein folding. *Science*, 273:666–669, 1996.

[57] L. Li, N. Krasnogor, and J. Garibaldi. Automated self-assembly programming paradigm: The impact of network topology. *Special Issue on Nature Inspired Cooperative Strategies for Optimization in the International Journal of Intelligent Systems*, in press, 2007.

[58] L. Li, N. Krasnogor, and J. M. Garibaldi. Automated self-assembly programming paradigm: A particle swarm realization. In *Proceedings of the Workshop on Nature Inspired Cooperative Strategies for Optimization*, pages 123–134, Granada, Spain, 2006. University of Granada.

[59] L. Li, N. Krasnogor, and J. M. Garibaldi. Automated self-assembly programming paradigm: Initial investigations. In *Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems*, pages 25–34, Potsdamn, Germany, 2006. IEEE Computer Society.

[60] L. Li, P. Siepmann, J. Smaldon, G. Terrazas, and N. Krasnogor. *Systems Self-Assembly: Multi-Disciplinary Snapshops*, chapter Automated Self-Assembling Programming, page (to appear). Studies in Multidisciplinarity. Elsevier, 2007.

[61] S. Lin. Effective use of heuristic algorithms in network design. In *The Mathematics of*

Networks, Proceedings of Symposia in Applied Mathematics, volume 26, pages 63–84. Providence, R.I.: Amercian Mathematical Society, 1982.

[62] J. Lohn, G. Hornby, and D. Linden. An evolved antenna for deployment on nasa's space technology 5 mission. *In U.-M. O-Reilly et al., editors, Genetic Programming Theory and Practice II*, 13-15 May 2004.

[63] P. L. Luisi. *The Emergence of Life.* Cambridge University Press, 2006.

[64] M. Lynch and J. S. Conery. The origins of genome complexity. *Science*, 302:1401–1404, 2003.

[65] M. D. Mcilroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proceedings of the NATO Conference on Software Engineering*, pages 138–155, Garmish, Germany, October 1968. NATO Science Committee.

[66] M. Mitchell. Computation in cellular automata: A selected review. *Non-standard Computation*, pages 385–390, 1996.

[67] H. J. Morowitz. *Beginning of Cellular Life.* Yale University Press, 1992.

[68] R. Nagpal, A. Kondacs, and C. Chang. Programming methodology for biologically-inspired self-assembling systems. *AAAI Spring Symposium on Computational Synthesis*, March 2003.

[69] H. Noguchi and M. Takasu. Self-assembly of amphiphiles into vesicles: A brownian dynamics simulation. *Phys. Rev. E*, 64(4):041913, Sep 2001.

[70] M. O'neill and A. Brabazon. Grammatical swarm: The generation of programs by social programming. *Natural Computing*, 5(4):443–462, November 2006.

[71] M. O'Neill, A. Brabazon, and C. Adley. The automatic generation of programs for classification problems with grammatical swarm. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 104–110, Portland, Oregon, 20-23 June 2004. IEEE Press.

[72] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Trans. Evolutionary Computation*, 5(4):349–358, 2001.

[73] A. I. Oparin. *The Origin of Life On Earth.* Academic Press, 1957.

[74] U. O'Reilly. Using a distance metric on genetic programs to understand genetic operators. In *Proceedings of 1997 IEEE International Conference on Systems, Man, and Cybernetics*, pages 4092–4097, 1997.

[75] T. Ray and K. M. Liew. A swarm metaphor for multi-objective design optimization. *Engineering Optimization*, 34(2):141–153, 2002.

[76] O. Rössler. Recursive evolution. *Biosystems*, 11:193–199, 1979.

[77] D. H. Rouvray. Predicting chemistry from topology. *Scientific American*, 255(3):40–47, Sep 1986.

[78] C. Ryan. Book review: Genetic programming 3: Darwinian invention and problem solving. *Genetic Programming and Evolvable Machines*, 1(4), October 2000.

[79] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris, 14-15 1998. Springer-Verlag.

[80] K. Saitou. Conformational switching in self-assembling mechanical systems. *IEEE Transactions on Robotics and Automation*, 15:510–520, 1999.

[81] A. Sali, S. A. Shakhnowich, and M. Karplus. How does a protein fold? *Nature*, 369:248–251, 1997.

[82] T. Schank and D. Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9(2):265–275, 2005.

[83] F. Schweitzer and B. Tilch. Self-assembling of networks in an agent-based model. *Physical Review*, E66:1–9, 2002.

[84] W. Shen, P. Will, and B. Khoshnevis. Self-assembly in space via self-reconfigurable robots. In *Proceedings of the International Conference on Robotics and Automation*, pages 2516–2521, Taipei, Taiwan, 2003.

[85] D. Soloveichik and E. Winfree. Complexity of self-assembled shapes. In *DNA*, pages 344–354, 2004.

[86] K. Stoy, W. Shen, and P. M. Will. Using role-based control to produce locomotion in chain-type self-reconfigurable robots. *IEEE/ASME TRANSACTIONS ON MECHATRONICS*, pages 410–416, 2002.

[87] G. Terrazas, M. Gheorghe, G. Kendall, and N.Krasnogor. Evolving tiles for automated self-assembly design. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation*, Singapore, August 25-28. 2007.

[88] G. Terrazas, N. Krasnogor, G. Kendall, and M. Gheorghe. Automated tile design for self-assembly conformations. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 1808–1814, Edinburgh, UK, Sep. 2005.

[89] R. Unger and J. Moult. A genetic algorithm for 3d protein simulation. In *In Proceedings of the Fifth Annual International Conference on Genetic Algorithms*, pages 581–588, 1993.

[90] J.E. Varner. *Self-Assembling Architecture*. Willey, 1998.

[91] S. Vauthey, S. Santoso, H. Gong, N. Watson, and S. Zhang. Molecular self-assembly of surfactant-like peptides to form nanotubes and nanovesicles. *Proceedings of the National Academy of Science*, 99:5355–5360, Apr 2002.

[92] H. Wang. Proving theorems by pattern recognition. *Bell Systems Technical Journal*, 40:1–42, 1961.

[93] D. J. Watts. *Small Worlds: The Dynamics of Networks between Order and Randomness (Princeton Studies in Complexity)*. Princeton University Press, November 2003.

[94] G. M. Whitesides and B. Grzybowski. Self-assembly at all scales. *Science*, 295(5564):2418–2421, 2002.

[95] G. M. Whitesides, J. P. Mathias, and C. P. Seto. Molecular self-assembly and

nanochemistry: A chemical strategy for the synthesis of nanostructures. *Science*, 29:1312–1318, 29 November 1991.

[96] R. J. Wilson and J. J. Watkins. *Graphs: An Introductory Approach.* New York: Weiley, 1990.

[97] M. Yim, J. Lamping, E. Mao, and J. G. Chase. Rhombic dodecahedron shape for self-assembling robots. *SPL TechReport P9710777*, 1997.

[98] S. S. Zumdahl. *Chemical Principles.* Houghton Millfin Company, 1998.