AMERICAN UNIVERSITY OF BEIRUT

THE EFFECT OF SYSTEM DECOMPOSITION ON PRODUCT PERFORMANCE AND PROCESS CONVERGENCE TIME

by
REEM AKRAM ALI AHMAD

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Engineering Management
to the Department of Industrial Engineering and Management
of the Maroun Semaan Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
June 2020

AMERICAN UNIVERSITY OF BEIRUT

THE EFFECT OF SYSTEM DECOMPOSITION ON PRODUCT PERFORMANCE AND PROCESS CONVERGENCE TIME

by
REEM AKRAM ALI AHMAD

Approved by:

Dr. Ali Yassine, Professor — Advisor
Department of Industrial Engineering and Management

Dr. Samih Isber, Professor — Member of Committee
Department of Physics

Dr. Saif Al-Qaisi, Assistant Professor — Member of Committee
Department of Industrial Engineering and Management

Date of thesis defense: June 19, 2020

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: __Ali Ahmad_______ _____ Reem__________ __ Akram_______

Last First Middle

● Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☐ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☒ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after:

One -X- year from the date of submission of my thesis, dissertation, or project.

Two ---- years from the date of submission of my thesis, dissertation, or project.

Three ---- years from the date of submission of my thesis, dissertation, or project.

Reem Ali Ahmad_____ 05-07-2020___

Signature Date

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my thesis advisor Dr. Ali Yassine for his continuous support and guidance throughout my graduate studies. His permanent motivation, enthusiasm and patience are of the major factors that helped me complete my thesis.

I thank my thesis committee members: Dr. Samih Isber and Dr. Saif Al-Qaisi for their cooperation and support.

I would also like to acknowledge my father Akram, mother Sanaa and my sisters Dana and Sara who always supported and motivated me throughout my research years.

Finally, I would like to express my deepest gratitude to all my friends, especially Rania, Chaza and Jowanna, for their endless encouragement and inspiration.

AN ABSTRACT OF THE THESIS OF

Reem Akram Ali Ahmad for Master of Engineering
Major: Engineering Management

Title: The Effect of System Decomposition on Product Performance and Process Convergence Time

The theory of complex systems, which has been applied successfully in evolutionary biology, is gaining popularity for the modeling and analysis of complex product development (PD) systems. Modeling complex PD systems is essential to understand how system elements and their dependencies impact system properties in several aspects such as performance, convergence, and evolution. In this thesis we use the NK and NKC models to simulate and analyze complex PD systems, which are represented by the design structure matrix (DSM). The main objective is to assess whether these models can be useful in analyzing DSMs; particularly, assessing the effect of architecture and system decomposition on product performance and evolution. All in all, this thesis mainly focuses on the effect of system decomposition when dealing with complex systems and how it helps in reducing the convergence time. However, we show that this comes at reduced system performance, represented by the fitness values. In addition, we include situated learning and design rules in this thesis to examine their impact on system convergence time and process performance and how they contribute to reaching system resolution. Once systems are decomposed, we use design rules through the process of standardization in which components agree in advance, upon each other, on some of the interfaces between the various decomposed subsystems Finally, we present our case study of a gas turbine aero engine decomposed into seven subsystems, each with a different number of components, with external interdependencies among them, to demonstrate our model and present the corresponding results. The case study shows how design rules play an important role in reaching system resolution faster and how standardization of the components' state and fitness saves the system from running any additional needed iterations. This has shown to be a major reason of system convergence with less cost.

CONTENTS

ACKNOWLEDGMENTS v

ABSTRACT vi

LIST OF ILLUSTRATIONS x

LIST OF TABLES xiv

Chapter

1. INTRODUCTION……………………………………………..……ERROR! BOOKMARK NOT DEFINED.

1.1 Background……………………………………………………………...…1
1.2. Motivation…………………………………………………………….….…...3
1.3. Research Objectives…………………………………………………………….3
1.4. Thesis Outline…………………………………………………………….….4

2. LITERATURE REVIEW…………………………………………......5

2.1. NK Model………………………. ……………………………………..…..……..Error! Bookmark not defined.

2.1.1. Effect of Varying K on the fitness values in the NK Model…..………………………………………………...........9

2.1.2 Effect of N and K on the NK Model……………….….….…11

2.1.3. NK Model using Sub-blocks…………………………….....11

2.1.4. Effect of N and K on Random NK model and NK model with Subblocks………………………………………………………12

2. NK Model Extensions………………………………………………………..…14

3. A state only Model……………………………………………………...15

1. Simulation Results……………………………………………..….16

4. Complementary and Conflicting Dependencies……………..………………..…18
1. Simulation Results:AllComplementary and AllConflicting…………………………………………..….20
2. Simulation Results: Low, Moderate and High Complementarity………………………………………..….22

5. Applications of the NK Model…………..…………………………..…...25
1. Manufacturing Fitness ……………………………….…26
2. Organizational Design………………………….……..…26
3. Situated Learning…………………………………….….27

6. Integration vs Modularity in Complex Systems……………………………....28

7. NKC Model Fundamentals……………………………………………..….29

8. Examining the Difference between NK and NKC Models…………………......35
1. Effect of N and K on the Performance of NK and NKC Models ………………………………………………………….....36
2. Effect of N, K and Architecture on NK and NKC Models……………………………………………………..…37
9. Design Rules……………………………………………………………………40

3. MODEL……………………………………………………...42
1. Base Model…………………………………………………………………...42
1. Model Landscape……………………………………….43
2. Methodology of the Search Process………………….……....45
3. Impact of external dependencies on resolved components……51
2. Extended Model: With Design Rules………………..……………..……....54

4. ANALYSIS AND DISCUSSION……………………….… ….....ERROR! BOOKMARK NOT DEFINED.
1. Investigation of product performance and convergence time with different N and S……………………………………………………………………………….....60

2. Divergence of the Number of Unresolved Tasks……………………………..…69

3. Sensitivity Analysis of the threshold value $\bar{t}$………………………………..…...74

4. Investigation of product performance and convergence time considering Design Rules ……………………………………………………………………………..…78

5. Investigation of product performance and convergence time and dynamics as a function of the reinforcement learning………………………………………..….84

5. CASE STUDY…………………………………………………..
ERROR! BOOKMARK NOT DEFINED..89

1. Case Study Simulation-Without Design Rules…………………………………..92

2. Case Study Simulation-With Design Rules…………………………………..…94

6. CONCLUSION…………………………………………
……...…**ERROR! BOOKMARK NOT DEFINED.**98

REFERENCES…………………………………………………..... 101

APPENDIX……………………………………………………...102

ILLUSTRATIONS

Figure		Page
1.	Graph representation of the dependencies between components in the DSM…………………………………………………………	6
2.	DSM Representation of a Complex System………………………	7
3.	Evolution of the Fitness for Various Values of K (N=5)……………	10
4.	Effect of N and K on the DSM’S behavior…………………………	11
5.	Sample of 12 sized DSMs having different dependencies’ distribution…………………………………………………………	12
6.	Variation of the fitness and number of iterations of the random and sub-blocks DSMs as a function of K………………………………	13
7.	DSM Representation of a Complex System………………………	17
8.	Variation of the maximum average fitness and average number of iterations as a function of K for the standard NK model, AllComp and AllConf……………………………………………………..	21
9.	Variation of the maximum average fitness as a function of K with low, moderate and high complementarity……………………….	24
10.	DSM matrix for NKC model……………………………………..	30
11.	Fitness Values (Left) and Number of Iterations (Right) in NK & NKC Models…………………………………………………….	36
12.	Variation of the average max. fitness as a function of K in the NK & NKC models…………………………………………………..	37
13.	Sample DSM architectures (N=12, K=2)………………………….	37
14.	Variation of the fitness values (left-side panels) and the number of iterations (right-side panels) as a function of N in NK and NKC models for different DSM architectures…………………………	39
15.	Initialization Process………………………………………………	45
16.	Choosing a random subsystem j on which learning will be applied…………………………………………………………………	45

17. Individual Learning Process…………………………………………… 47

18. Social Learning Process…………………………………………….. 47

19. Updating Internal Dependencies……………………………………. 48

20. Steps when the focal component becomes resolved………………... 49

21. Steps when the focal component stays unresolved………………… 51

22. Dealing with Unresolved External Dependencies (1)………………. 52

23. Dealing with Unresolved External Dependencies (2)………………. 52

24. Dealing with Unresolved External Dependencies (3)……………… 53

25. Design Rules Model (1)…………………………………………… 54

26. Design Rules Model (2)…………………………………………… 56

27. Design Rules Model (3)……………………………………………. 56

28. Design Rules Model (4)…………………………………………... 57

29. Design Rules Model (5)………………………………………….. 58

30. Variation of the number of unresolved tasks and average performance with time for a system of N=10……………………..... 62

31. Variation of the number of unresolved tasks and average performance with time for a system of N=20…………………… 64

32. Variation of the number of unresolved tasks and average performance with time for a system of N=50……………………… 66

33. Variation of the number of unresolved tasks with time for a system of N=10 for different β values…………………………………… 70

34. Variation of the number of unresolved tasks with time for a system of N=20 for different β values……………………………………… 71

35. Variation of the number of unresolved tasks with time for a system of N=50 for different β values…………………………………….. 71

36. Variation of the number of unresolved tasks with time for two 74

different systems' decompositions of N=50 at β=0.3…………..

37. Variation of the average fitness values and average number of iterations as a function of t bar for N=10……………………… 76

38. Variation of the average fitness values and average number of iterations as a function of t bar for N=20………………………. 77

39. Variation of the average fitness values and average number of iterations as a function of t bar for N=50……………………… 78

40. Matrix representing a system of N=20 components, before the selection of design rules………………………………………. 79

41. Matrix representing a system of N=20 components, after the selection of design rules……………………………………. 80

42. Variation of the average number of iterations with the number of design rules developed…………………………………………… 82

43. Variation of the average fitness values and average number of iterations as a function of Φ for N=10………………………… 85

44. Variation of the average fitness values and average number of iterations as a function of Φ for N=20………………………… 86

45. Variation of the average fitness values and average number of iterations as a function of Φ for N=50……………………….. 87

46. Representation of the gas turbine aero engine (Mascoli, 1999)…….. 89

47. DSM of the Complete Gas Turbine Aero Engine ………………… 91

48. DSM of the Considered Gas Turbine Aero Engine………………… 91

49. Variation of the average performance values of the case study system……………………………………………………………. 93

50. Variation of the average number of unresolved tasks of the case study system……………………………………………………….. 93

51. Variation of the average performance values with and without design rules……………………………………………………….. 96

52. Variation of the average number of unresolved tasks with and without design rules……………………………………………....... 97

TABLES

Table		Page
1.	DSM Representation..	6
2.	Enumeration of the fitness values of the DSM in Figure 2..................	9
3.	Sensitivity Analysis Results on Braha's Model...................................	17
4 (a).	Maximum Average Fitness of the uniform and triangular distribution with low complementarity..	23
4 (b).	Maximum Average Fitness of the uniform and triangular distribution with moderate complementarity..	23
4 (c).	Maximum Average Fitness of the uniform and triangular distribution with high complementarity..	23
5.	Detailed Simulation of the NKC model...	32
6.	Average Fitness when having different components' states	34
7.	Variables of the DSM in Figure 5b in NK and NKC Models	35
8.	Average maximum fitness and number of iterations in along the 1000 runs...	35
9.	Used Notations in the Model	43
10.	Two Cyclic Design Rules between Components	80
11.	Average Fitness Values with and without Design Rules	81
12.	Average Number of Iterations with and without Design Rules	82

13. Distribution of the number of components in the subsystems 89

14. All Cyclic Design Rules between Components 92

CHAPTER 1

INTRODUCTION

1.1 Background

The theory of complex systems, which has been applied successfully in evolutionary biology (to study the dynamics and evolution of biological systems), is gaining popularity in product development (PD) to model and analyze man-made systems (e.g., Frenken and Mendritzki, 2012; Oyama et al., 2015). In fact, the biological domain is considered an analogy to a complex PD system where the genes in biological organisms correspond to the components in a complex PD system, and genes in biological organisms depend on each other in a similar way to the components in man-made systems. Complexity of biological organisms is reflected by the dependencies between the genes. That is, when one gene is mutated, it may not just affect its own functionality but also affects the functionality of all other interdependent genes (Frenken, 2006). The main difference between the two systems is that man-made systems are designed by designers who are responsible for making the design decisions whereas biological systems depend on natural selection (Beesemyer et al., 2011).

This analogy between biological organisms and man-made complex systems is valid in terms of product evolution as well. Products evolve throughout the generations due to the continuous changes in the (interdependent) components' design, which increases the systems' performance. It has been argued that the way these interdependencies are distributed between the system's components (which constitute the product architecture) affects the product's performance and its evolvability (Rivkin and Siggelkow, 2007; Luo, 2015). In this context, modeling complex PD

systems is essential to understand how the system elements and their dependencies impact system properties in several aspects such as product/process performance, development cost, product quality, convergence time, and evolution dynamics.

In this thesis we model the Product development (PD) process as a search on rugged landscape . In particular, we use the NK model to simulate this search. According to the NK model, a product system can be defined as a complex system consisting of a set of N components (or modules), each of which is intended to deliver a specific functionality (Kauffman, 1993). Hence, each component delivers a specific function and, in turn, contributes some value to the overall product system. This value is referred to by the performance or the fitness value of the component. This component's performance depends on its own (design) decision and the decisions of one or more other components (depending on the system architecture). The decisions made at the component level are binary. That is, each component is available in two variants, which represent two alternative designs. A complete product contains exactly one variant of each component. A vector of length N whose i^{th} element represents a variant of the ith component is called a design configuration. Standard practice in the NK literature denotes the variants by 0 and 1, which allows a configuration to be represented by a binary string (e.g., 0010 for a vector of length $N = 4$). The N-dimensional possibility space is called the design space and a specific component configuration defines a product design. Moreover, a complete product has a corresponding product (system) fitness that depends on the fitness values of its components. The actual resemblance of this product fitness is a measure of the performance of the system as a whole. For example, if the system is a team of employees, then the fitness of the system resembles the problem-solving effectiveness of this team (Solow et al., 2000).

1.2 Motivation

Many studies have discussed various approaches to examine the performance of Product Development (PD) projects. These studies were not able to show how design decisions are impacted by the product decomposition and the resulting independencies between the various subsystems. For this, we need a new model to be able to study decomposition and also include important features such as learning and design rules.

In this thesis, we first perform a comprehensive study of the NK model in the literature review section to establish a better understanding of this model. We thoroughly examine the effect of dependencies between components within the same subsystem and between different subsystems. Given our findings in the literature review section, we define a new variant of the NK model in which we divide the whole system into subsystems, taking into consideration both, internal and external, dependencies between components. We extend our model to include the concept of design rules in which we standardize certain components based on well-defined, non-random choosing strategies.

1.3 Research Objectives

The goal of the thesis is to investigate the NK model, to study how decomposition affect the product and process performance of the PD project. We define the product performance as the final design's quality/fitness, and we define the process performance in terms of convergence time. Moreover, we consider additional features in the NK model and examine different scenarios (dividing the system into sub-blocks, NKC model, complementary and conflicting dependencies…) to assess the behavior of the NK model in each of these scenarios. The thesis

also aims to introduce a new model, based on the concept of situated learning and design rules, to test the effect of components' standardization and consequently the impact of lower complexity on the system's performance. The objective of the proposed model is to reach the whole system's resolution with the least cost, i.e. needed number of iterations, and at the highest possible performance.

1.4 Thesis Outline

The thesis is divided into six chapters. The next chapter represents our literature review in which we will present theoretical background related to our research area, in addition to introducing the fundamentals, contributions and applications of NK and NKC models. Chapter 3 represents our model, based on the concept of design rules and situated learning. In chapter 4, we perform several sensitivity analyses test cases to investigate the effect of product/system decomposition on the product performance and convergence time. In chapter 5, we present our case study highlighting the results obtained from applying our model. Finally, we conclude in chapter 6 with suggestions for future research.

CHAPTER 2

LITERATURE REVIEW

Product Development (PD) projects require the collective effort of various teams and members with different backgrounds and conflicting objectives sometimes (Yassine, 2004). Processes in PD projects follow a sequential manner as the completion of a certain task requires the completion of another. This dependency between tasks and hierarchy in information transition in PD projects is a main cause of resulting in complex systems. Complexity makes projects harder to develop and optimize, the thing that results in unexpected outcomes in the system (Simon, 1969, pg. 195). To address this complexity, projects should be well-modeled in a way to manage interconnection between modules and keep track of information within components.
The Design Structure Matrix (DSM) has been introduced to model the relationships between systems and processes and keep track of information from one stage into another. Various project management tools have been previously used to model this flow between processes, however they failed to completely capture the interdependency between the tasks, which is a major complicating factor in PD projects (Yassine, 2004).

A DSM is a matrix representation of a complex system which includes all patterns of information exchange between the dependent elements or components of the system. One of the main advantages of the DSM compared to traditional project management tools is the fact that it keeps track of the feedback between components by marks between the components' nodes. For example, consider the graph in Figure 1, where C feeds A, E, D and F at the time A is being fed by B,C and E. This means that element or component C affects elements A, E and D and element

A depends on elements B,C and E. This dependency between the elements shows how one element needs and uses the information of another which highlights the dependency structure between the elements.

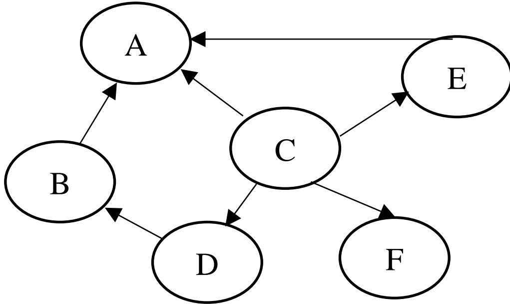


Figure 1: Graph representation of the dependencies between components in the DSM

Table 1: DSM Representation

	A	**B**	**C**	**D**	**E**	**F**
A	X	X	X		X	
B		X		X		
C			X			
D			X	X		
E			X		X	
F			X			X

Moreover, the DSM can be partitioned which decreases the number of dependencies between elements or components, reduces complexity and thus ease the search process. We will discuss this further in details in the coming sections.

In this paper, we will use the DSM as a modeling tool of complex PD projects then will apply the NK model on projects represented by DSMs. Moreover, we will focus on the effect of introducing the learning factor into our model, as well as how design rules help in reaching the desired output with less cost.

2.1 NK Model

We consider a system of *N* components, where each component depends on K other components (Kauffman, 1993). The NK Model is a mathematical representation of these dependencies, i.e. it assigns to each component a mathematical measure that represents the component's fitness value, taking into account the dependencies between components. To apply the NK model, a N size Design Structure Matrix (DSM) is used to model and represent the system and its components' dependencies, as illustrated in Figure 2. Suppose we have 3 components in a system, where the performance of each component depends on its own (design) decision and on the decisions of other components. In this case, N=3 and K=1.

	1	2	3
1		X	

2			X
3	X		

Figure 2: DSM Representation of a Complex System

Figure 2 represents the scenario where each off-diagonal mark "X" represents a dependency between two components (Yassine and Braha, 2003). For example, the DSM (assuming that row *i* depends on column *j*) shows that the performance of component 1 depends on its own (design) decision and the decisions of component 2. Similarly, component 2 depends on component 3, and component 3 depends on component 1.

The NK model starts by randomly assigning to each of the *N* components discrete random states (either 0 or 1) and corresponding random fitness values sampled from a uniform distribution ranging between 0 and 1. The fitness of the system, call it F_1, is the average of the fitness values of the N components and can be calculated according to the formula in Equation (1).

$$F_1=\sum fi/N \qquad (1)$$

Where f_i is the fitness value of component i. In our case, shown in Figure 2, *i* ranges between 1 and 3 ($1 \leq i \leq 3$)since there are 3 components in the system.

Then, one of the *N* components is randomly chosen to change its state and its corresponding fitness value. Furthermore, we change the fitness values of all the components that depend on this chosen component. For example, if we choose to change the state of component *i* ($1 \leq i \leq N$),

then if its state is 0 it becomes 1 and vice-versa. Then, we change the fitness value of component *i* as well as the fitness values of all the components *j* ($1 \le j \le N$) that depend on component *i*.

Finally, the average fitness is recalculated, to obtain a new average fitness, call it F_2. If F_2 is greater than F_1, then we repeat the above process starting with the new obtained string of states and their corresponding fitness values. If F_2 is less than F_1, then we repeat the above process after choosing a component other than one previously chosen. This simulation process continues until a maximum average fitness is reached. Note that if a string of states is revisited, then their corresponding fitness values should be retained.

For the DSM in Figure 2, the NK model works as follows. After randomly initializing the states and the fitness values of these components, we obtain initial states 110 and their corresponding fitness values 0.85, 0.57 and 0.63, resulting in an initial average fitness F_1=0.68 (Refer to the 7th row in Table 1). Then, the third component is randomly chosen so its state changes from 0 to 1 and its corresponding fitness value as well as that of component 2 change to 0.02 and 0.55 respectively, resulting in the 8th row in Table 2

Table 2: Enumeration of the fitness values of the DSM in Figure 2

	States	**f_1**	**f_2**	**f_3**	**F**
1	**000**	0.31	0.72	0.37	0.47
2	**001**	0.31	0.42	0.51	0.41
3	**010**	0.38	0.57	0.37	0.44
4	**011**	0.38	0.55	0.51	0.48
5	**100**	0.15	0.72	0.63	0.5
6	**101**	0.15	0.42	0.02	0.2
7	**110**	0.85	0.57	0.63	0.68
8	**111**	0.85	0.55	0.02	0.47

This iteration results in the new average fitness $F_2=0.47<0.68=F_1$. For this, we return to the initial '110' string states and randomly choose a new component, i.e. any component other than the 3rd component. This process is repeated until a maximum average fitness F_{max} is reached. Table 2. enumerates the total 8 cases of this DSM. It is worth noting that the fitness values are almost always between 0.5 and 0.7 since we are sampling from a Uniform distribution between 0 and 1.

2.1.1 Effect of Varying K on the fitness values in the NK Model

To study the effect of K on the evolution of the fitness values, we ran the NK model on three DSMs of size 5, but with different number of dependencies K: a) K=0, b) 0<K<N-1 and c) K=N-1. The variation of the fitness values in these 3 cases is shown in Figure 3.

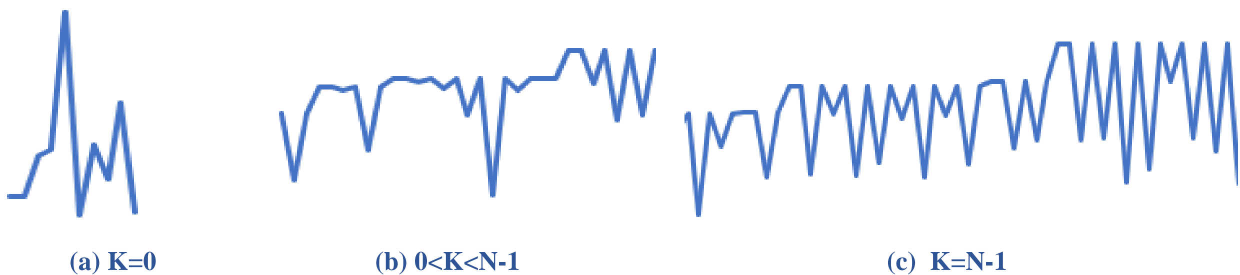


Figure 3: Evolution of the Fitness for Various Values of K (N=5)

Figure 3a represents the case where K=0, i.e. the system has no interactions among its components. In this case, there will only be one state (either 0 or 1) for each element that is responsible for making the highest fitness contribution to the system. This maximum fitness, i.e. the only global optimum, is represented by the highest single peak in Figure 3a. All other sub

optimal fitness values will eventually reach the global optimum after having passed through all their neighboring states, which obviously have lower fitness than the global optimum.

We notice from Figure 3b that as the number of dependencies increases to take any value between 0 and N-1 (K=2 in our case), the number of fluctuations increases, and the graph becomes multi-peaked. In this case, each element depends on multiple other elements in the system, causing the number of the local optima to increase significantly and thus making it harder for each element to reach an optimum.

In the third case, as K reaches it maximum value, i.e. K=N-1 (K=4 in our case), the DSM become a completely rugged landscape where each element depends on all other elements in the system. This property causes the search process for the maximum fitness to be very difficult, as represented by the huge increase in the peaks of the graph, in Figure 3c.

2.1.2 Effect of N and K on the NK Model

To study the effect of the number of elements N and number of dependencies K on the system's behavior, the NK model is applied on several DSMs having different N and K. The corresponding changes in the fitness values and number of iterations are observed and shown in Figure 4.

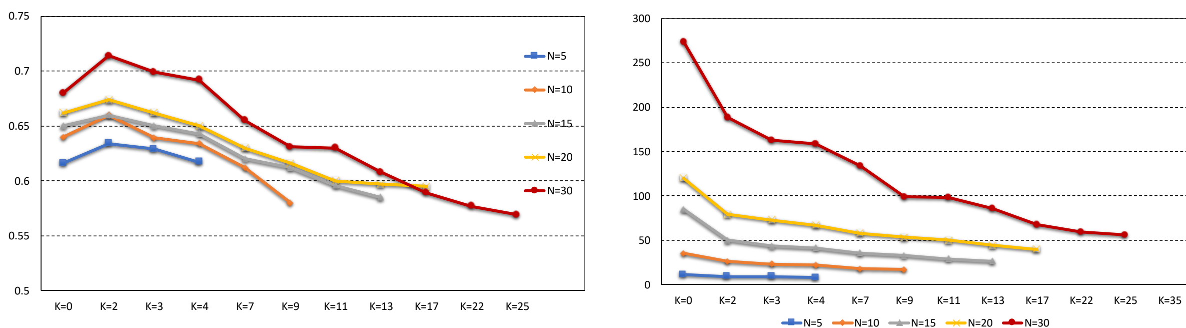


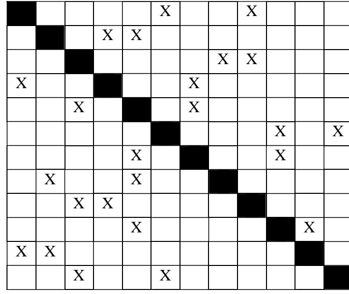
Figure 4: Effect of N and K on the DSM'S behavior

Observation 1: As shown in Figure 4, for a fixed *N* (the number of components) and as *K* (the number of dependencies) increases, both the fitness and the number of iterations are not significantly affected. However, both the fitness and the number of iterations increase with *N* for a fixed *K*.

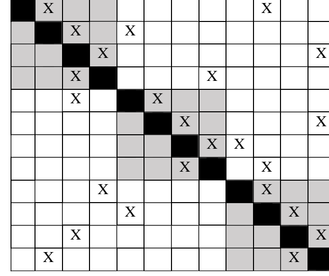
2.1.3 NK Model using Sub-blocks

The NK model is also applied in this section; however, the DSM is divided into sub-blocks prior to simulation. In this case, K is divided into two components; K_i and K_o, where $K_{i+}K_o$=K, K_i is defined as the number of dependencies within the same sub-block, and K_o as the number of dependencies outside the sub-block. For example, consider Figure 4b, where a DSM of size 12 and K=2 (K_i =1 and K_o=1), is divided into three sub-blocks of four components each.

Both Figures 5a and 5b have the same number of components N and dependencies K; however, the main difference is the way these dependencies are distributed. In Figure 5a, interactions between components are randomly distributed, however, in Figure 5b, they are classified according to the number of dependencies within and outside each sub-block, as described above. For example, the first DSM row has 2 marks (i.e. K=2). One of these marks is within the first block in the grey part of the row (since K_i =1) and the other mark is within the white part of the first row (since K_o=1). The rest of the marks are similarly allocated for each row in the DSM.



(a) 12 sized Random DSM

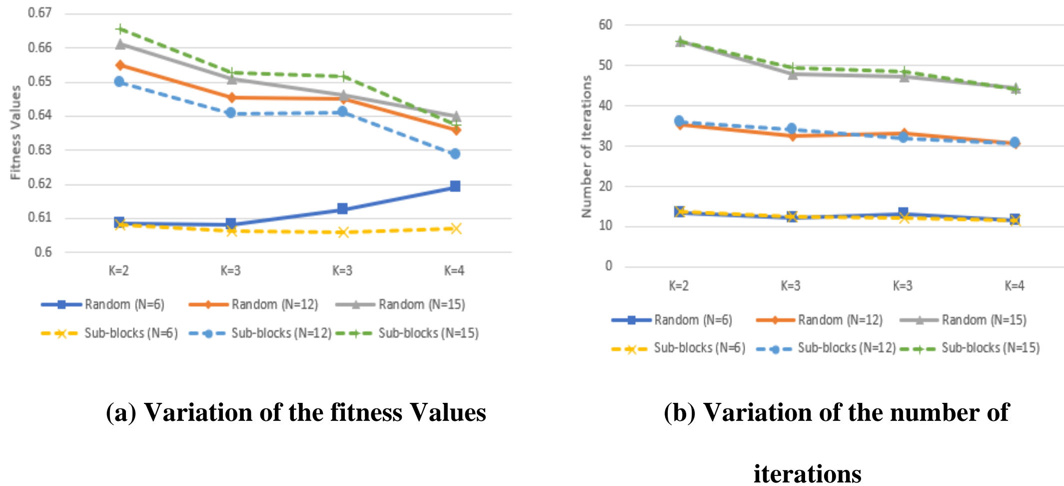


(b) 12 sized DSM with sub-blocks

Figure 5: Sample of 12 sized DSMs having different dependencies' distribution

2.1.4 Effect of N and K on Random NK model and NK model with Sub-blocks

To study the behavior of the DSM with sub-blocks and test how it differs from the random DSM, the NK model is tested for 200 runs on both random and sub-blocks DSMs. This test is applied on DSMs with different number of components (N=6, 12 and 15) and dependencies (K=2, 3 and 4) to compare the maximum average fitness values and the average number of iterations executed by different cases. Also, note that two cases have been considered for K=3; either K_{in} =1 and K_{out} =2 or K_{in}=2 and K_{out}=1.



(a) Variation of the fitness Values

(b) Variation of the number of iterations

Figure 6: Variation of the fitness and number of iterations of the random and sub-blocks DSMs as a function of K

Observation 2: As shown in Figure 6, both random and sub-blocks DSMs behave similarly with an increase in K. We can conclude that the effect of distributing the dependencies between the components using sub-blocks is almost negligible on the system's fitness and number of iterations.

A DSM can be divided into different number of sub-blocks. For example, the 12 sized DSM, shown in Figure 5b, is divided into three sub-blocks of 4 components each; however, it can be divided into 2 sub-blocks of 6 components each, or into 4 sub-blocks of 3 components each, etc. Accordingly, we tested the NK model on a 12 sized DSM, with K=4, divided into different number of sub-blocks to study the effect on the fitness values and the number of iterations. We observed that changing the number of sub-blocks did not significantly impact the fitness values nor the number of iterations.

2.2 NK Model Extensions

As discussed in the introduction, the NK model is a system of N components that are related through interdependencies in which a change in the fitness of one component may affect the fitness of others. Frenken (2006) has introduced an Altenberg's generalized NK model in which the interactions do not have to be between the N components, but occur between N components and F functions, where N and F do not have to be equal. This model has resulted in reducing the static and dynamic transaction costs. Static transaction cost of an element increase as the element depends on more functions whereas the dynamic transaction cost represents the long-term cost, i.e. the cost to reach a local optimum, which requires changes in the fitness values of multiple elements (Frenken 2006).

Product architecture is another major component that affect the system's performance where it has been proven that product architectural patterns play an important role in affecting product evolvability (Luo 2015). Luo concludes that a single change in highly integral complex systems require changes of many other components, unlike the case in highly modular systems. Thus, as the product evolves, less cyclic architectures are favored. A similar approach concerning the effect of product architecture on design evolvability over time has been studied through the Genetic Algorithm (Brabazon et al., 2002). Brabazon concluded that the searching process becomes more difficult as K increases and that a combination between the NK model and the genetic algorithm (GA) is a suitable framework to study the effect of different modular architectures on the evolution of product designs.

2.3 A state only model

A system represents a network of interdependent components where each component can have either a resolved or unresolved state, i.e. state is either 1 or 0 respectively. The state of a component is affected by the number of unresolved elements on which it depends, i.e. the greater the number of unresolved dependencies of a component is, the less the probability of it being resolved (Braha and Bar-Yam, 2007). Other reasons that affect the resolution of the states are the strength of the dependencies between the elements, i.e. how much does a component knows about its unresolved dependencies, which will be discussed in more details in the situated learning section, and the completion rate of a task or a component.

Braha presents his model by introducing two possible cases, either the component's state is initially resolved or unresolved at certain time **t**. The component will either retain its state , at time **t+1**, or switch from resolved to unresolved or vice-versa, according to the equations presented by the model's piecewise function, represented by the below set of Equation 2 (Braha and Bar-Yam, 2007). Moreover, the policies on which a component is modified accordingly influences the final output and specifically affects the number of iterations needed to reach a completely resolved system. It has been shown by Braha and Bar-Yam that modifying components based on a specified planned scheme results in a better performance and less cost than when selecting random components to modify.

In addition, statistical properties of organizational networks represent a significant factor in improving the decision-making process of the organizations. It has been shown that planned task modification policies, that specify the way changes are applied to PD performances, resulted in increasing the performance of the processes to become twice its original value. (Braha et al. 2007).

As a summary of Braha's model, Equation 2 represent the probability of component **i** retaining its current state or changing from a resolved state to an unresolved one or vice versa, as per cases 1 and 2.

Case 1: If component i is resolved at time t

$$\boldsymbol{ci(t+1)} = \begin{cases} \mathbf{0} \;\; \boldsymbol{with\, probability} \;\; \left(\tanh\left(\boldsymbol{\beta i} * \mathbf{yi}\,(\mathbf{t})\right)\right) \\ \mathbf{1} \;\; \boldsymbol{with\, probability} \;\; \left(\mathbf{1} - \tanh\left(\boldsymbol{\beta i} * \mathbf{yi}\,(\mathbf{t})\right)\right) \end{cases} \quad (2a)$$

Case 2: If component i is unresolved at time t

$$\boldsymbol{ci(t+1)} = \begin{cases} \mathbf{0} \;\; \boldsymbol{with\, probability} \;\; \mathbf{1} - \boldsymbol{ri}\left(\mathbf{1} - \tanh\left(\boldsymbol{\beta i} * \mathbf{yi}\,(\mathbf{t})\right)\right) \\ \mathbf{1} \; \boldsymbol{with\, probability} \;\; \boldsymbol{ri}\left(\mathbf{1} - \tanh\left(\boldsymbol{\beta i} * \mathbf{yi}\,(\mathbf{t})\right)\right) \end{cases} \quad (2b)$$

Where $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, $\boldsymbol{ri}$ is the completion rate, $\boldsymbol{\beta i}$ is the sensitivity parameter of component **i** with respect to its neighboring unresolved components and $\mathbf{yi}\,(\mathbf{t})$ is the number of unresolved components on which component **i** depends on.

2.3.1 Simulation Results

We applied Braha's model on the matrix shown in Figure 7, where N=6 and K=3. We considered that each component is affected with a percentage of 35% by its neighboring unresolved components ($\boldsymbol{\beta i}$ =0.35) and that all components are provided with the needed resources to be internally resolved ($\boldsymbol{ri}$ =1).

	1	2	3	4	5	6
1		X	X		X	
2	X		X			X
3	X	X		X		
4			X		X	X
5		X		X		X
6		X		X	X	

Figure 7: DSM Representation of a Complex System

After running Braha's model on the matrix in Figure 7, for 100 runs, we obtained an average of **145.3** needed number of iterations for the whole system to resolve.

In order to further test the effect of the completion rate and betta values on the needed number of iterations for the system to resolve, we performed a sensitivity analysis by varying their corresponding values between 0 and 1. Obtained results are summarized in the below Table 3.

Table 3: Sensitivity Analysis Results on Braha's Model

Number of Iterations				
betta/completion rate	**0**	**0.5**	**0.75**	**1**
0.1	diverge	48.5	19.2	11.2
0.35	diverge	diverge	1808.3	145.3
0.75	diverge	diverge	diverge	diverge
0.95	diverge	diverge	diverge	diverge

In addition, it has been noticed that the product of betta and the number of unresolved states ($\boldsymbol{\beta i} * \mathbf{yi}$) must be less than the completion rate $\boldsymbol{ri}$ or else we will not obtain a definite number of iterations for the system to resolve. This is justified and proved mathematically by Braha and Bar-Yam (Braha and Bar-Yam, 2007).

2.4 Complementary and Conflicting Dependencies

Another significant contribution to the NK model is classifying dependencies into complementary and conflicting dependencies, in addition to sampling the fitness values from a triangular distribution instead of a uniform one. Having a large number of complementary dependencies leads to increase in the average maximum fitness with the incline of K at a lower rate than the original NK model, unlike the conflicting dependencies that result in a decline in the fitness values at a faster rate than the original NK model. Sampling from the triangular distribution results in lower fitness values than when sampling from uniform distribution. (Kyle Oyama et al. 2015).

In this model, the basic NK model is extended in which the following two arguments hold (Oyama et al., 2015).

1. There are two types of dependencies: complementary and conflicting dependencies. A complementary dependency exists if component A depends on component B and any increase in the fitness of component B will thus lead to an increase in the fitness of component A. As for the conflicting dependency, it is the opposite of the complementary where an increase in the fitness of component B will decrease that of

component A. However, if the fitness of component B decreased, no change will occur in the fitness of component A.

This is illustrated in the set of Equations (3) below:

$Y' = y + \frac{(x'-x)}{(1-x)}(1-y)$ for complementary dependencies, when x'>x (3a)

$Y' = y + \frac{(x'-x)}{(1-x)}(-y)$ for conflicting dependencies, when x'>x (3b)

$Y' = y$ when x'<x, irrespective of the dependency type (3c)

Where y' is the new fitness value of the affected component, y=old fitness values of the affected component, x'=new fitness value of the ***focal*** component and x=old fitness value of the ***focal*** component.

The focal component is the component that has its state, and thus its fitness, changed. The method to calculate the new fitness value of the focal component is explained in assumption 2 below.

In this section, the extreme cases will be tested, i.e. when all dependencies are complementary (***Allcomp*** model) and when all dependencies are conflicting (***AllConf*** model).

2. The second assumption is removing the uncertainty when sampling for the new fitness value of the focal component where in the original NK model the sampling used to be random from a uniform distribution. In this assumption, the sampling will be from the triangular distribution by applying the following steps:

✓ Set a probability p that represent the probability that the fitness of the focal component will increase.

p=0.75 in all the following results

- ✓ Model the fitness using the triangular distribution parameters: min, max, mode and p.

  mode=x

  max=x+0.5*(1-x)

  min=(mode-max*(1-p))/p

  where x is the old fitness value of the focal component

2.4.1 Simulation Results: AllComplementary and AllConflicting

In this test, we ran each of the original NK model, the ***Allcomp*** and the ***AllConf*** models over 500 iterations to compare the change of their maximum average fitness values and their average number of iterations, as the number of dependencies K change.

The results of the average fitness and average number of iterations are shown in Figures 8(a) and 8(b) respectively.

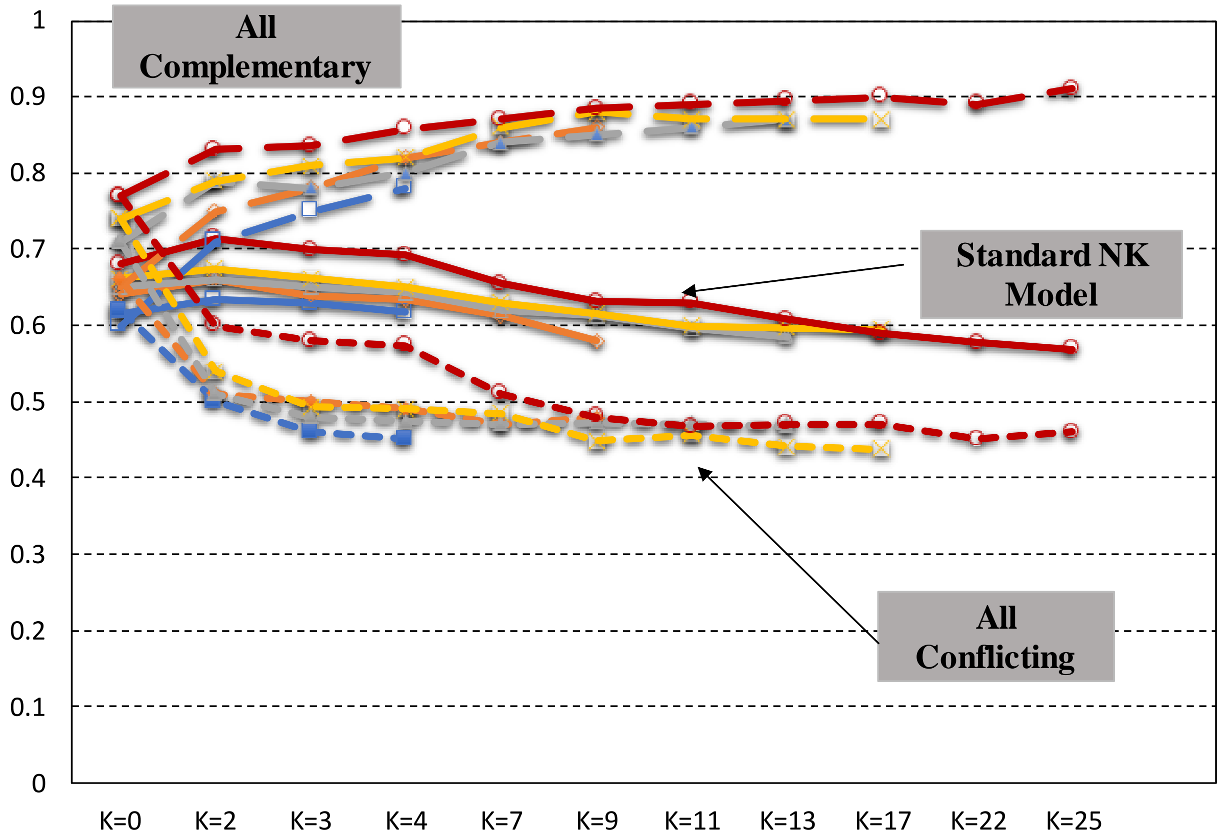


Figure 8(a): Variation of the maximum average fitness as a function of K for the standard NK model, AllComp and AllConf

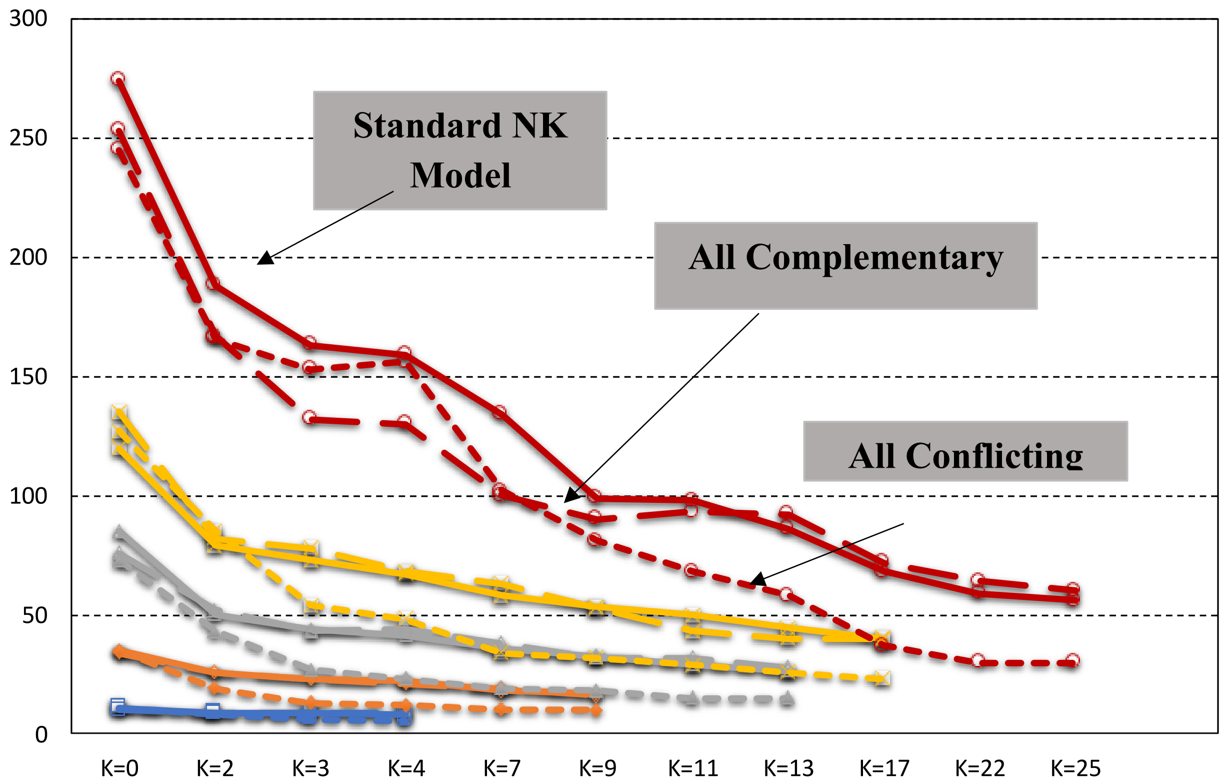


Figure 8(b): Variation of the average number of iterations as a function of K for the standard NK model, AllComp and AllConf

Figure 8: Variation of the maximum average fitness and average number of iterations as a function of K for the standard NK model, AllComp and AllConf

Observation 3(a): As a conclusion from Figure 8, as the number of dependencies increase in the original NK model, the fitness slightly decreases, as observed in previous experiments. However, as the number of complementary dependencies increase the average maximum fitness increase significantly. On the contrast, the average maximum fitness decreases significantly as the number of conflicting dependencies increase.

Observation 3(b): As for the number of iterations, they are slightly decreasing as K increases in the original NK model and the AllComp model, as shown in Figure 8(b). However, they are sharply decreasing in the AllConf model as the dependencies increase to somehow become constant at the end.

2.4.2 Simulation Results: Low, Moderate and High Complementarity

In this test, both assumptions 1 & 2 are combined where 2 different models are run over 500 iterations. The first model is the regular NK model in which the sampling is from a uniform distribution, while in the second model the sampling is from a triangular distribution, as

described in assumption 2 above. These two models are tested in three cases; ***low, moderate and high complementarity.***

- ***Low complementarity is when 25% of the dependencies are complementary***
- ***Moderate complementarity is when 50% of the dependencies are complementary***
- ***High complementarity is when 75% of the dependencies are complementary***

The following results average maximum fitness values are obtained.

Table 4(a): Maximum Average Fitness of the uniform and triangular distribution with low complementarity

	Uniform	***Triangular***
K=0	0.665	0.5
K=1	0.595	0.489
K=2	0.52	0.45
K=3	0.491	0.451
K=4	0.51	0.455
K=5	0.52	0.451

Table 4(b): Maximum Average Fitness of the uniform and triangular distribution with moderate complementarity

	Uniform	***Triangular***
K=0	0.655	0.495
K=1	0.644	0.5
K=2	0.612	0.506
K=3	0.606	0.484
K=4	0.607	0.494
K=5	0.634	0.497

Table 4(c): Maximum Average Fitness of the uniform and triangular distribution with high complementarity

	Uniform	***Triangular***
K=0	0.664	0.498
K=1	0.694	0.537

K=2	0.697	0.535
K=3	0.73	0.522
K=4	0.756	0.535
K=5	0.76	0.545

The results are shown more clearly in below in Figure 9(a) and 9(b).

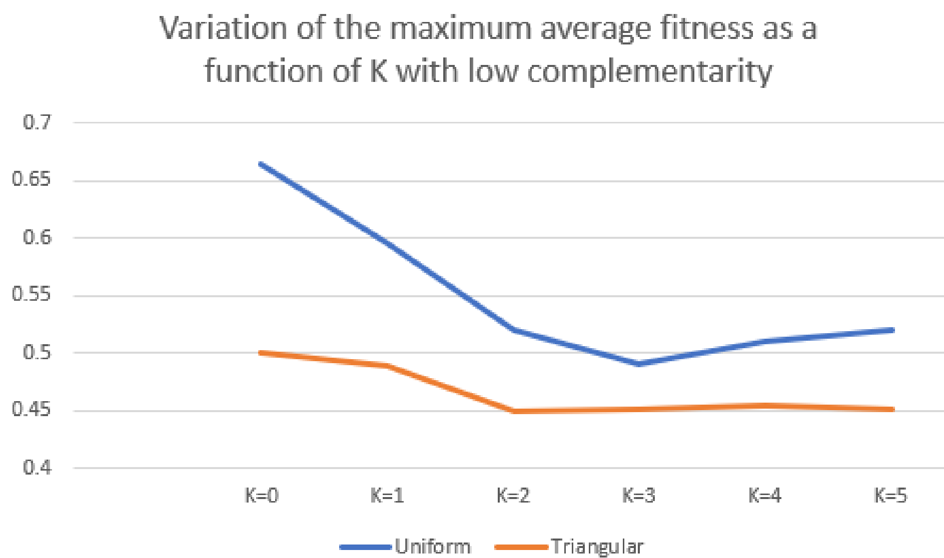


Figure 9(a): Variation of the maximum average fitness as a function of K with low complementarity

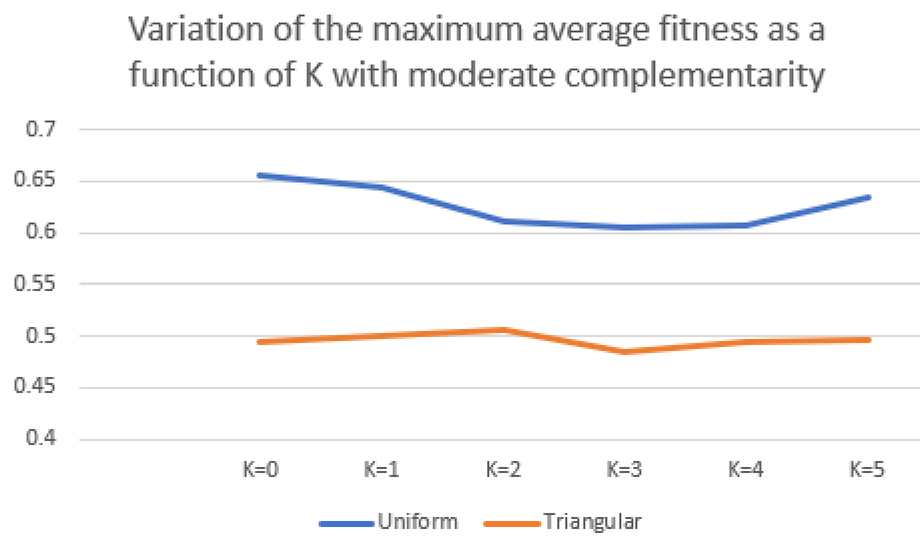


Figure 9(b): Variation of the maximum average fitness as a function of K with moderate complementarity

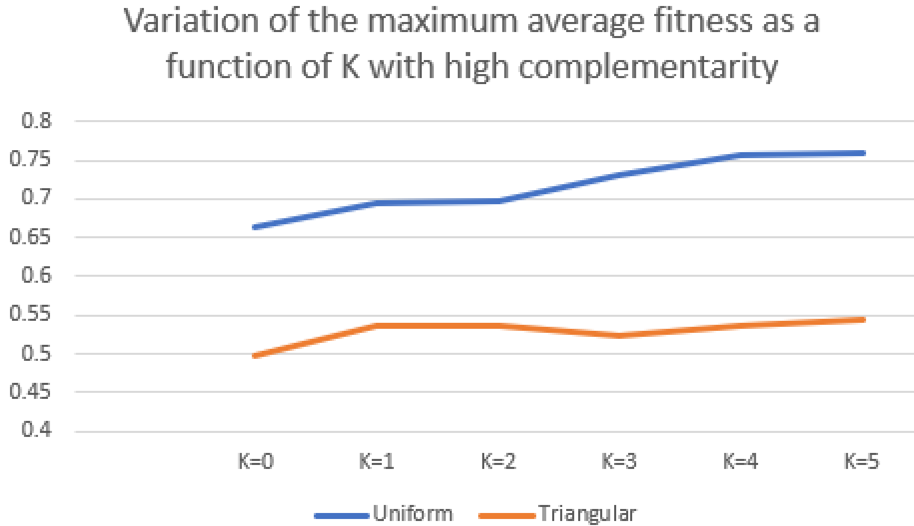


Figure 9(c): Variation of the maximum average fitness as a function of K with high complementarity

Figure 9: Variation of the maximum average fitness as a function of K with low, moderate and high complementarity

Observation 4: Starting with the low complementarity case (the conflicting dependencies are dominant over the complementary ones), it is noticed that as K increases, the maximum average fitness decrease in both the uniform and triangular distributions, as seen in Figure 9(a). As the number of complementary dependencies increase to become equal to the conflicting dependencies (50%-50% distribution), the maximum average fitness doesn't have a clear variation path, i.e. it varies between increasing and decreasing with the increase of K, as shown in Figure 9(b). However, as the number of complementary dependencies increase to 75%, the behavior of the maximum average fitness in Figure 9(c) becomes completely opposite to that in Figure 9(a), since as K increases the maximum average fitness increase in both uniform and triangular distributions. This is reasonable since the number of complementary dependencies

increased, that is the chance of having higher fitness values increased as well. As a conclusion, as the number of complementary dependencies increase, there is a higher chance of having the maximum average fitness increase with K.

Note that the maximum average fitness of the uniform distribution is always greater than that of the triangular distribution in the three cases; low, moderate and high complementary.

2.5 Applications of the NK Model

Despite the origin of the NK model from a biological perspective and its application in genetics, it has been used for modeling and analysis purposes in different separate areas. In fact, the NK model has become a popular option for an organization to refer to in its decision-making processes (McCarthy, 2002). In this section, we will discuss the general fields in which the NK model is been recently used.

In general, the idea of evolution is usually linked to the biological context and away from technological, industrial, social, etc. perspectives, however evolution is present and valid in various fields of product development. In the process of searching for the best solution or the optimal state of a product, we always tend to change the behavior/elements/components of the system, the thing that reflects the product's evolution (Oyama et al., 2015). This validates the application of NK model in real life product development projects, rather than just biological ones.

2.5.1 Manufacturing Fitness

A more specific application of the NK model in the NPD field is the Manufacturing Fitness area. McCarthy has translated the original definition of the NK model, by Kauffman, to the strategy and management domain such that, despite the system's nature, the NK model can be applied as long as this system evolves with time. McCarthy focused on the evolutionary process of the system where he described the term "fitness" as the "Darwinian fitness" which resembles the components' capacity to survive and reproduce. (McCarthy , 2002).

When applying the NK model in the manufacturing domain, McCarthy describes the N parameter to be the number of parts in an organization, K to the number of interconnectedness between these and A to represent the number of states that an organizational part can have.

2.5.2 Organizational Design

Besides the number of components and the number of their corresponding dependencies, the way in which these components interact plays an important role in product development. This interaction between components and the location of dependencies in a DSM is referred to as the organizational design which is also known by system architecture.

In this context, the NK model is used to explore how the structure of systems can affect product performance. (Yuan & McKelvey, 2004) has applied the NK model to real life relationship where the parameter N represents the number of people in a team and K represents the number of communication linkages between these people. It has been shown that there will an initial increase in the team performance as the number of communication linkages increase. However, if these linkages continue to increase more, the team performance would decrease (Yuan &

McKelvey, 2004). This obtained conclusion from the NK model reflects the expected theoretical consequences, according to Yuan.

Moreover, it has been claimed through the NK model that modular product architecture increases the speed of performance evolution (Luo, 2015).”Modular” product architecture is referred to when dividing the system into smaller modules with maximum interactions within these modules and eliminating or minimizing the external dependencies between them (Baldwin and Clark, 2000).

2.5.3 Situated Learning

Several interactions occur in a complex system, whether between the subsystem’s internal components or between the components of different subsystems. Components having internal dependencies within the same module are familiar with the design characteristics of their neighboring elements more than those having interactions with components in a different module. This knowledge between the subsystems’ components resembles the weight of these interactions and how familiar a component is with the strength of the connections (Songhori et al., 2017).

Search processes in complex systems need the effort of components from different modules in order to govern for both, internal and external interconnections which require local and global searches respectively. In local search, components conduct individual learning to improve their internal design choices within their own subsystem. As for global search, elements have limited knowledge about the search domains in other subsystems and are not certain enough about the strength of the external dependencies. To deal with these unknown external connections,

components perform coupled learning, a trial and error-based process, to improve their knowledge and learn more about the invisible outer connections. It is shown that when focusing and allocating the weight to the inner interactions, search is concentrated within the one module which makes the learning process easier and thus reaching the design solutions faster than when paying attention to the density of external dependencies interactions (Songhori et al., 2017).

2.6 Integration vs Modularity in Complex Systems

As shown above, in the NK model organizations follow a structured search process within the system taking into consideration the entire system's components simultaneously. Applying the NK model to a system and dealing with it as a one integrated block might be difficult to manage especially in complex systems where the number of the components N and their corresponding dependencies K is high.

To make complexity more manageable, another approach can be followed, where the organization can rather divide the system into smaller modules in a way that each module is responsible for a certain number of the entire system's components. This form of splitting the system into smaller parts is called "Modularity". Dividing the system into modules is not enough as, besides being responsible for its internal components, each module must account for the external dependencies between its components and components in other modules. Having these external dependencies obliges the subsystem to consider the effect of changing the fitness or state of any of its components on the dependent components in other subsystems. For example, the effect of any change in the performance or configuration of a **task A in subsystem 1** has to be taken into consideration when visiting **task B in subsystem 2,** knowing that task B in

subsystem 2 depend on task A in subsystem 1. By this, the system is changed from being integrated to modular. The way these external dependencies are designed and how teams decide on them will be discussed later in the Design Rules section.

The concept of integration in one system as a whole was clearly illustrated in the NK model, as shown in the previous section, however modularity will be modeled in the following section when discussing the extended version of the NK model-the NKC model.

2.7 NKC Model Fundamentals

In this section, we introduce the NKC model (Hordijk and Kauffman, 2005) in which the components' dependencies are divided into teams/units/subsystems. The NKC model classifies the dependencies of each component between internal and external dependencies and calculates the system's average performance based on this classification as described below.

We consider a system of size *N*, but with *S* subsystems, where:

- N: number of total components that are distributed along S sub systems
- K: number of inter dependencies inside the sub system
- C: number of external dependencies, that is each component in each sub system depends on C other components from other sub systems
- Nj': number of components inside subsystem j. Note that the number of components within a one subsystem may differ from the number of components in another subsystem

We start by randomly assigning discrete random states (either 0 or 1) and random fitness values sampled from a Uniform distribution ranging between 0 and 1 to all components in all

subsystems. Then, a random subsystem *j* ($1 \leq j \leq S$) is selected and a component *i* from subsystem *j* is randomly chosen to change its state and its corresponding fitness value. Next, we randomly sample for the fitness of all components in subsystem *j* that depend on component *i*. The average fitness of subsystem *j* is calculated as follows in Equation 4:

$$F_j = \sum fi / N_j' \qquad (4)$$

where f_i represent the fitness value of component *i* in subsystem *j*.

If the new average fitness of subsystem *j* is greater than the previous average fitness, then we sample for the fitness values of components, in subsystems other than subsystem *j*, which depend on component *i*. While if the new average fitness of subsystem *j* is lower than the previous average fitness, we chose another component from subsystem *j*.

These steps are repeated until a maximum average fitness of subsystem j is reached. After applying the above scenario for all subsystems S, the maximum average fitness of the whole system is calculated as follows in Equation 5.

$$F = \sum F_j / S \quad (5)$$

	s1n1	s1n2	s1n3	s2n1	s2n2
s1n1	x	X		X	x
s1n2		X	x	X	x
s1n3	x		X	X	x
s2n1	X	X		X	x
s2n2		X	X	X	x

Figure 10: DSM matrix for NKC model

Table 5: Detailed Simulation of the NKC model

n1	n2	n3	s2n1	s2n2	f1	f2	f3	f_{avg}
0	0	0	0	0	0.81	0.63	0.50	0.57
0	0	0	0	1	0.85	0.02	0.49	0.45
0	0	0	1	0	0.43	0.89	0.27	0.53
0	0	0	1	1	0.74	0.45	0.60	0.60
0	0	1	0	0	0.81	0.13	0.37	0.44
0	0	1	0	1	0.85	0.29	0.14	0.43
0	0	1	1	0	0.43	0.24	0.90	0.52
0	0	1	1	1	0.74	0.02	0.24	0.33
0	1	0	0	0	0.34	0.34	0.50	0.39
0	1	0	0	1	0.42	0.18	0.49	0.36
0	1	0	1	0	0.92	0.99	0.27	0.73
0	1	0	1	1	0.14	0.04	0.60	0.26
0	1	1	0	0	0.34	0.23	0.37	0.31
0	1	1	0	1	0.42	0.94	0.14	0.50
0	1	1	1	0	0.92	0.83	0.90	0.88
0	1	1	1	1	0.14	0.10	0.24	0.16
1	0	0	0	0	0.86	0.63	0.37	0.62
1	0	0	0	1	0.22	0.02	0.31	0.18
1	0	0	1	0	0.29	0.89	0.13	0.44
1	0	0	1	1	0.26	0.45	0.07	0.26
1	0	1	0	0	0.86	0.13	0.05	0.35
1	0	1	0	1	0.22	0.29	0.90	0.47
1	0	1	1	0	0.29	0.24	0.23	0.25
1	0	1	1	1	0.26	0.02	0.17	0.15
1	1	0	0	0	0.54	0.34	0.37	0.42
1	1	0	0	1	0.23	0.18	0.31	0.24
1	1	0	1	0	0.89	0.99	0.13	0.67
1	1	0	1	1	0.78	0.04	0.07	0.30
1	1	1	0	0	0.54	0.23	0.05	0.27
1	1	1	0	1	0.23	0.94	0.90	0.69
1	1	1	1	0	0.89	0.83	0.23	0.65
1	1	1	1	1	0.78	0.10	0.17	0.35

n1	n2	n3	s2n1	s2n2	f1	f2	f_{avg}
0	0	0	0	0	0.13	0.06	0.10
0	0	0	0	1	0.13	0.72	0.43
0	0	0	1	0	0.56	0.16	0.36
0	0	0	1	1	0.56	0.14	0.35
0	0	1	0	0	0.06	0.06	0.06
0	0	1	0	1	0.06	0.72	0.39
0	0	1	1	0	0.75	0.16	0.46
0	0	1	1	1	0.75	0.14	0.45
0	1	0	0	0	0.72	0.69	0.71
0	1	0	0	1	0.72	0.06	0.39
0	1	0	1	0	0.51	0.40	0.46
0	1	0	1	1	0.51	0.09	0.30
0	1	1	0	0	0.22	0.69	0.46
0	1	1	0	1	0.22	0.06	0.14
0	1	1	1	0	0.76	0.40	0.58
0	1	1	1	1	0.76	0.09	0.43
1	0	0	0	0	0.94	0.80	0.87
1	0	0	0	1	0.94	1	0.97
1	0	0	1	0	0.01	0.28	0.15
1	0	0	1	1	0.01	0.59	0.30
1	0	1	0	0	0.23	0.80	0.52
1	0	1	0	1	0.23	1	0.62
1	0	1	1	0	0.95	0.28	0.62
1	0	1	1	1	0.95	0.59	0.77
1	1	0	0	0	0.77	0.22	0.50
1	1	0	0	1	0.77	0.08	0.43
1	1	0	1	0	0.21	0.04	0.13
1	1	0	1	1	0.21	0.92	0.57
1	1	1	0	0	0.86	0.22	0.54
1	1	1	0	1	0.86	0.08	0.47
1	1	1	1	0	0.12	0.04	0.08
1	1	1	1	1	0.12	0.92	0.52

Note that changing the fitness values of components in other subsystems, due to the external dependencies, may not necessarily improve the average fitness values of those subsystems

For the DSM in the Figure 10, the NKC model work as follows. After randomly initializing the states and fitness values of these components, we obtain initial states of **101** in subsystem 1 and **01** in subsystem 2 resulting in a string of states **10101,** in the first table, with fitness of **0.47** and in a string of states **01101**, in the second table, with fitness **0.14.**

When choosing a subsystem at random, we chose the first subsystem with a randomly chosen component (component 2). So, the string of states becomes **111** in subsystem 1 and **01** in subsystem 2, which results in states **11101,** in the first table. In the first subsystem. we sample for the fitness of the second component and component 1 as well since it depends on component 2 in the 1st subsystem from a uniform distribution, resulting with an average of **0.69** in the first subsystem.

Since the new average fitness of subsystem 1 (**0.69)** is greater than the old average fitness of subsystem 1 **(0.47),** so we now deal with the external dependencies, i.e. we sample for the fitness values of components in subsystem 2 which depend on component 2 in subsystem 1, which are both, components s2n1 and s2n2. So, we sample from a uniform distribution the fitness values of components 1 and 2 in subsystem 2, resulting in an average fitness of **0.43** in subsystem 2.

Now we continue our search in subsystem 1, until we reach a maximum fitness in it (like applying the NK model on the 1st subsystem). When choosing another random component from the string **11101,** and sampling for the fitness values of this randomly chosen component and its dependents, it turns that the string **11101** has the greatest average fitness in subsystem 1 among its neighboring states. This can be summarized in Table 6 below.

Table 6: Average Fitness when having different components' states

States	Subsystem 1 Average Fitness
11101	0.69 (highest)
10101	0.47
01101	0.5
11001	0.24

So now that we have reached the maximum average fitness in subsystem 1, which is **0.69**, with a string of states **11101**, we move to the 2nd subsystem. We start with the states at which we have reached the maximum fitness in subsystem 1, i.e. **11101.** This string corresponds to the string **01111** in subsystem 2, with the average fitness **0.43.**

We choose component 2 in the 2nd subsystem at random and we change its state and the corresponding fitness and that of component 1 in the 2nd subsystem. We end up with the string of states **00111** in subsystem 2 with an average fitness of **0.45**. Since **0.45** is greater than **0.43**, then we go with the new configuration **00111**, in the second subsystem, which corresponds to **11100** in the 1st subsystem. Here we account for the external dependencies and we resample for the fitness values of all 3 components in the 1st subsystem, since they all depend on the 2nd component in subsystem 2, resulting in an average fitness of **0.27.**

We keep searching all local neighbors of the string **00111**, in the 2nd subsystem to end up with the string **10111** having the maximum average fitness **0.77**, This corresponds to the states **11110** in the 1st subsystem with an average fitness **0.65.**

Finally, the total fitness of the system is the average of the two, which is the average of **0.77** and **0.65**, which is **0.71.**

2.8 Examining the Difference between NK and NKC Models

To notice the difference between the NK and NKC models, both models were run in parallel for 1000 runs on the 12 sized DSM, represented in Figure 5b. The variables of this DSM, represented in both NK and NKC models, are shown in Table 7.

Table 7: Variables of the DSM in Figure 5b in NK and NKC Models

	NK Model	NKC Model
N	12	12
K	2	1
C	0	1
S	1	3
N'	-	4

The average fitness values and average number of iterations of the 1000 runs are recorded in Table 8. The simulated 1000 maximum fitness values and number of iterations are displayed in Figure 11.

Table 8: Average maximum fitness and number of iterations in along the 1000 runs

	NK Model	NKC Model
Average Maximum Fitness	0.6505	0.5874
Average Number of Iterations	32.99	24.468

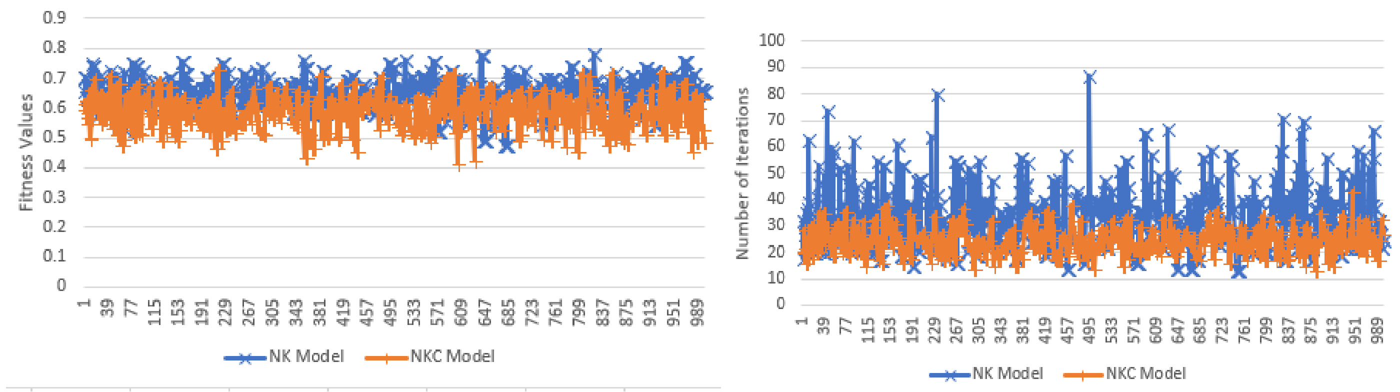


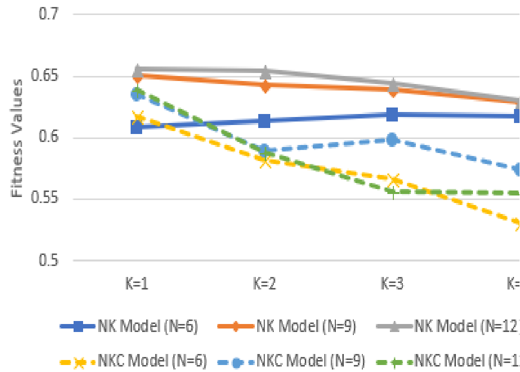
Figure 11: Fitness Values (Left) and Number of Iterations (Right) in NK & NKC Models

Observation 5: As shown in Table 8 and Figure 11, the NKC model reach a lower maximum fitness, on average, than the NK model and at a lower average number of iterations as well.

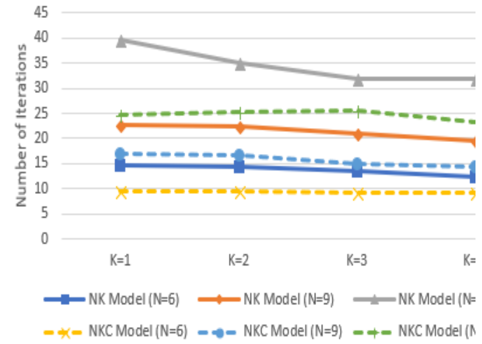
However, along the 1000 runs, there is not a clear relation between the maximum fitness of the NK and NKC models in each run. As for the number of iterations, the NKC model clearly takes less iterations than the NK model, almost throughout all the 1000 runs, as shown in Figure 11 (right).

2.8.1 Effect of N and K on the Performance of NK and NKC models

To test the effect of changing N and K on the system's performance in each of the NK and NKC models, both models are applied on DSMs of different sizes (N=6, 9 and 12) and different dependencies (K=1, 2, 3 and 4). Note that changing the number of dependencies in the NKC model is done by either increasing the number of internal dependencies K or the number of external dependencies C.



(a)Variation of the average maximum fitness



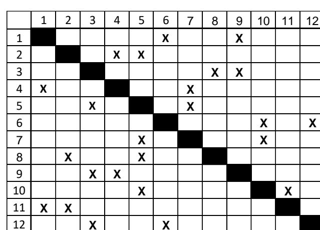
(b)Variation of the avg. number of iterations

Figure 12: Variation of the average max. fitness as a function of K in the NK & NKC models

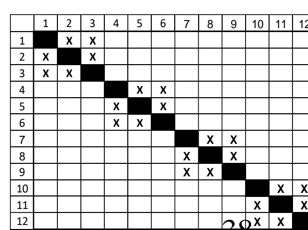
Observation 6: As shown in Figure 12, as the number of dependencies K increase, the (average) maximum fitness of both the NK and NKC models generally decrease. In the NK model, the decrease occurs slowly as K increases and the curve somehow remains flat, however the fitness in the NKC model decreases at a faster rate, and this is clear from the slopes that appear to be steeper in the NKC model.

2.8.2 Effect of N, K and Architecture on NK and NKC Models

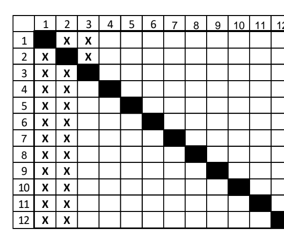
In this section, we perform a comprehensive study in which we test both, the NK and NKC models, on different numbers of elements N (12 and 16), different number of dependencies K (1,2 and 3) and different architectures (Random, Block-Diagonal, and Centralized).



(a) Random



(b) Block Diagonal



(c) Centralized

Figure 13: Sample DSM architectures (N=12, K=2)

Sample DSMs of the different architectures is shown in Figure 13. The results of this test (fitness and number of iterations), for each of the three architectures, in the NK and NKC models are presented in Figure 14.

Observation 7: When comparing the fitness values of the DSMs of different architectures in Figure 14 (left-side panels) it is noticed that the Random DSM almost has the lowest fitness values for both values of N=12 and N=16 in the NK and NKC models. On the other hand, we can see that the Centralized DSM always has the highest fitness values. As for the Block-Diagonal DSM, its fitness values vary between those of the Random and Centralized DSMs. In Figures 14 (right-side panels), we can see that the Centralized DSM execute the highest number of iterations, whereas the Block-Diagonal takes the lowest number of iterations for both values of N=12 and N=16 in the NK and NKC models. It is noticed that the difference in the number of iterations executed between the three architectures increase in each of the NK and NKC models as N increases, i.e. the difference in the number of iterations between the three architectures is greater when N=16 than when N=12. Also, when comparing the variation of the number of iterations for the different values of K (Figures 14 (b), (d), (f)), it is noticed that the behavior and pattern of variation is the same.

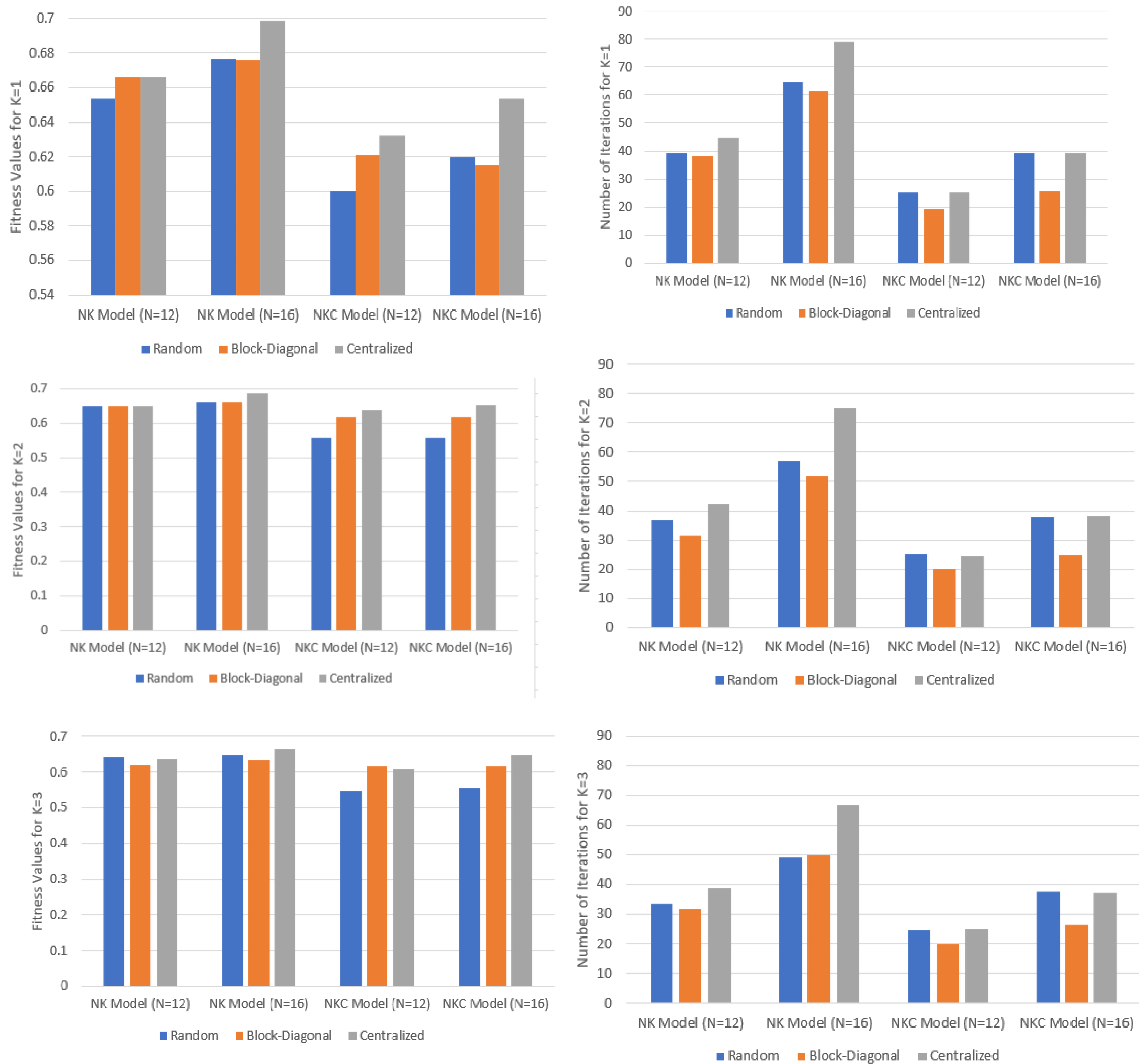


Figure 14: Variation of the fitness values (left-side panels) and the number of iterations (right-side panels) as a function of N in NK and NKC models for different DSM architectures

2.9 Design Rules

As previously mentioned, modularity is essential to manage system complexity that promotes the concurrent development of various modules in the system. However, these separate subsystems are just chunks of one whole system and are designed independently of each other where each perform its work internally separate from the others. This internal work accounts for the internal dependencies between the subsystem's components. After the internal work, all dependent modules must work together and collaborate as a whole (Baldwin and Clark, 2000). For this, external dependencies occur between the system's modules in order to ensure that coherence between dependent components in separate modules is taken into account.

To account for these external dependencies and for modules to function as one system, design rules are introduced to the model.

Design rules are developed through the agreement between the system's components on standardizing certain decisions, which are modeled by the external dependencies in the DSM. By standardization it is meant that organizations ensure that early design decisions that impact a number of downstream tasks or components in the system will be fixed early on by mutual agreement by both upstream and downstream teams. Such early agreement prevents against possible future iteration, which can be expensive (Baldwin and Clark, 2000). So, when components in a certain subsystem start working to enhance their performance, they won't be affected by the external dependencies developed as design rules as they will follow the standard decision previously set.

Before determining a design rule, the designers have to be familiar with the details of the system. This is essential because good knowledge of the system and its options and dependencies allows for efficient development of design rules. Once components define the unnecessary external dependencies and come up with the design rule, these dependencies are removed (Woodard and Clemons, 2014).

Consequently, design rules reduce system complexity by minimizing the number of interactions between modules through eliminating the unnecessary ones, which decrease the development time by reducing iterations. Introducing design rules is essential to decrease system complexity and thus reducing the possibility of Kauffman's "Complexity Catastrophe", as it is shown that overwhelming the system with interdependencies results in inefficient output (Yuan & McKelvey, 2004).

Irrespective of the advantages of design rules and their benefits in resulting in a better overall system performance, it is costly to decide on a design rule and establish it. Organizations will have to employ their components in an experimentation process, i.e. R&D, to determine which components are compatible with the system and end up with the best possible result (Baldwin and Clark, 2000). Basically, the cost of specifying the design rule is the time and effort organizations spend to agree on the standardization process of a dependency. Therefore, organizations must look deeper into this when specifying a design rule to check whether it is worth the cost or not.

CHAPTER 3

MODEL

In this chapter, we introduce a new model of product development based on complex systems. We assess the model's behavior through measuring the product/system performance using the average fitness value and convergence time (i.e. the number of iterations) within each subsystem and in the overall system. The objective of our model is to resolve all components in all subsystems taking into consideration the interdependencies between components and subsystems (that is the resolution of some components may lead to a change of state for other components). We present our base model in section 3.1 and our extended model, in which we introduce design rules, in section 3.2.

3.1 Base Model

Our base model consists of three main parts: (a) description and specification of the model landscape over which the search process occurs, (b) the methodology of the search processes in which the states and fitness values of components are updated, (c) the resolution of components which were unresolved due to the update of external dependencies. Table 9 contains a summary of all notations used in our model.

Table 9: Used Notations in the Model

Notation	Representation
N	Number of total components
S	Number of subsystems
K_{in}	Number of internal dependencies
K_{out}	Number of external dependencies

c_{ij} (t)	State of component i in subsystem j at time t
f_{ij} (t)	Fitness value of component i in subsystem j at time t
$\beta_{i,j}^{i',j'}(t)$	Dependency (Sensitivity) value of component i in subsystem j on component i' in subsystem j' at time t
r_i	Internal completion rate of component i sampled from U(0,1)
x_{ij} (t)	Number of unresolved components in subsystem j on which component i depends on at time t
y_{ij} (t)	Number of internal and external unresolved components at time t
$\bar{t}$	Threshold to check whether to update the external dependencies' states or not
$0 \leq \Phi \leq 1$	Reinforcement learning parameter

3.1.1 Model Landscape

Consider a system with N components where each component has a state and a corresponding fitness value that represents its performance. The state of a component is either "one", i.e. resolved, or "zero", i.e. unresolved. The whole system of N components is divided into S subsystems, where each subsystem j is composed of N_j components.

Let $c_{ij}(t)$ be the state of component i in subsystem j at time t and $f_{ij}(t)$ be its corresponding fitness value. We begin by assigning the components in all subsystems at time 0 to a random state configuration $C(0)=\{c_{11}(0), c_{21}(0),\ldots, c_{12}(0), c_{22}(0), \ldots\}$ and randomly sampling their fitness values from a uniform distribution between zero and one, **U(0,1)**, which is represented by the fitness contribution $F(0): \{f_{11}(0), f_{21}(0),\ldots, f_{12}(0), f_{22}(0), \ldots\}$. The fitness of each subsystem j is the average of the fitness values of each component i in subsystem j. We denote the fitness of subsystem j by $F_j(t)$ as in Equation (6). Furthermore, the fitness of the whole system is the average of the fitness values of each subsystem j as shown in Equation (7)

$$F_j(t) = \frac{\sum_{i=1}^{Nj} f_{ij}(t)}{N_j} \qquad (6)$$

$$F(t) = \frac{\sum_{j=1}^{N} Fj(t)}{N} \qquad (7)$$

Each component is affected by its own state/decisions and the decisions/states of other components on which it depends and in return affects its dependent components. A change from an unresolved to a resolved state depends stochastically on the number of unresolved components it depends on and its internal completion rate (Braha and Bar-Yam, 2007).

In our model, K_{in} represents the number of internal dependencies a component i in subsystem j has with components in the same subsystem j. As for the external dependencies, K_{out} represents the number of external dependencies component i in subsystem j has with components in other subsystems. $K = K_{in} + K_{out}$.

Each component has a dependency strength that represents how much component i in subsystem j knows about the influence of component i' in subsystem j', irrespective if component i' is in the same module (subsystem) with component i or not (i.e. $j=j'$). We symbolize this dependency value by $\beta_{i,j}^{i',j'}(t)$. Another major factor which affects the search process is the completion rate which we refer to by $r_{ij}(t)$. Completion rate represents the resource allocation intensity allocated for each component to be successfully resolved (Braha and Bar-Yam, 2007).

Starting with the initialization process, summarized in Figure 15, we set the betta values $\beta_{i,j}^{i',j'}(t)$ for each component i with respect to every dependent component j to a random value, sampled from a Uniform distribution U(0,1). We continue with the initialization process by defining a threshold parameter $\bar{t}$ which specifies whether component i should attend to the external dependencies from other subsystems or not. That is $\bar{t}$ specifies when we must apply modifications on the external dependent components. In addition, $\bar{t}$ is also a metric that defines the interval in which we update the dependency values, $\beta_{i,j}^{i',j'}(t)$, according to the recent changes in the system.

Start with the Initialization process by:

- Assigning a random configuration for the states
- Sampling for the fitness values from a Uniform distribution U (0,1)
- Assigning random values for the parameters in our model, such as β, and $\bar{t}$

Figure 15: Initialization Process

3.1.2 Methodology of the Search Process

Since our model aims to have a resolved state for all components, we start dealing with systems that contain components in an unresolved state, as we do not apply any intended changes on resolved components, whose state is one. First, we randomly choose a subsystem ***j*** on which we will apply one of two learning processes, as shown in Figure 16.

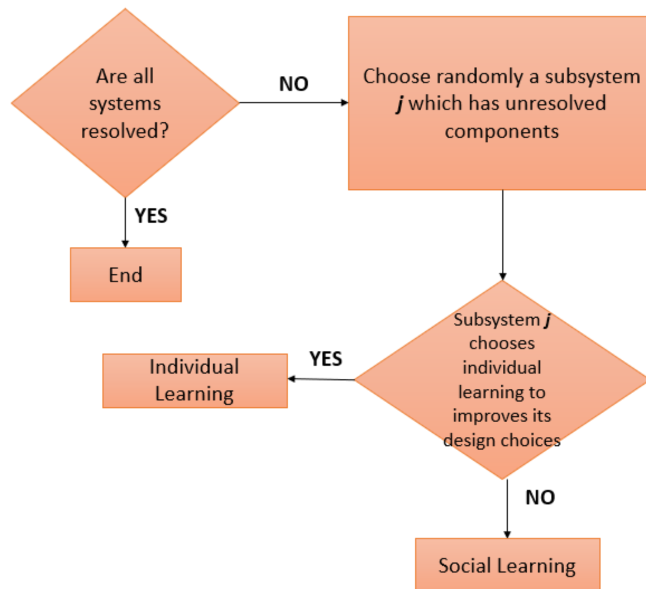


Figure 16: Choosing a random subsystem j on which learning will be applied

There are two types of learning and improvement that occur during the system simulation. The following is a description of each type:

1- **Individual Learning:** In individual leaning, ***subsystem j*** chooses itself and performs local search among its components to resolve its states. At time t, a random unresolved ***component i*** is chosen from ***subsystem j*** on which Equation (8) is applied, where $\mathbf{x_{ij}(t)}$ **is the number of unresolved components in subsystem** ***j*** on which component ***i*** depends, $\mathbf{r_{ij}(t)}$ and is the internal completion rate of component ***i***, which is randomly sampled from a uniform distribution between 0 and 1, **U(0,1).** So, according to Equation (8), the lower the number of unresolved components $\mathbf{x_{ij}(t)}$, the higher the possibility component ***i*** would be resolved (Braha and Bar-Yam, 2007).

$$c_{ij}(t) = \begin{cases} 0 \;\; with\, probability \;\; 1 - r_{ij}(t)\,(1 - tanh\,(\beta_{i,j}^{i',j'}(t) * x_{ij}(t)) \\ 1\, with\, probability \;\; r_{ij}(t)\,(1 - tanh\,(\beta_{i,j}^{i',j'}(t) * x_{ij}(t)) \end{cases} \quad (8)$$

(Braha and Bar-Yam, 2007).

Where $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

According to Equation (8)**,** the component's new state depends on the number of unresolved states that it depends on inside ***subsystem i,*** as previously explained. When applying Equation (8) to change the focal component's state, the corresponding fitness of ***component i*** is affected as well. To adapt for this change, a new fitness of ***component i*** is sampled from a Uniform distribution U (0,1). Individual learning steps are shown in Figure 17.

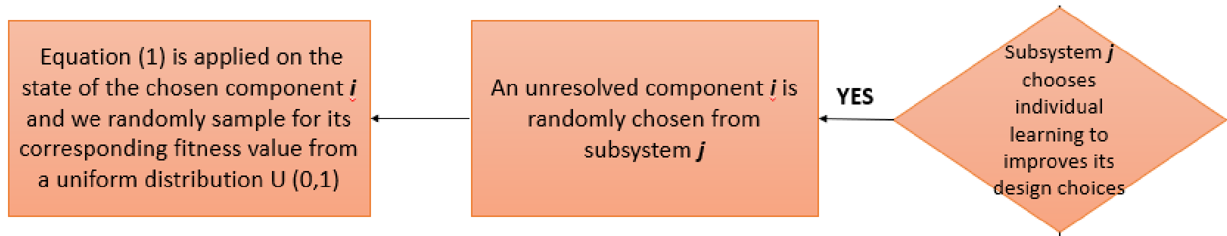


Figure 17: Individual Learning Process

2- **Social learning:** As for social learning, summarized in Figure 18*,* ***subsystem j*** chooses another ***subsystem j',*** provided that there are dependencies between the components of the two subsystems*.* A random unresolved ***component i*** of ***subsystem j*** is selected to be partially imitated from a random ***component i'*** from ***subsystem j'***. So, the state of ***component i*** will be equal to that of ***component i'***, thus, ***component i*** will either stay unresolved or becomes resolved by imitating a selected design choice from ***subsystem j'***. By this, resolution of the focal component is achieved through social learning between interdependent subsystems. Similar to individual learning, after applying changes to the state of the ***focal component i***, its fitness is affected and thus a new fitness is sampled from a Uniform distribution U (0,1).

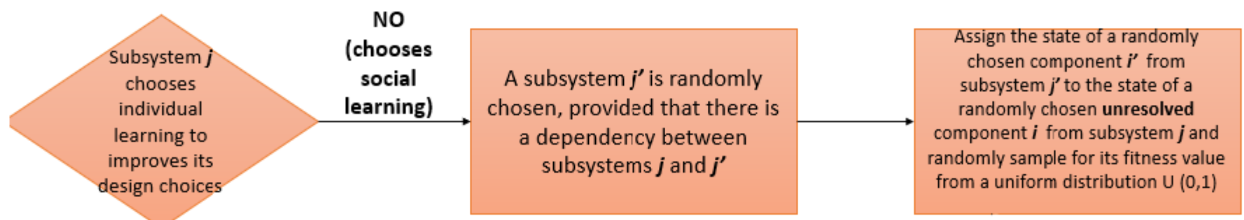


Figure 18: Social Learning Process

After changing the focal component's state and fitness, whether in the individual or social learning approach, the state and performance of components which depend on the ***focal component i***, internally and externally, will be affected. For this, the states and corresponding fitness values of these dependent components have to be updated by the current changes which happened internally in the system. So, as shown in Figure 19, Equation (8) is applied on components which depend on ***component i*** within ***subsystem j,*** taking into account the updated number of unresolved components in ***subsystem j***.

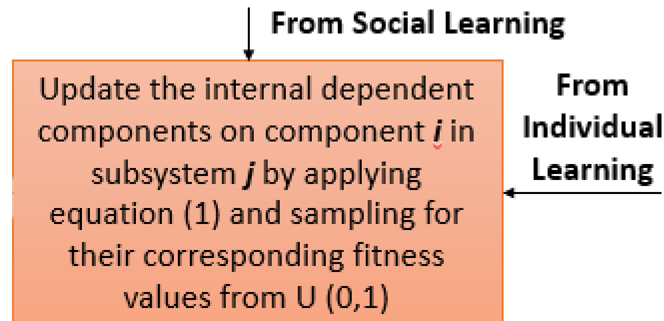


Figure 19: Updating Internal Dependencies

After applying the needed changes on the focal component and its dependent internal components, the result must be evaluated to decide whether to continue with the new configuration and contribution of the subsystem or to keep the old one. This is based on the new state of the ***focal component i*** after applying Equation (8), i.e., it may become resolved or stay unresolved.

In the first scenario, in which the new state of the focal component becomes resolved, we continue with the new updated ***subsystem j***'s configuration and its corresponding fitness contribution, regardless if the new ***subsystem j***'s average fitness is better than the original or not,

as our aim is to resolve the whole system's states with the highest possible performance. Figure 20 summarizes this scenario and the simulation steps performed as discussed next.

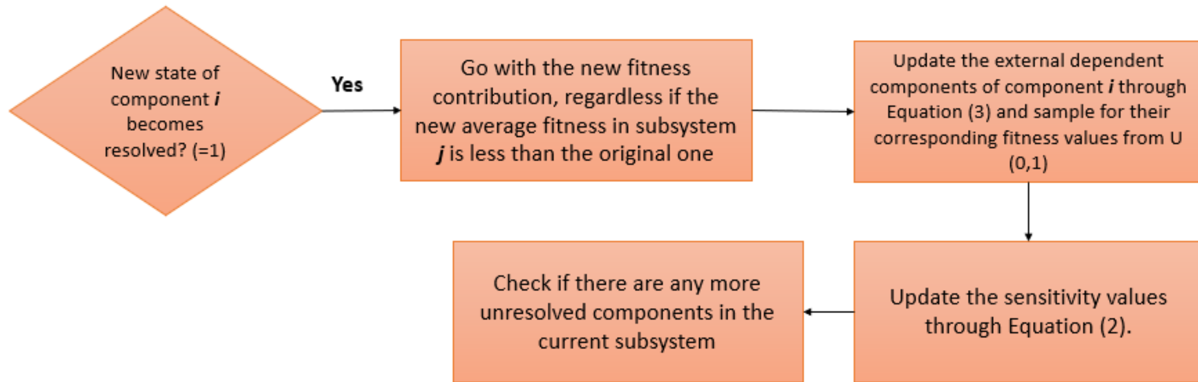


Figure 20: Steps when the focal component becomes resolved

After changing the state and the fitness of the ***focal component i***, and taking into account its dependent internal components, we must deal with the dependent components in other subsystems as well. Updating external dependent components with the changes of the ***focal component*** depends on the parameter $\bar{t}$, defined earlier. This is illustrated by the fact that changing the design choices of a certain subsystem or department in a company does not have to be always announced or communicated (from other subsystems) to other departments. For this, $\bar{t}$ represents a specification metric to which the ***focal subsystem j*** refers when deciding whether to update external dependent components or not. This is done by checking the value of **t mod** $\bar{t}$**;** if it's equal to 0, then we must inform the external dependencies with the recent changes and adapt them accordingly, else we do not. Informing the external dependencies is done by applying Equation (9) on the dependent external component on the ***focal component i***.

Equations (8) and (9) are somehow similar, where the main difference is parameter $\mathbf{y_{ij}(t)}$. In Equation (9), the number of unresolved components, resembled by $\mathbf{y_{ij}(t)}$, represents both, internal and external unresolved components which each dependent external component depends on.

$$c_{ij}(t) = \begin{cases} 0 \;\; with\, probability \;\; 1 - r_{ij}(t)(1 - \tanh\left(\beta_{i,j}^{i',j'}(t) * y_{ij}(\mathrm{t})\right)) \\ 1 \; with\, probability \;\; r_{ij}(t)(1 - \tanh\left(\beta_{i,j}^{i',j'}(t) * y_{ij}(\mathrm{t})\right)) \end{cases} \quad (9)$$

Product Development (PD) usually requires mutual effort from different departments for functional groups in a firm, where each department has its own specialization. Each department has limited knowledge about other departments, at the time where one employee in the department can have a higher (lower) level of knowledge than another employee (Songhori et al.,, 2017). For this, knowledge is earned along the search process and each person in the firm will learn more about his relationships and strength of dependency with his colleagues. In out model, fitness values are being updated and components tend to learn more about the strength of their interdependencies. This is obvious because when components are being resolved and their corresponding fitness values are being updated accordingly through the search process, **the focal component** is having more knowledge about its internal and external dependencies. For this, after the update of the fitness values through the search process, the level of knowledge of the strength between these dependencies is updated accordingly. This is accomplished by updating the dependency values, if $\mathbf{t \bmod \bar{t}}$ is within the acceptable range, as shown in Equation (10) below.

$$\beta_{i,j}^{i',j'}(t) = \beta_{i,j}^{i',j'}(t-1) + \Phi[(f_i(\mathrm{t}) - f_i(\mathrm{t} - \bar{t})) - \beta_{i,j}^{i',j'}(t - \bar{t}) \quad (10)$$

Where Φ is called the reinforcement learning factor which is sampled randomly from a uniform distribution U (0,1).

As for the second scenario**,** represented in Figure 21, if the focal component remains unresolved, then we look into the subsystem's average fitness in this case. We check whether the new ***subsystem j's*** average fitness has improved (i.e. is greater than the previous subsystem's average fitness). Based on this comparison, we go with the subsystem states' contribution having a greater average fitness. In this scenario, there is no need to update the external dependencies accordingly as the state of the focal component do not change. Similar to the first scenario, components need to learn about the strength of the dependencies within other components, irrespective whether the focal component remained unresolved or not. For this, we check if **t mod** $\bar{t}$ is within the acceptable range to update the dependency values, accordingly, as stated previously in Equation (10).

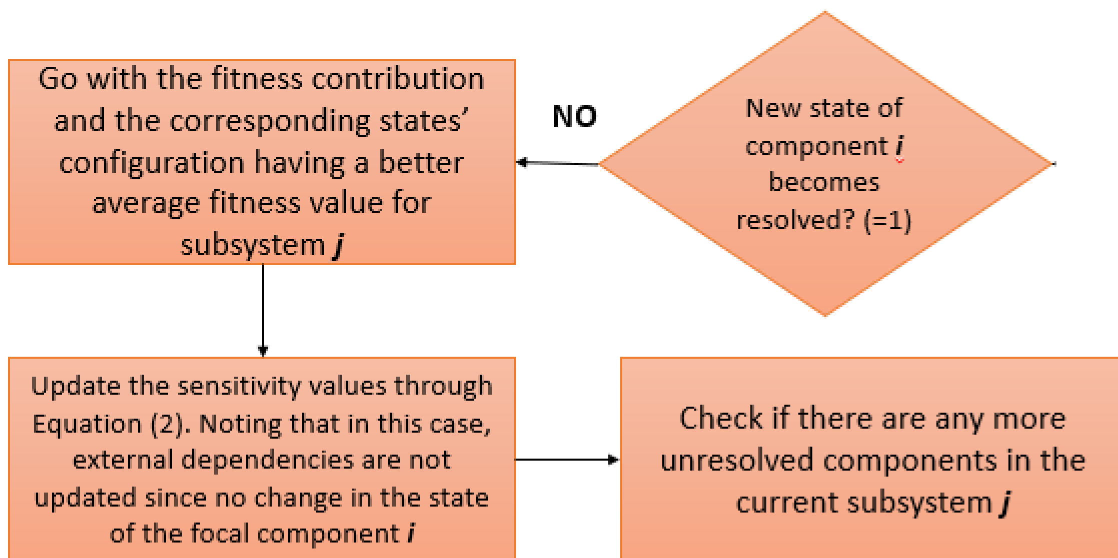


Figure 21: Steps when the focal component stays unresolved

3.1.3 Impact of external dependencies on resolved components

As subsystems are being resolved, the process of updating the dependent external dependencies (in other subsystems) may render some already resolved components to become unresolved again. For this, a check must be done on these components. An unresolved component ***i*** is chosen from a randomly chosen subsystem ***j***, on which we apply Equation (9) and sample for its fitness from a uniform distribution U(0,1). Then, we adapt the states and fitness values of the internal dependencies of component ***i*** accordingly through applying Equation (9) on all internal dependent components within the subsystem ***j***, and sample for their fitness values from U(0,1), as shown in Figure 22.

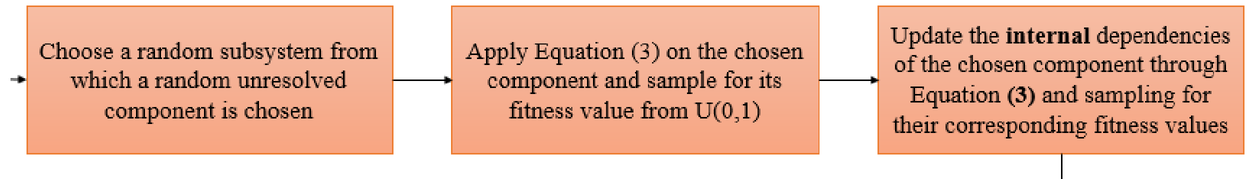


Figure 22: Dealing with Unresolved External Dependencies (1)

After applying these changes, we check whether the state of component ***i*** has been resolved or not. If the new state of component ***i*** is 1, i.e. resolved, we go with the new subsystem ***j***' states' configuration and its corresponding fitness contribution, regardless if the new average fitness is better or not, as shown in Figure 23.

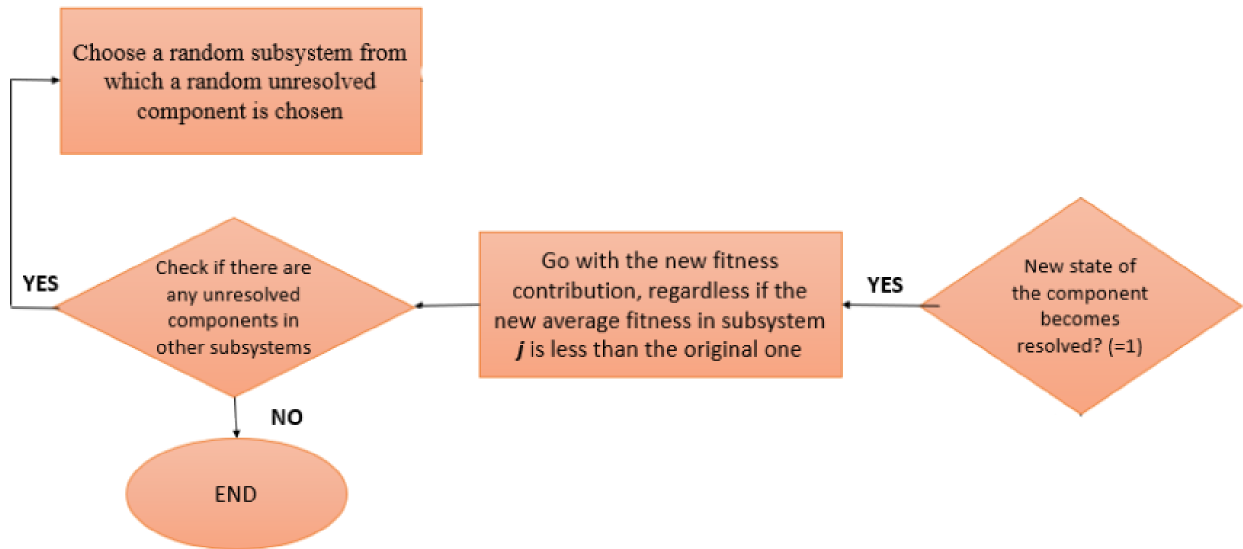


Figure 23: Dealing with Unresolved External Dependencies (2)

Otherwise, if the state of the focal component ***i*** remained unresolved, then we check if the new average fitness contribution of subsystem ***j*** is better than the old one and go with the states' configuration and corresponding fitness contribution having higher average fitness, as shown in Figure 24.

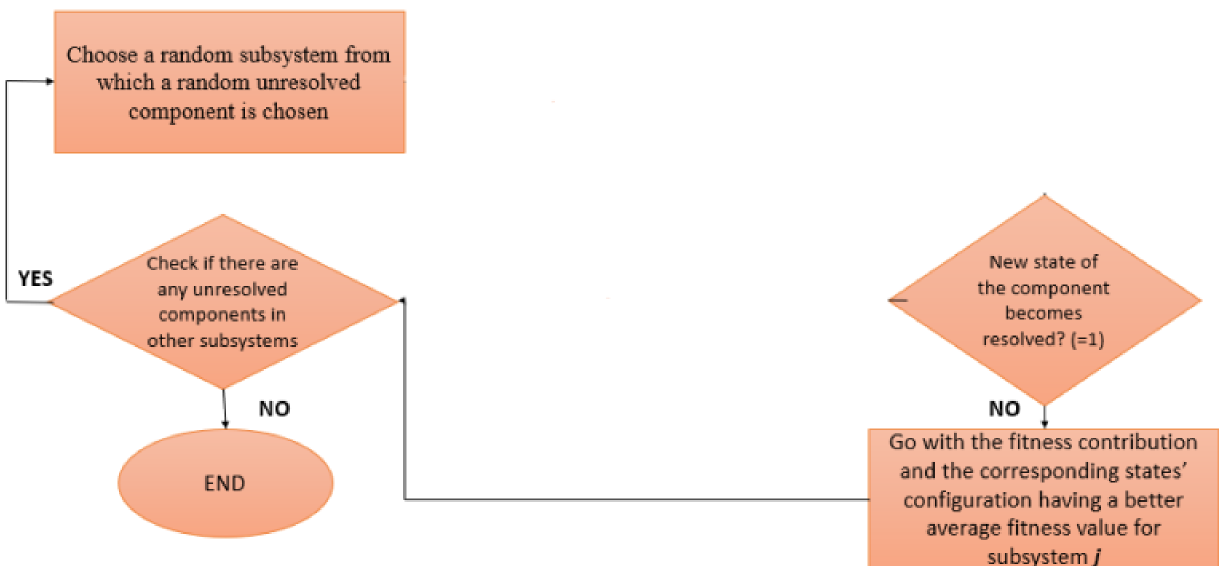


Figure 24: Dealing with Unresolved External Dependencies (3)

After we finish with component ***i***, we deal with all unresolved states within subsystem ***j***, until it becomes resolved. Then, we choose another subsystem ***j'***, having unresolved components, at random, until we have all subsystems within the whole system totally resolved. In the Appendix, we show the full flowchart of the base model in Figure A1.

3.2 Extended Model: with Design Rules

In this section, we introduce into our base model the concept of design rules. As discussed earlier in the literature review section, design rules help in decreasing system complexity by providing upfront mutual agreement in design decisions between components in different subsystems. Design Rules are based on the concept of cyclic dependencies between components, i.e. if **component A** depends on **component A', component A'** depends on **component A''** and **component A''** depends on **component A**, we have a cyclic dependency of length 3. After initializing the states and fitness values for all components in subsystems, dependency values, completion rate and defining the parameter $\bar{t}$**,** we check if cyclic dependencies exist to start setting the design rules, else, we proceed similar to our base model, i.e. without design rules, as shown in Figure 25.

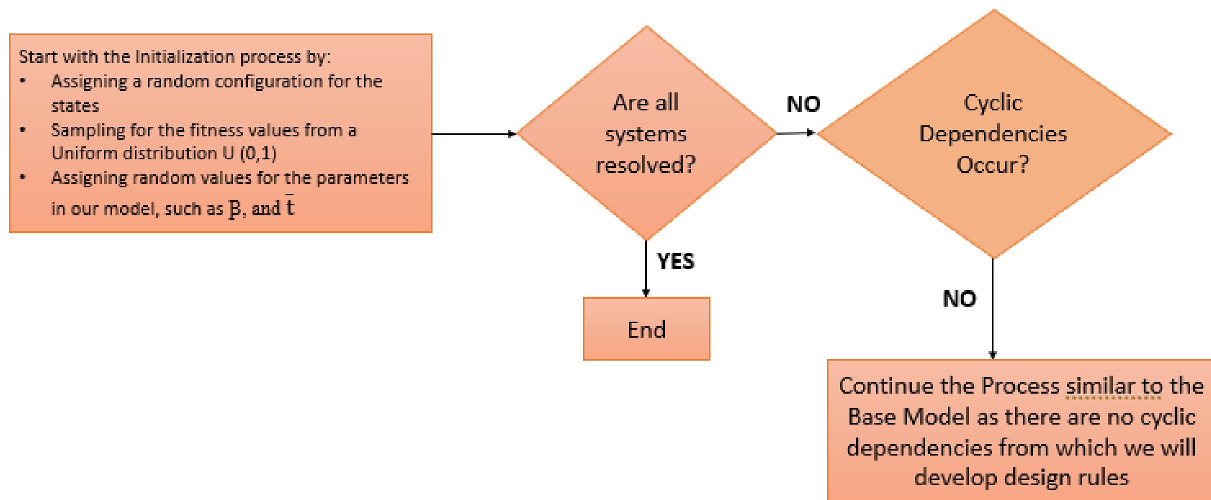


Figure 25: Design Rules Model (1)

If cyclic dependencies occur, we specify a certain number of design rules to be developed and the corresponding size of the cycle connecting the components, i.e. the number of the components to be included in the design rule. Specifying which design rules upfront might be more beneficial for the system than randomly selecting the design rules and this is due to the knowledge of the system (some maybe easier than others to be developed as design rules). After specifying the number of design rules to be developed, we search for cyclic dependent components through the Depth First Search (DFS) Algorithm (Morin, P., 2013). Depth First Search Algorithm (DFS) is an algorithm used in graph data structures and network theory. The algorithm begins the search process starting from the root node and "explores one subtree before returning to the current node and then exploring another subtree". For the selected cyclic dependencies, we standardize the states of all components involved in this cycle, i.e. we set their states to 1, as shown in Figure 26. In addition, the fitness of all components within the same cycle is set to the minimum fitness value of all components within the cycle[1].

[1] Alternatively, we can take the average, but we chose the minimum as a worst-case situation since Design Rules usually reduce the design freedom and potential for innovation/increase

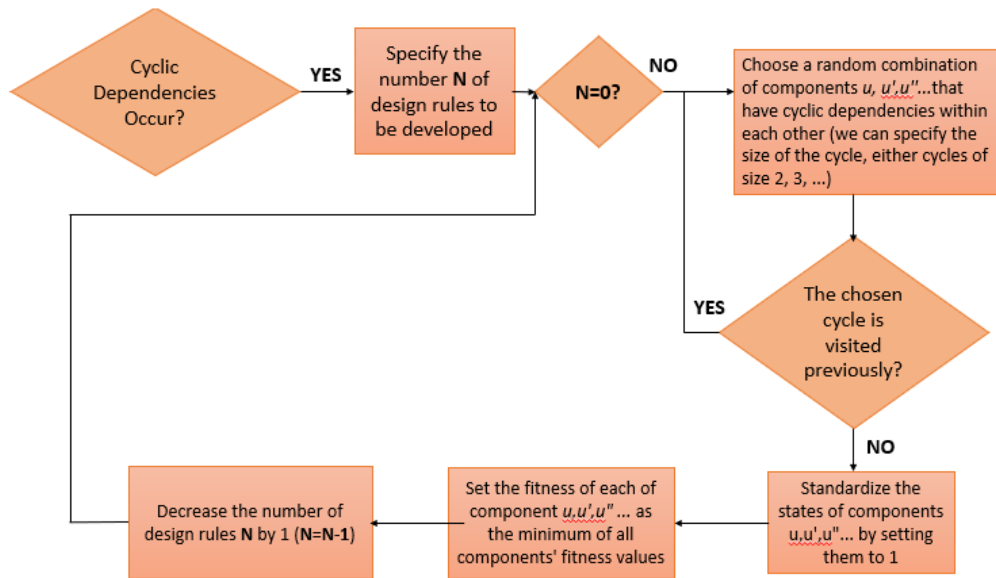


Figure 26: Design Rules Model (2)

After specifying the design rules and standardizing the states and fitness values of the corresponding components, we continue similar to the base model. We first start with either individual or social learning, depending on the chosen subsystem, as in our base model. We apply Equation (8) on the state of the focal component $\boldsymbol{i}$, provided that the chosen focal component $\boldsymbol{i}$ is not involved in a design rule and change its corresponding fitness value by sampling from a Uniform distribution U (0,1), as shown in Figure 27.

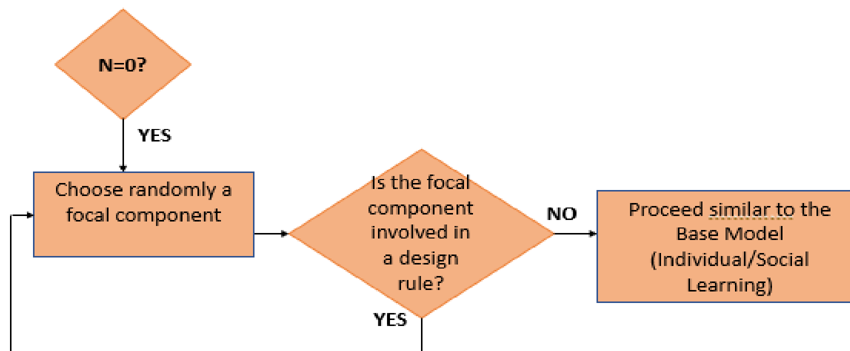


Figure 27: Design Rules Model (3)

First, we update the internal dependent components on the focal component ***i*** and we check if the internal dependencies are design rules or not. If an internal component that depends on component ***i*** is a design rule, we keep its state standardized, i.e. equal to 1, and its fitness fixed. Else, if the dependent component is not a design rule, we apply Equation (8) on it and sample for its fitness from U(0,1). These steps are shown in Figure 28. Similar to the base model, we will have two cases based on the current new state of the focal component ***i.*** as follows:

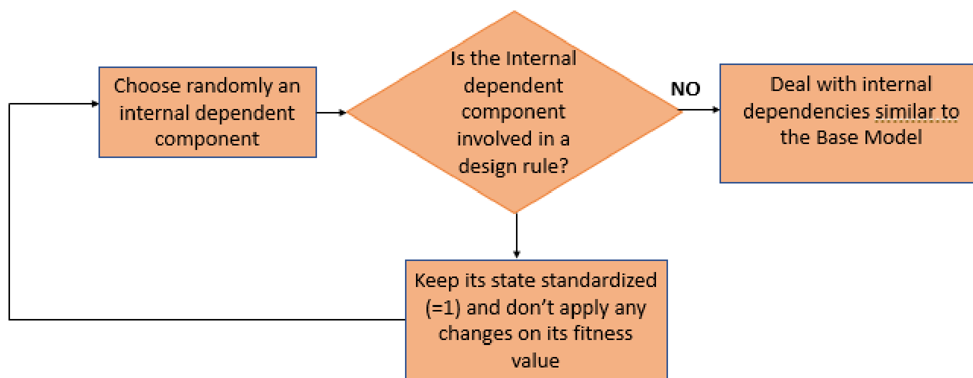


Figure 28: Design Rules Model (4)

In the first scenario, the new state of the focal component becomes resolved, hence we continue with the new updated ***subsystem j***'s configuration and its corresponding fitness contribution, regardless if the new ***subsystem j***'s average fitness is better than the original or not. Then, we deal with the external dependencies of component ***i***, after checking the value of **t mod** $\bar{t}$, as discussed earlier in the base model. If it is equal to 0, we must inform the external dependencies with the recent changes and adapt them accordingly, else we do not. Informing the external

dependencies is done by applying Equation (9) on the dependent external components on the ***focal component i.*** Similar to the internal dependencies, external dependencies which are categorized as design rules, are kept standardized, i.e. we do not change neither their states nor their fitness values, as shown in Figure 29. After dealing with internal and external dependencies, we update the dependency values accordingly through applying Equation (10), for the components to be up to date with the amount of strength with its dependencies.

As for the second scenario**,** if the focal component remains unresolved, we look into the subsystem's average fitness in this case. We check whether the new ***subsystem i's*** average fitness is greater than the previous subsystem's average fitness and we go with the states of the subsystem having a greater average fitness. As previously stated, in this case we don't update the external dependencies accordingly as the state of the focal component didn't change. Similar to the first scenario, components need to learn about the strength of the dependencies within other components, irrespective whether the focal component remained unresolved or not. For this, we check if **t mod** $\bar{t}$ is within the acceptable range to update the dependency values, accordingly, by applying Equation (10).

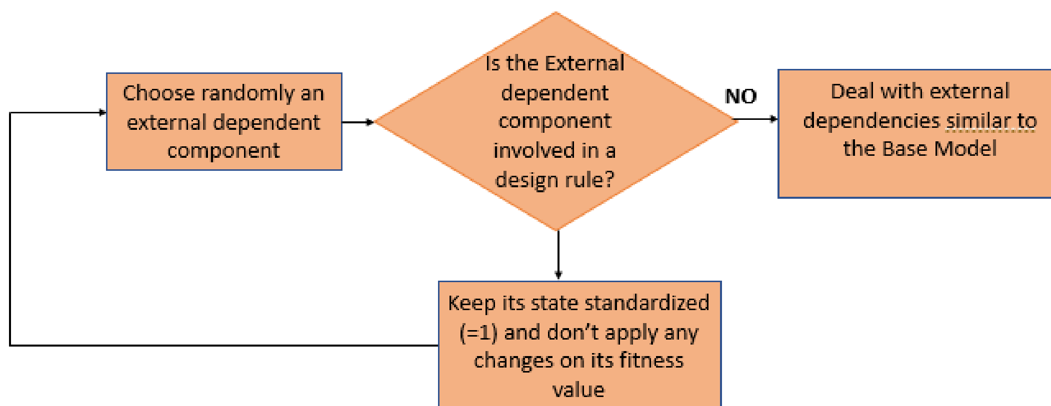


Figure 29: Design Rules Model (5)

Finally, we check whether the resolution of some systems forced already resolved external dependent components to become unresolved again. For this, an unresolved component ***i*** is chosen from a randomly chosen subsystem ***j***, on which we apply Equation (9) and sample for its fitness from a uniform distribution U(0,1). Then, we adapt the states and fitness values of the internal dependencies of component ***i*** accordingly through applying Equation (9) on all internal, non-design rules, dependent components within the subsystem ***j***, and sample for their fitness values from U(0,1). After applying these changes, we check whether the state of component ***i*** has been resolved or not. If the new state of component ***i*** is 1, i.e. resolved, we go with the new subsystem ***j*** states' configuration and its corresponding fitness contribution, regardless if the new average fitness is better or not. Otherwise, if the state of the focal component ***i*** stayed unresolved, then we check if the new average fitness contribution of subsystem ***j*** is better than the old one and go with the states' configuration and corresponding fitness contribution having higher average fitness. After we finish with component ***i***, we deal with all unresolved states within subsystem ***j***, until it becomes resolved. Then, we choose another subsystem ***j'***, having unresolved components, at random, until we have all subsystems within the whole system totally resolved.

The main aim of introducing the design rules concept into our model is to reach system resolution with the least needed cost and effort during the search process. In the Appendix, we show the full flowchart, of the extended-design rules model, in Figure A2.

CHAPTER 4

ANALYSIS AND DISCUSSION

In this chapter, we perform several sensitivity analyses using test cases (i.e. experiments) to improve our understanding of the dynamics of the model. First, we investigate product performance and convergence time in various systems having different number of components N and different architectures (i.e., decompositions); that is, different values of S and K. Then, we study the variation in the unresolved components (i.e., tasks) due to varying the dependency value $\boldsymbol{\beta}$. Moreover, we check how the average performance and average convergence time of systems with different decompositions are impacted by the value of $\bar{\boldsymbol{t}}$**.** We also investigate how the average performance and convergence time is impacted using design rules. In the last section of this chapter, we perform sensitivity analysis on the learning procedure as we examine the variation of the product performance and convergence time with the variation of the reinforcement learning parameter Φ.

4.1 Investigation of product performance and convergence time with different *N* and *S*

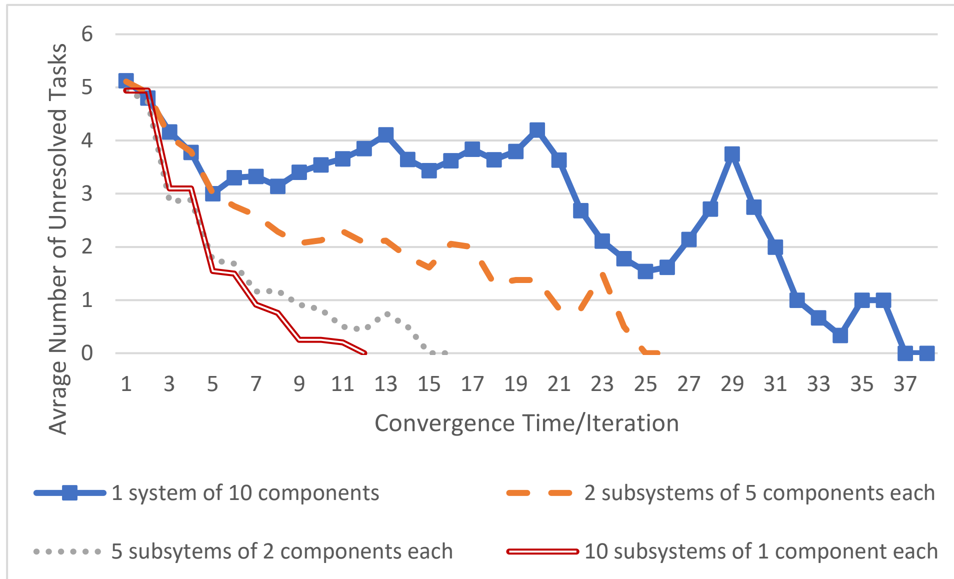
In this section, we examine the variation of the average number of unresolved tasks (i.e., process performance) and the average (product) performance (i.e., fitness values) of three systems, each having a different number of components **N.** We assume that all required resources to reach a resolution are available (i.e. the completion rate $\boldsymbol{r_i}$ =1 for all subsystems), and that the dependency value $\boldsymbol{\beta}$ is fixed to 0.15. The value of $\boldsymbol{\beta}$ is constant and not variable as stated in the Model section, per Equation (10), since learning is suspended in this section so that the effect of

system decomposition will not be tainted by the learning feature. As for the value of $\bar{t}$, we fixed it to be 1 in this section. Moreover, for each system with number of components **N,** we divide the whole system into different number of subsystems (blocks) to examine whether different decompositions have any effect on the average product and process performance (i.e. the number of unresolved tasks and fitness values).

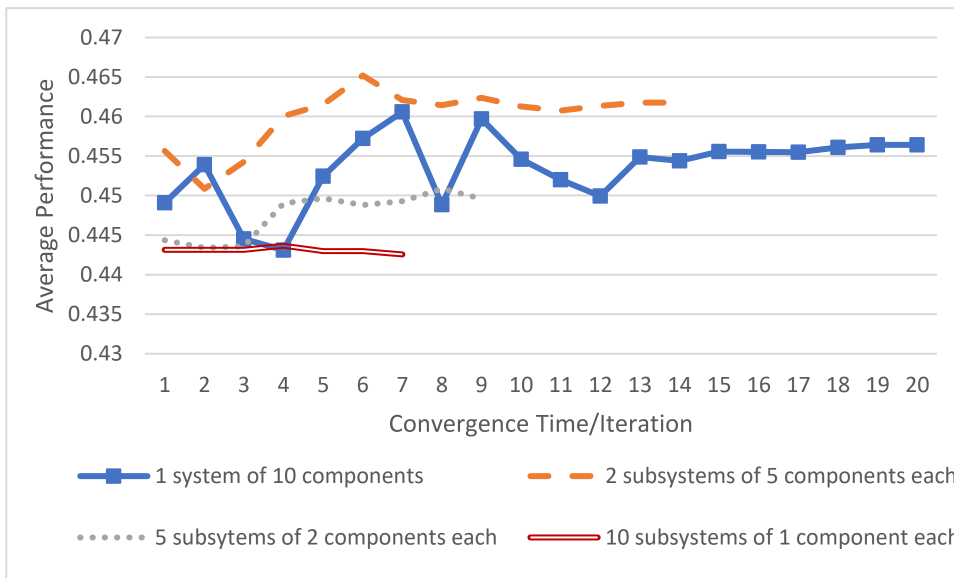
We first consider a system of ten components and make different system decomposition: one subsystem of ten components, two subsystems of five components each, five subsystems of two components each and 10 subsystems of 1 component each. Then, we simulate our base model in each of these four different decompositions to obtain the results shown in Figure 30.

The same simulation is repeated on a system of twenty components, where we take into consideration six cases: one subsystem of twenty components, two subsystems with ten components each, four subsystems with five components each, five subsystems with four components each, ten subsystems with two components each and finally twenty subsystems with one component each. Simulation results are shown in Figure 31.

Finally, we study large systems, of fifty components and take into consideration six cases: one subsystem made up of fifty components, two subsystems with twenty-five components each, five subsystems with ten components each, ten subsystems with five components each, twenty-five subsystems with two components each and finally fifty subsystems of one component. The variation of the number of unresolved tasks and fitness values of these six scenarios are shown in Figures 32.

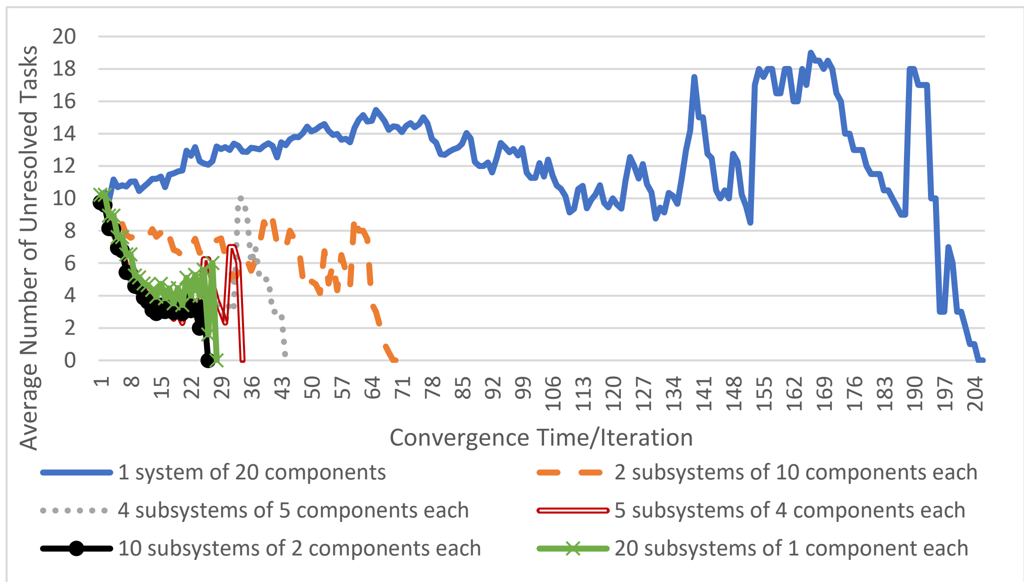


(a) Average Number of Unresolved Components / Tasks

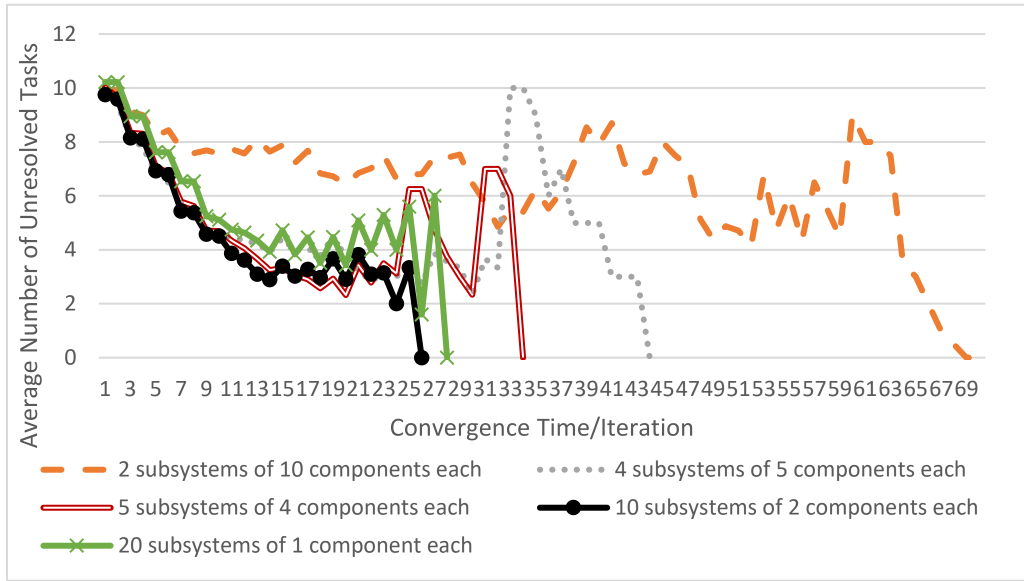


(b) Average Performance (i.e. fitness)

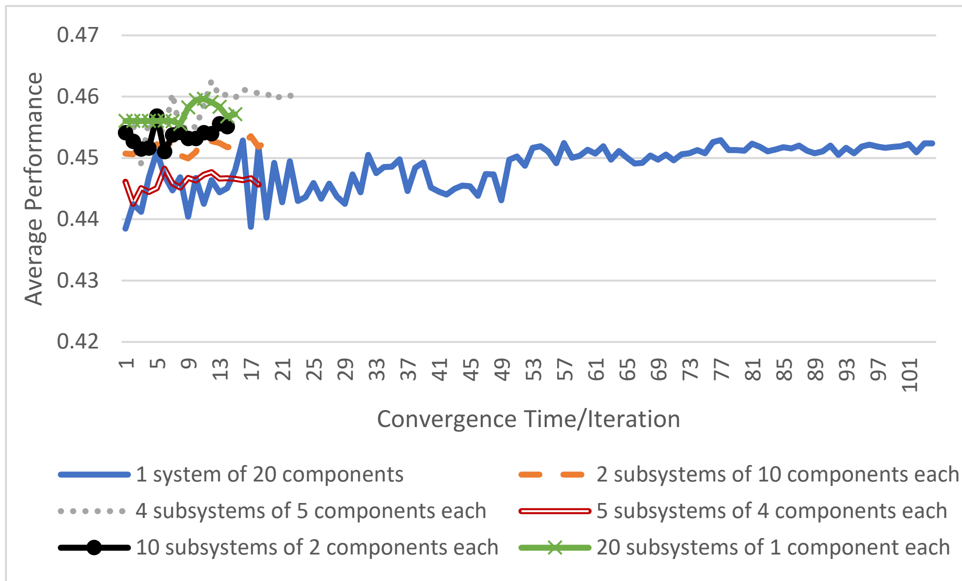
Figure 30: Variation of the number of unresolved tasks and average performance with time for a system of N=10



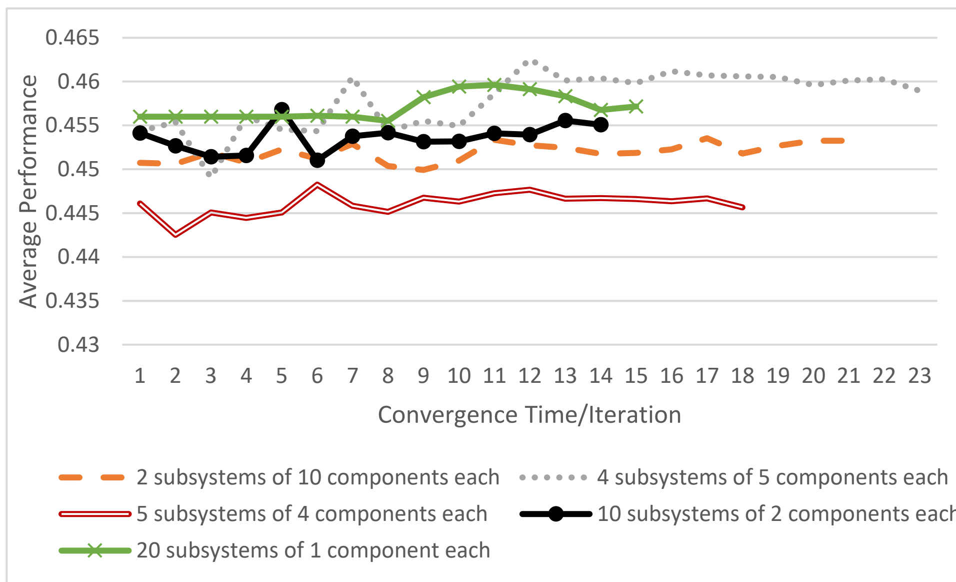
(a) Average Number of Unresolved Components / Tasks



(b) Average Number of Unresolved Tasks (without the 1 system with 20 components)

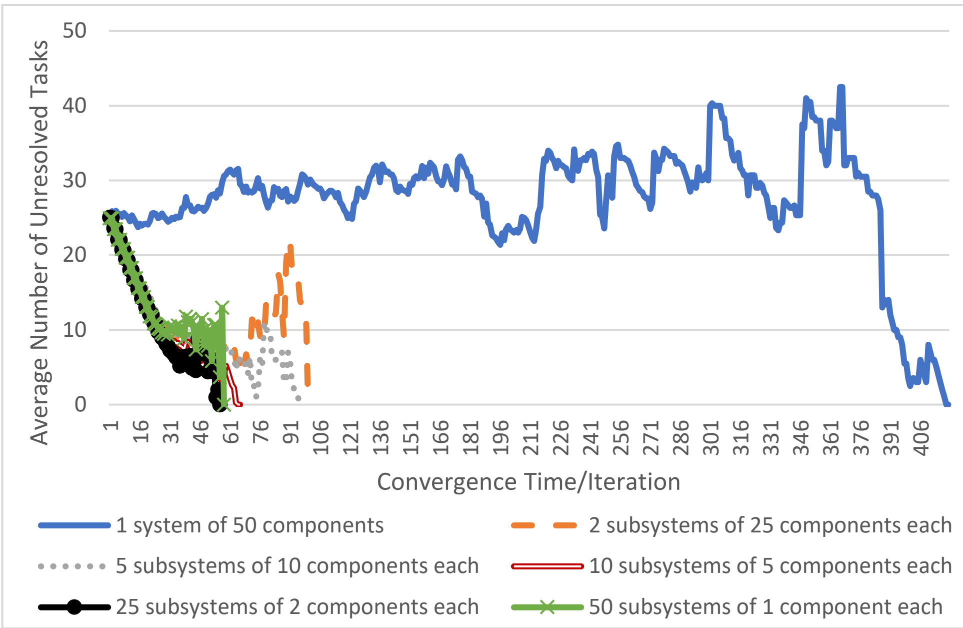


(c) Average Performance (i.e. fitness)

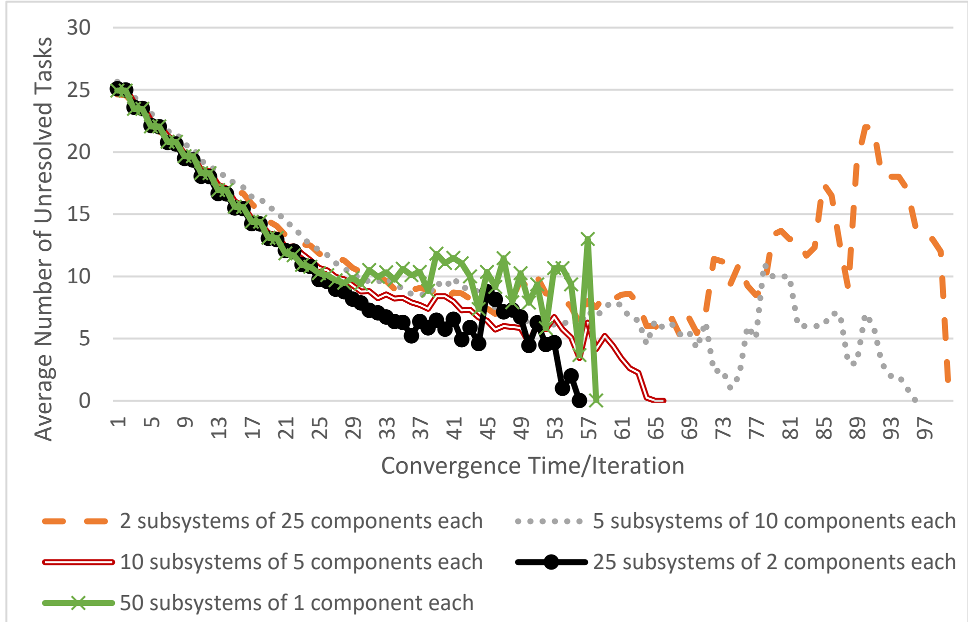


(d) Average Performance (i.e. fitness) (without the 1 system with 20 components)

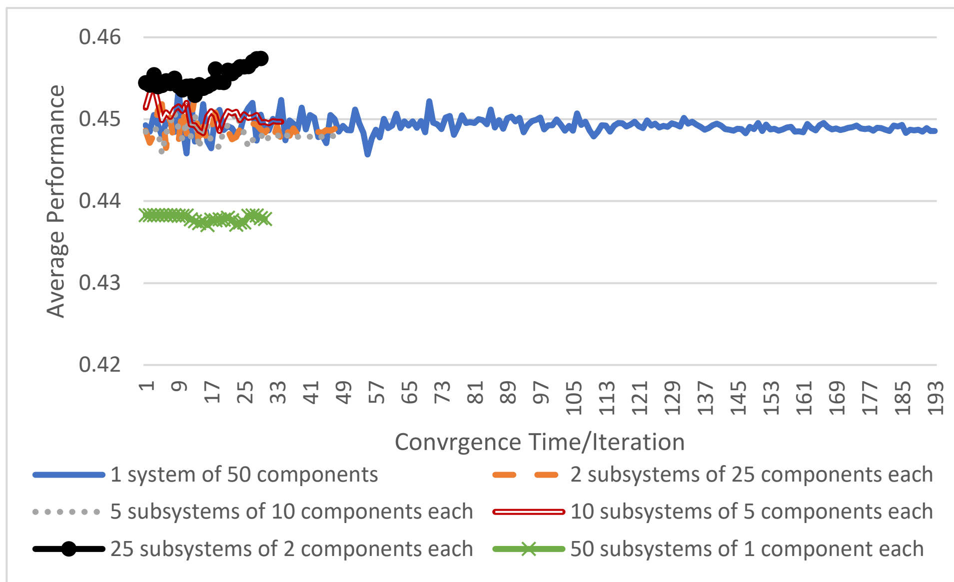
Figure 31: Variation of the number of unresolved tasks and average performance with time for a system of N=20



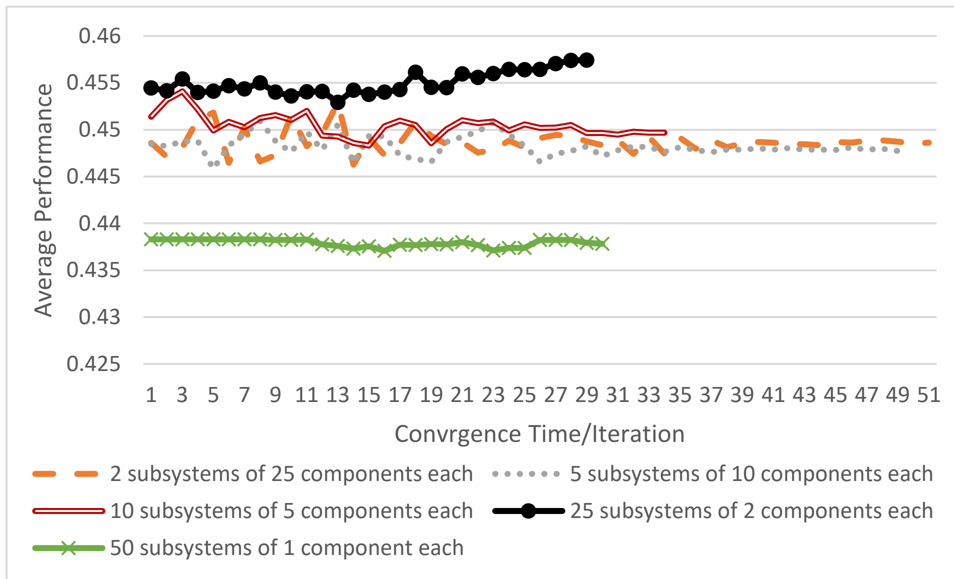
(a)Average Number of Unresolved Components / Tasks



(b)Average Number of Unresolved Tasks (without the 1 system with 50 components)



(c)Average Performance (i.e. fitness)



(d)Average Performance (i.e. fitness) (without the 1 system with 50 components)

Figure 32: Variation of the number of unresolved tasks and average performance with time for a system of N=50

We can observe that the average number of unresolved tasks (components), in all three systems, decreases along the simulation process, which means that the system converges to a solution (i.e. reaches resolution). It is also noticed that the number of unresolved tasks is not monotonically decreasing. It goes down then up and then down, several times, before eventually converging to zero, a phenomenon described earlier as design churn (Yassine et al. 2004). The resolution of

tasks in small systems (N=10), as shown in Figure 30 (a), is smooth and achieved fast convergence due to the small "complexity" in the system. Here, complexity is reflected by the number of dependencies in the system, denoted by the parameter ***K,*** defined earlier in our model. The number of dependencies ***K*** is changing in response to varying the system decomposition; that is, the number of subsystems (sub-blocks) ***S*** is considered during system decomposition. For example, in our case, when a system having N=10 components is decomposed into two subsystems, each of five components, the number of inner dependencies K_{in} is =3 and that of outer dependencies K_{out} is=1. However, when the same system is decomposed into five subsystems, with each having two components, the number of dependencies automatically change to become K_{in} is =1 and K_{out} is=3. As such, the complexity of the system consequently changes when changing the system decomposition.

As for the medium sized systems (N=20), as shown in Figure 31, the variation of the components' states between resolved and unresolved is greater than that in smaller systems, as some resolved components are revisited and are forced to become unresolved again until complete system resolution is reached. This is expected as more dependencies occur in larger systems, which renders the search and resolution processes harder to achieve.

Finally, in large sized systems (N=50) we obtain the widest variations to reach convergence, as it is shown in Figure 32. As the number of components in a system increases, the trend and variation of the number of unresolved tasks is more rigid and less smooth and it becomes more time consuming to reach system resolution. Comparing the average number of unresolved tasks between different system's decomposition, for the fixed number of components ***N,*** we significantly observe that as we divide the whole system into more subsystems (sub-blocks), it takes a smaller number of iterations to reach system resolution. For example, in Figure 32, when

we had one whole system of 50 components, system resolution was achieved after 420 iterations. As for the other extreme case, when we divided the system into 50 subsystems, each having only one component, it took only 58 iterations to achieve system resolution. This applies as well in small and medium sized systems, i.e. N=10 and N=20 respectively. Therefore, the higher the number of subsystems, the faster system convergence is reached. When we are dividing the system more and thus having a higher number of subblocks, system complexity is affected, i.e. the distribution of the number of inner and outer dependencies (K_{in} and K_{out}) changes, as discussed earlier. So, as the number of subsystems increase, having a lower number of components within each subsystem (subblock), the number of inner dependencies K_{in} is decreasing while that of the outer dependencies K_{out} is increasing (conserving the total number of $K= K_{in}+ K_{out}$), which makes the system complexity highly dependent on or represented by the external, rather than the internal, dependencies. Hence, we conclude that system convergence is mainly impacted by the complexity within the subsystems internally more than that represented by the external interactions and that "when teams give more weight to paying attention to their own subsystem than to between-subsystem interactions, they concentrate their search efforts on small search domains, and converge to a local optimum in a short time period" (Songhori et al., 2017).

As for the average performance values, as shown in Figures 30, 31 and 32 that average fitness values are almost stable where they vary between values of 0.44 and 0.47 maximum. Comparing how the fitness values are varying according to the system decomposition, in small sized systems (N=10), we reach the highest performance when the system is decomposed into two subsystems, each with five components, whereas the lowest performance is reached in this system when we decompose it into ten systems, each with one component, as shown in Figure 30 (b). As for

medium sized systems (N=20), the highest performance is achieved when we decompose the system into four subsystems, each with five components, where the system reaches an average fitness of 0.46. As for the lowest performance, it mainly occurs in two decompositions: one system, each with twenty components in which the system performance remains low, in a range between 0.44 and 0.45 , despite the several iterations the system passes through in the resolution process, as shown in Figure 31 (c and d), and when we decompose the system into five systems, each of four components. Finally, in large systems (N=50), the lowest performance is reached when we decompose the system into fifty subsystems, each with one component, as shown in Figure 32 (c and d), where the highest fitness values are when the system is decomposed into twenty five subsystems, each with 2 components. All in all, we conclude that system decomposition has an impact on the system performance, where as shown in Figures 30 (c and d), 31 (b) and 32 (c and d), one decomposition may cause the system to have the lowest performance at the time this performance increases significantly when we decompose the system to a different number of subsystems and components ($\boldsymbol{S}$, $\boldsymbol{K_{in}}$ and $\boldsymbol{K_{out}}$).

4.2 Divergence of the Number of Unresolved Tasks

As discussed in the previous section, the number of unresolved tasks in the system vary along the convergence time until resolution is reached. However, the system might not converge to achieve complete resolution under certain conditions. As proved by Braha and Bar-Yam (2007), convergence is achieved if the result of multiplying β with the number of unresolved components ($\beta_i * y_i$) is less than the completion rate r_i , in the system as a whole**.**

To investigate this case, we performed a sensitivity analysis by varying the value of parameter $\boldsymbol{\beta}$ in one of the scenarios of each of the above subsystems (N=10, 20 and 50), keeping the value of

$\bar{t}$ and $\mathbf{r_i}$ fixed at 1. In this case, we applied the analysis on a system of N=10 with five subsystems, each of two components, on a N=20 system divided into five subsystems each of four components, and on a N=50 system representing five subsystems each with ten components. The results are respectively shown below in Figures 33, 34 and 35.

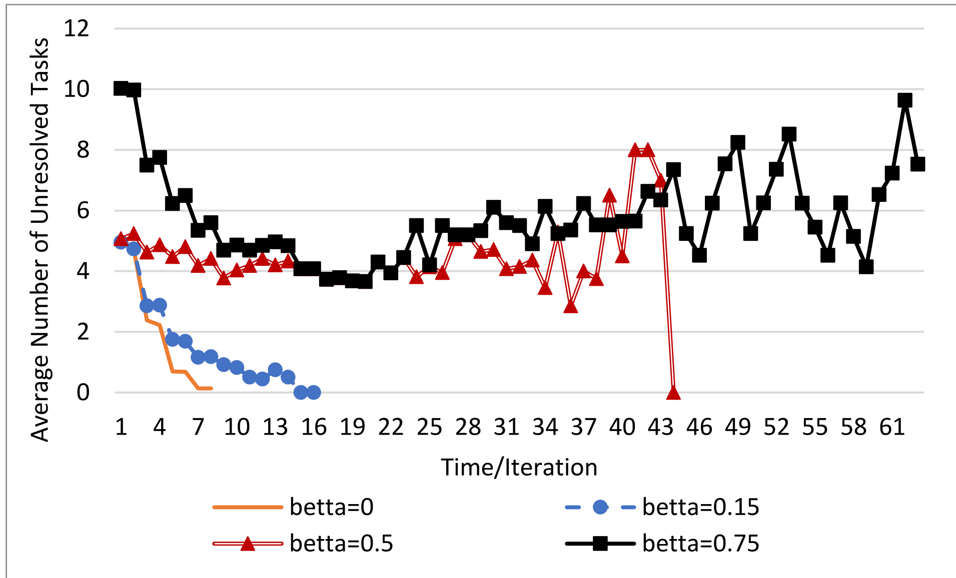


Figure 33: Variation of the number of unresolved tasks with time for a system of N=10 for different β values

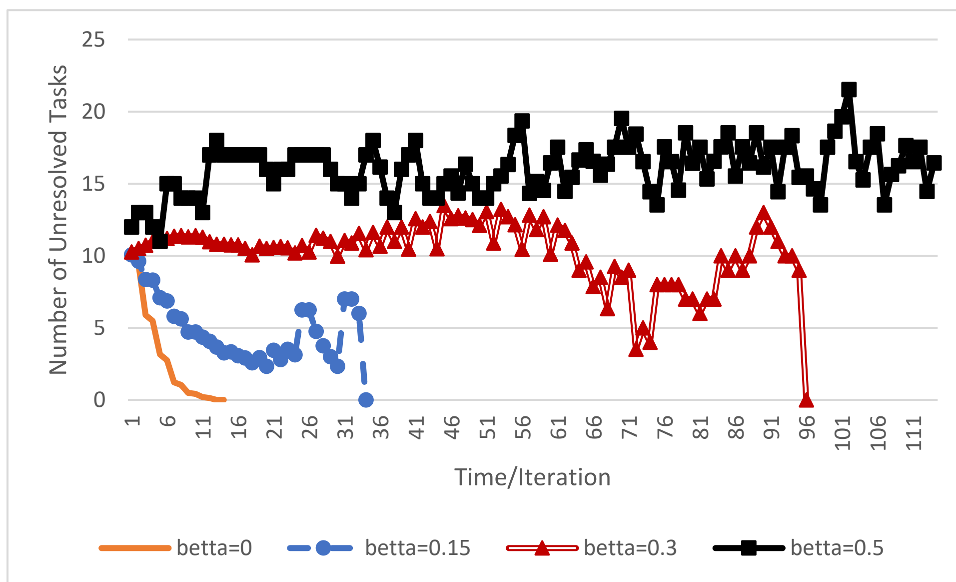


Figure 34: Variation of the number of unresolved tasks with time for a system of N=20 for different β values

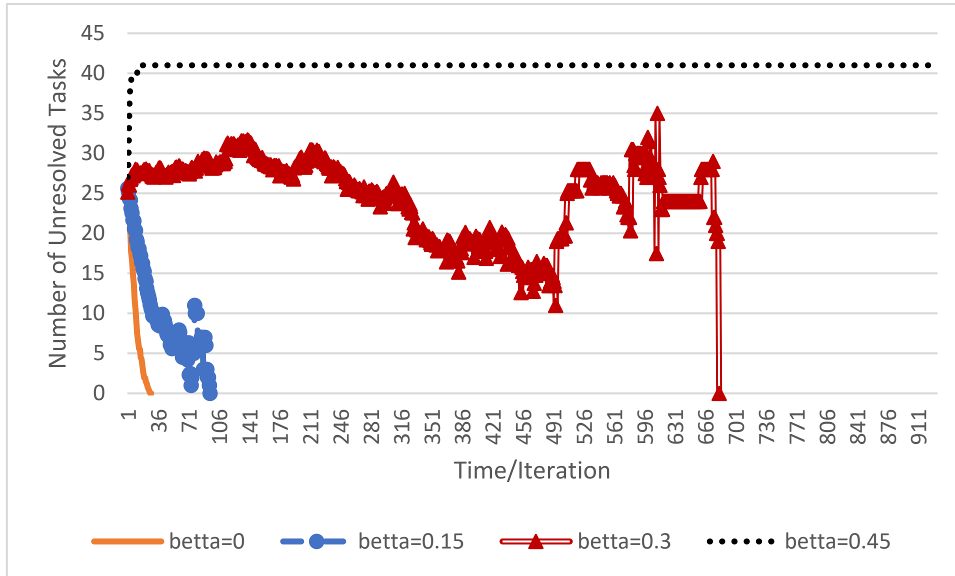


Figure 35: Variation of the number of unresolved tasks with time for a system of N=50 for different β values

As we can see from Figures 33, 34 and 35, each system has a specific value for $\boldsymbol{\beta}$ at which it diverges. For example, at $\boldsymbol{\beta} = 0.75$, the small sized systems (N=10) diverge, unlike the medium sized system (N = 20) which diverges at $\boldsymbol{\beta} = 0.5$. As for the large sized systems (N = 50), we can see from Figure 35 that they diverge at $\boldsymbol{\beta} = 0.45$. So, we conclude that as the number of components in a system increases, it becomes more possible to diverge at lower values of the sensitivity parameter $\boldsymbol{\beta}$. For this, when the system becomes very large (≥500 component), it will eventually diverge at small values of $\boldsymbol{\beta}$, which makes it difficult for very large systems to reach resolution unless a proper and careful decomposition is applied to the system.

It is worth checking whether system decomposition has any effect on the system convergence, i.e. if a system converges, in case it is decomposed into subsystems, will it convergence in case it wasn't decomposed?

First, we assume a system A, decomposed into ***N*** subsystems, where each subsystem ***i*** converges and has its own number of unresolved components, resembled by y_1, y_2 ...., y_N respectively. As for the values of the completion rate *r* and β, they are constant in all subsystems.

According to Braha's equation, and since each subsystem converges, this validates the following set of ***N*** formulas, in each of the subsystems:

$y_1 < \frac{r}{\boldsymbol{\beta}}$ ... (1)

$y_2 < \frac{r}{\boldsymbol{\beta}}$ … (2)

……..

$y_N < \frac{r}{\boldsymbol{\beta}}$ … (N)

Now, we combine the ***N*** subsystems into one system A, and we check if the system converges as a whole at the dependency value $\boldsymbol{\beta}$, or not. Adding the two sides of the above set of the ***N*** equations, we obtain the below:

➔ $(y_1 + y_2 + \ldots + y_N) < (\underbrace{\frac{r}{\boldsymbol{\beta}} + \frac{r}{\boldsymbol{\beta}} + \ldots \frac{r}{\boldsymbol{\beta}}}_{\text{N times}})$

➔ $(y_1 + y_2 + \ldots + y_N) < \frac{N*r}{\boldsymbol{\beta}}$ (To note here that the dependency value and completion rate are same and fixed in all subsystems)

➔ $y_A < \frac{N*r}{\boldsymbol{\beta}}$

So here we have two scenarios: $y_A < \frac{r}{\boldsymbol{\beta}}$ or $y_A > \frac{r}{\boldsymbol{\beta}}$

Then, system A here can either diverge or converge at the time it was converging when divided into smaller subsystems. Hence, we conclude that system decomposition plays an important role in system convergence in which a system can converge when dividing it into smaller subsystems,

where each subsystem converge under the following condition: $y < \frac{r}{\boldsymbol{\beta}}$, at the time the system was diverging as a whole.

To validate the proof and check whether system decomposition has any effect on system divergence, we simulate the large sized system, of fifty components, at $\boldsymbol{\beta}$ =0.3 over 100 runs, in two different cases: one system of fifty components and two subsystems, each with twenty-five components. The simulation of these two cases resulted in the divergence of the subsystem when it is undecomposed; that is, one subsystem with fifty components. It converges, at an average of 186 iterations, when we decompose it into two subsystems, each with twenty-five components. The results of these two cases are shown in Figure 36. To note here that the system diverges, in all its decompositions at $\boldsymbol{\beta}$ =0.35, 0.4 and 0.45. Thus, we conclude that system decomposition affects divergence at certain values of $\boldsymbol{\beta}$. These values of $\boldsymbol{\beta}$, at which the system converges when decomposing it at the time it diverges as a whole, can be determined through a trial and error process. To minimize the options of the $\boldsymbol{\beta}$ values in the trial and error method, we first solve for the $\boldsymbol{\beta}$ value in equation $y_A < \frac{r}{\boldsymbol{\beta}}$, i.e. $\boldsymbol{\beta} < \frac{r}{yA}$. By this, we are assuring that the system as a whole is converging, so we start increasing the obtained value of $\boldsymbol{\beta}$ to reach a value at which the system as a whole diverges. After obtaining the $\boldsymbol{\beta}$ value at which the whole system diverges, we decompose it into ***N*** subsystems and start testing if the decomposed system will converge or not at the obtained $\boldsymbol{\beta}$ value and so on to obtain a converging decomposed system using the trial and error process.

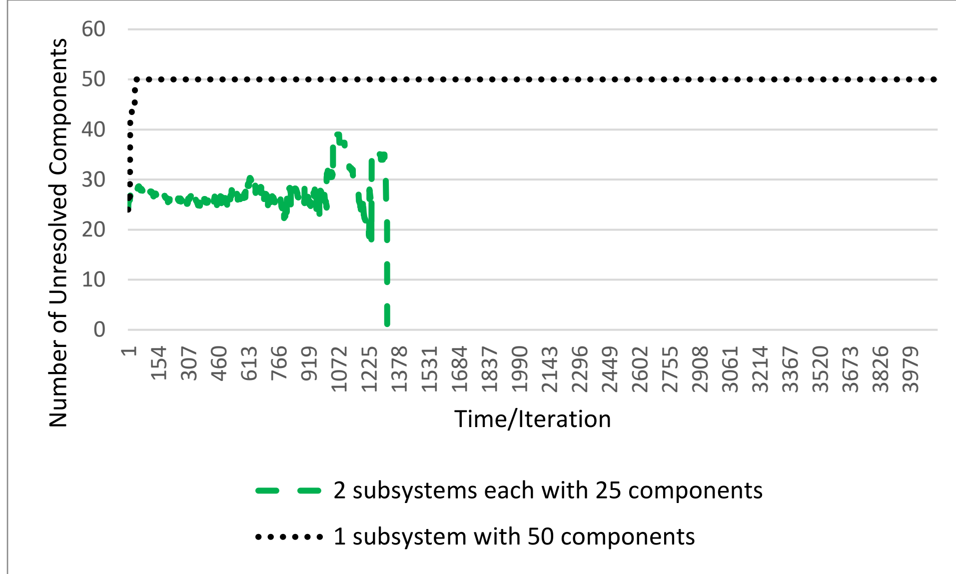


Figure 36: Variation of the number of unresolved tasks with time for two different systems' decompositions of N=50 at β=0.3

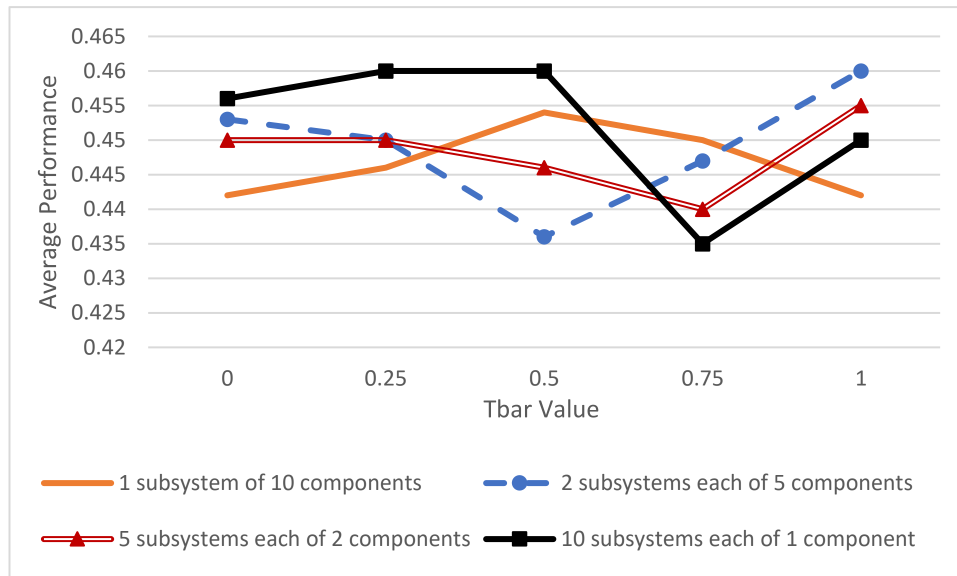
4.3 Sensitivity Analysis of the threshold value $\bar{t}$

In this section, we test the variation of the average convergence time and average performance values, over 100 runs, as a function of the threshold value $\bar{t}$. We perform this test on small, medium and large-sized systems (N=10, 20 and 50) and results are obtained in Figures 37, 38 and 39 respectively. As mentioned in section 4.1, we assume that the completion rate for all subsystems $\mathbf{r_i}$ is 1 and that the sensitivity value $\boldsymbol{\beta}$ is fixed at 0.15.

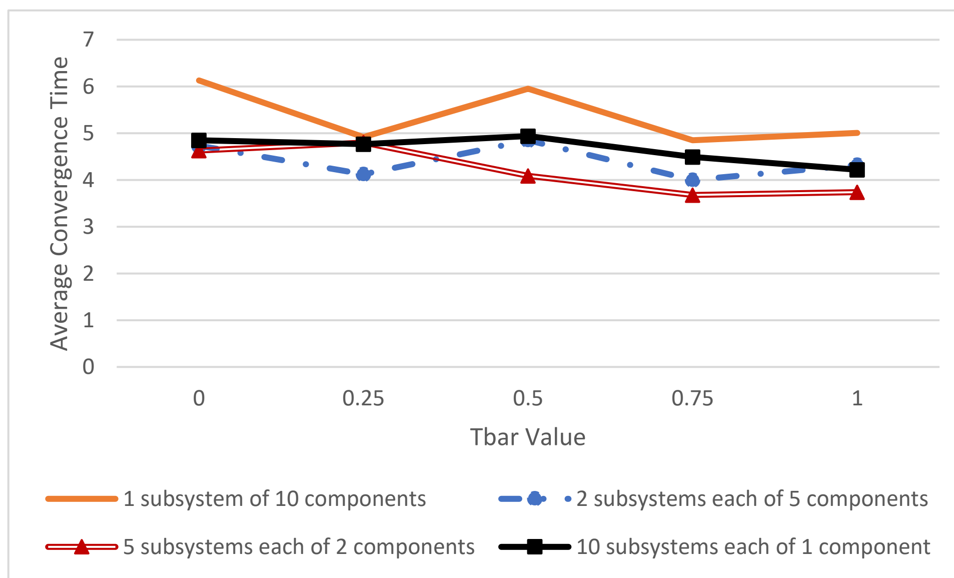
We notice that as the value of $\bar{t}$ increases from 0 to 0.5, i.e. when the probability of dealing with external dependencies increase, almost in all systems, with different $\boldsymbol{N}$ and different corresponding decomposition, the number of iterations increases. Then, as the value of $\bar{t}$ continues to increase from 0.5 to 1, at which we always deal with all external dependencies, the average convergence time stays the same.

As for the fitness values, it can be shown that the trend of all average fitness values is similar and vary in a range between 0.44 and 0.46. Investigating how the fitness values vary according to the system decomposition, it can be shown from Figures 37 (a), 38 (a) and 39 (a) that average performance values are not impacted directly by the system decomposition. For example, in small sized systems (N=10), highest performance is achieved when we decompose

the system into ten subsystems, with one component each, when $\bar{t}$ is between 0 and 0.5. However, as $\bar{t}$ increases from 0.5 to 1, the system decomposed into ten subsystems, with one component each, tends to have the lowest performance values. Similar analysis is observed in medium (N=20) and large (N=50) sized systems, where there is no specific system decomposition which has the highest/lowest performance, across all values of $\bar{t}$. Hence, according to our simulation results, it can be concluded that system decomposition does not matter for the average fitness values as we vary $\bar{t}$.

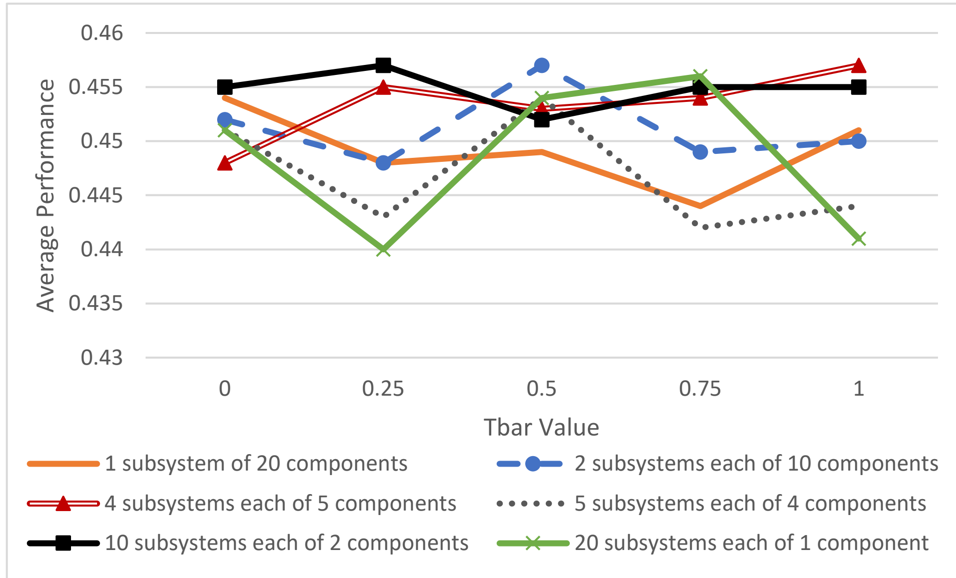


(a)Average Performance (i.e. fitness)

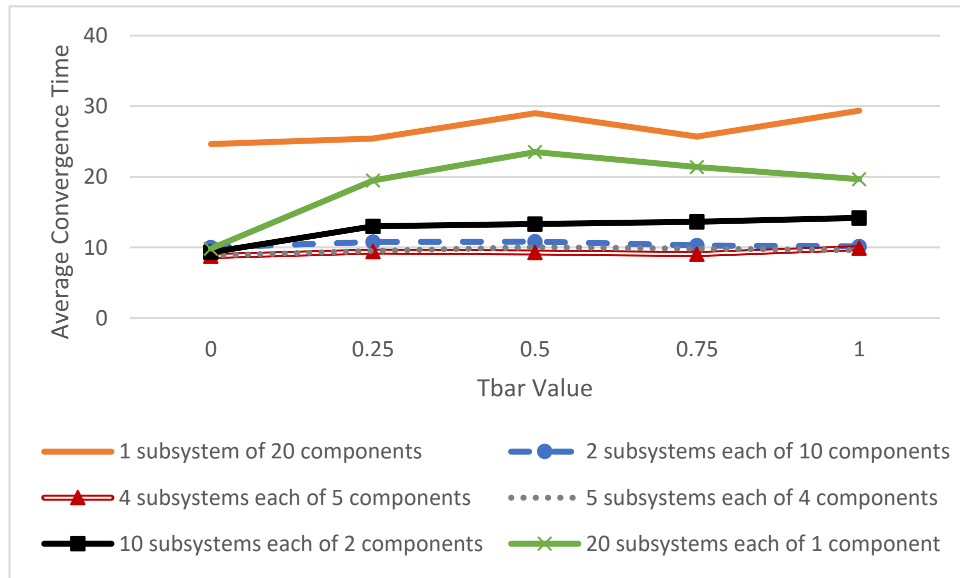


(b)Average Convergence Time

Figure 37: Variation of the average fitness values and average number of iterations as a function of t bar for N=10

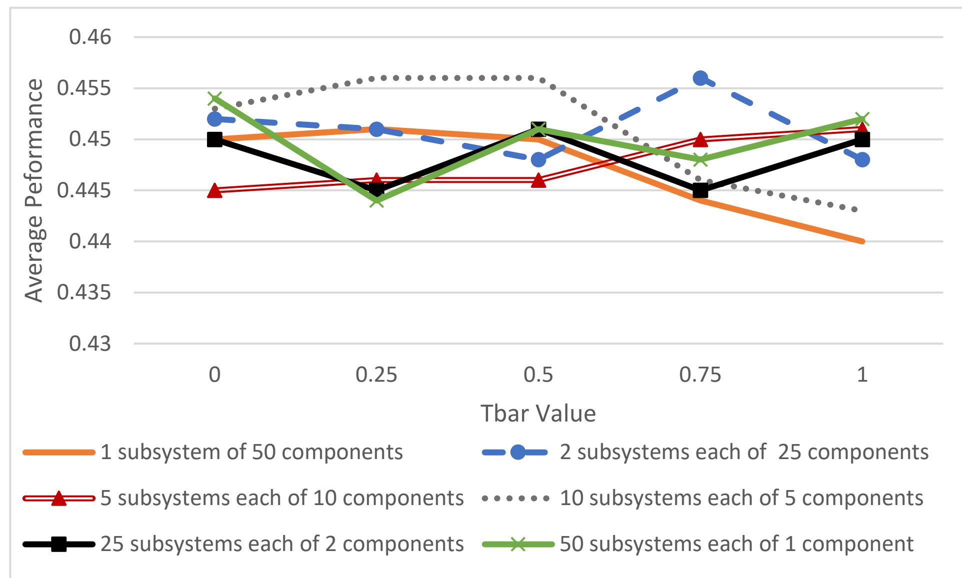


(a)Average Performance (i.e. fitness)

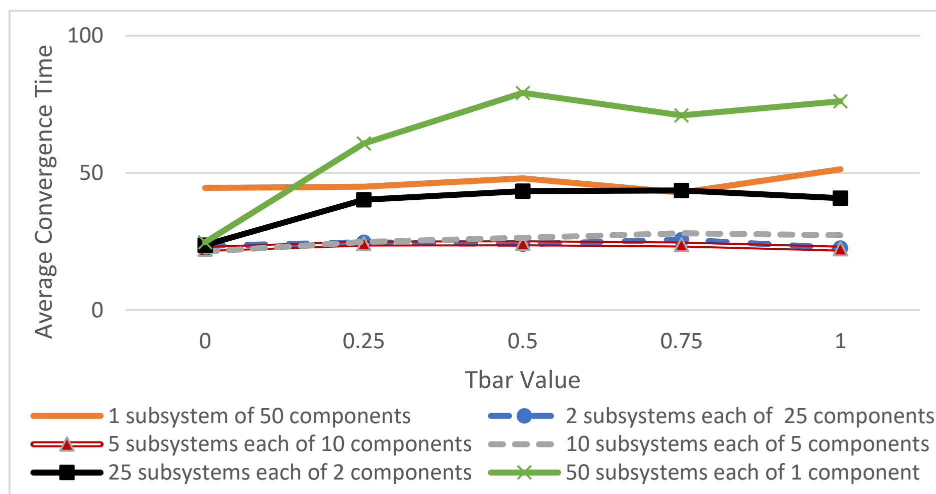


(b)Average Convergence Time

Figure 38: Variation of the average fitness values and average number of iterations as a function of t bar for N=20



(a)Average Performance (i.e. fitness)



(b)Average Convergence Time

Figure 39: Variation of the average fitness values and average number of iterations as a function of t bar for N=50

4.4 Investigation of product performance and convergence time considering Design Rules

To study the effect of design rules on the system convergence and fitness (i.e., product and process performance), we examine the variation of the needed number of iterations to reach a

resolution (i.e. convergence) as a function of the number of developed (i.e. implemented) design rules. We simulate this experiment on a system of twenty components, decomposed into five subsystems with four components each, where $\boldsymbol{K_{in}} = 3$ and $\boldsymbol{K_{out}} = 4$, as represented in the DSM of Figure 40. The cyclic dependencies of size 2, developed in this system are the off block-diagonal elements highlighted (in yellow) in the DSM, as shown in Figure 40. As for the rest of the dependencies, they either represent the external dependencies between components from different subsystems which cannot be developed as design rules, or higher order design rules, i.e. cyclic dependencies of size higher than 2. It should be noted that not all external dependencies can be developed as design rules, since, for example, if a component X depends on component Y, then component Y doesn't necessarily depend on component.

X	X	X	X			X					X		X			X			
X	X	X	X		X								X		X	X			
X	X	X	X		X			X							X	X			
X	X	X	X							X	X		X				X		
X		X	X	X	X	X	X					X							
				X	X	X	X		X		X					X	X		
				X	X	X	X				X					X	X	X	
				X	X	X	X	X			X					X			X
			X	X			X	X	X	X	X		X						
X	X	X		X				X	X	X	X								
	X				X		X	X	X	X	X			X					
		X				X		X	X	X	X	X		X					
		X	X	X		X						X	X	X	X				
			X	X					X			X	X	X	X		X		
						X			X	X		X	X	X	X		X		
	X			X		X			X			X	X	X	X				
	X				X			X		X						X	X	X	X
					X			X		X		X				X	X	X	X
			X					X		X		X				X	X	X	X
										X		X	X	X		X	X	X	X

Figure 40: Matrix representing a system of N=20 components, before the selection of design rules

To have a detailed view of these design rules, Table 10 represents the components between which these two-cyclic design rules are developed. It is worth noting here that the DSM in Figure 40 is symmetric.

Table 10: Two Cyclic Design Rules between Components

Design Rule #	Component #	Subsystem #		Component #	Subsystem #
1	2	1	With	4	4
2	2	1	With	1	5
3	4	1	With	2	4
4	1	2	With	1	4
5	2	2	With	1	5
6	2	2	With	2	5
7	3	2	With	4	3
8	4	2	With	1	3
9	3	3	With	3	4

After developing and selecting all the design rules that represent the dependency cycles of size two, shown in Table 10, these dependencies are removed from the original DSM (Figure 40) and the resultant DSM becomes as shown in Figure 41.

X	X	X	X			X					X		X			X			
X	X	X	X		X								X						
X	X	X	X		X			X							X	X			
X	X	X	X							X	X						X		
X		X	X	X	X	X	X												
				X	X	X	X		X		X								
				X	X	X	X									X	X	X	
				X	X	X	X				X					X			X
			X	X				X	X	X	X		X						

X	X	X		X				X	X	X	X								
	X				X		X	X	X	X	X								
		X						X	X	X	X	X		X					
		X	X			X						X	X	X	X				
				X					X			X	X	X	X		X		
						X			X			X	X	X	X		X		
				X		X			X			X	X	X	X				
								X		X						X	X	X	X
								X		X		X				X	X	X	X
			X					X		X		X				X	X	X	X
										X		X	X	X		X	X	X	X

Figure 41: Matrix representing a system of N=20 components, after the selection of design rules

Obtained simulation results are shown in Figure 42.

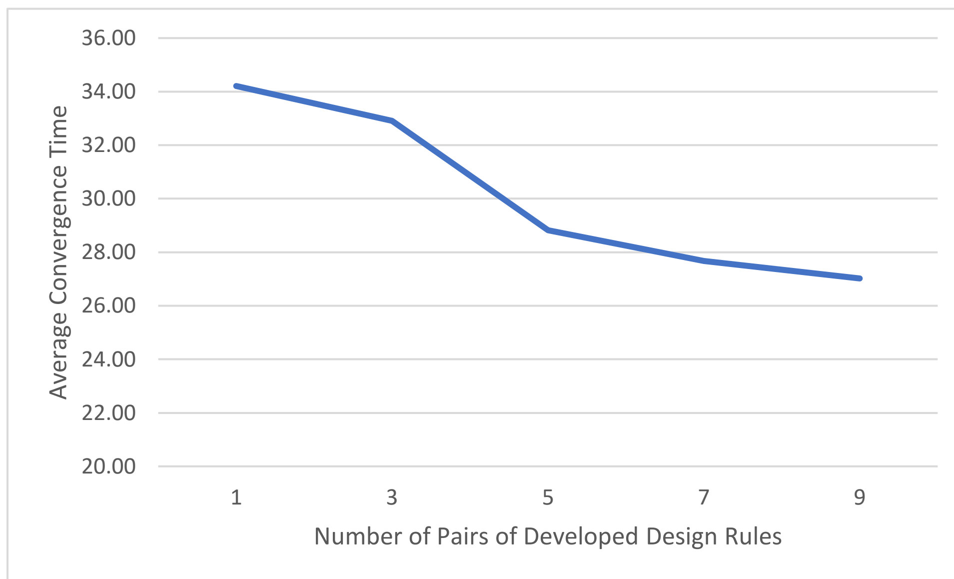


Figure 42 (a): First run showing the variation of the average number of iterations with the number of design rules developed

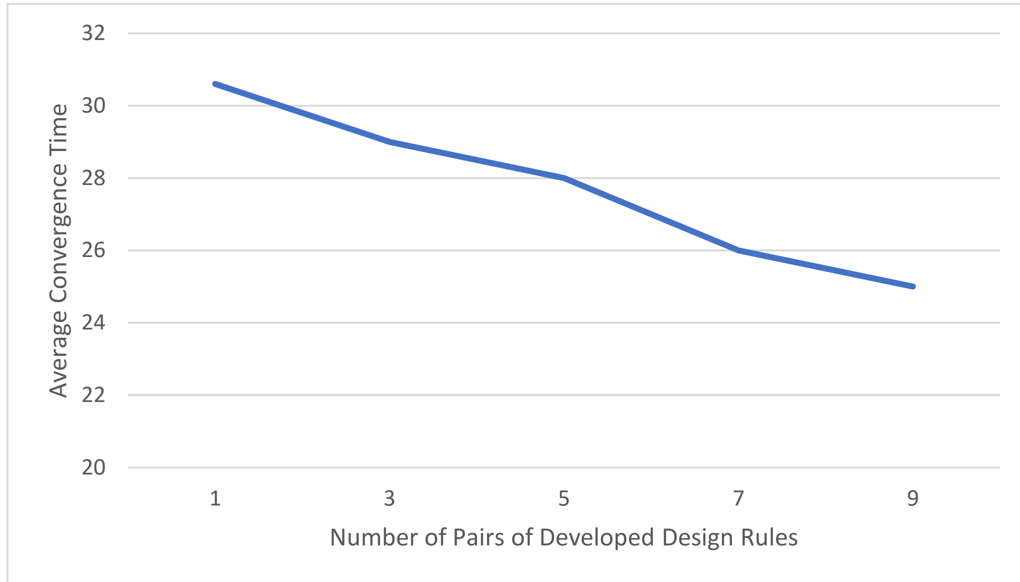


Figure 42 (b): Second run showing the variation of the average number of iterations with the number of design rules developed

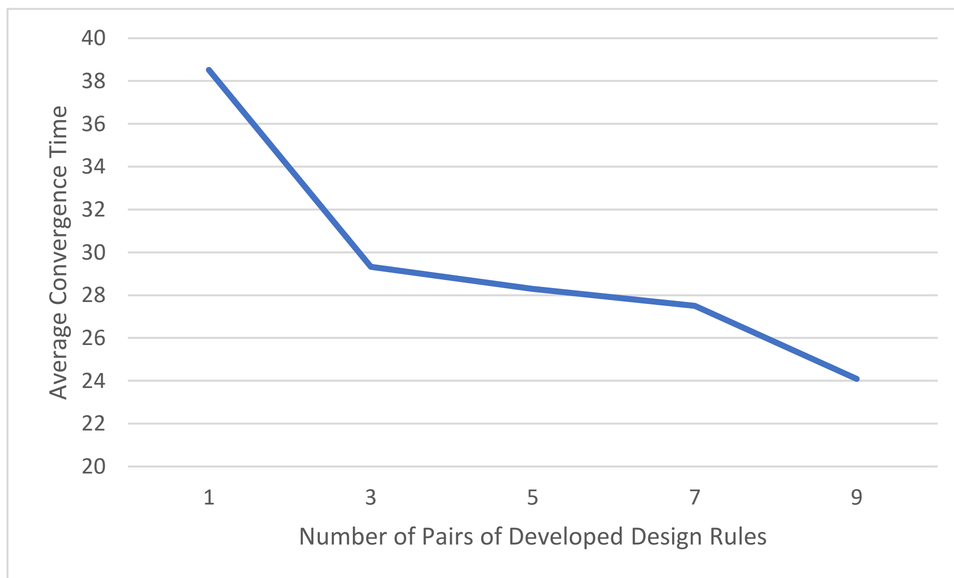


Figure 42 (c): Third run showing the variation of the average number of iterations with the number of design rules developed

As it can be shown from Figures 42(a), (b) and (c), the number of iterations decrease as the number of developed design rules increase (considering only dependency cycles of size two). It

should be noted here that the design rules selected in Figures 42 (a), (b) and (c) are selected randomly, and not in the order shown in Table 10 as the search algorithm chooses randomly the pairs of developed design rules, based on the specified number. This is because as we develop more design rules, a higher number of components are standardized, i.e. their corresponding state becomes "one", and thus no need to dedicate any effort (and / or resources) to resolve them. Therefore, we will need a smaller number of iterations to resolve all system's components, which will consequently reduce system complexity. To note here that the trend and linearity of the decrease of the number of iterations is not linked to a certain number of developed design rules. For example, in Figure 42 (a) we can see that number of iterations decreased significantly when increasing the number of developed design rules from three to five, unlike in Figure 42 (b) where we observed this significant decrease when increasing the number of design rules from five to seven. As for the third run, represented by Figure 42 (c), we notice that the number of iterations decreased notably when increasing the number of developed design rules from seven to nine.

In order to check the effect of design rules on both number of iterations and average fitness values, we apply a simulation of our model, without and with design rules respectively on a DSM of size 20 (i.e., N=20). To note here that we removed all design rules, that is all design rules with all cycles' size (cycles of 2, cycles of 3, etc..), and not only design rules with cycles of two, shown in Table 10. The obtained results are shown in Tables 11 and 12.

Table 11: Average Fitness Values with and without Design Rules

Average Fitness of 100 runs (N=20):						
Number of Pairs of Developed Design Rules	Subsystem 1	Subsystem 2	Subsystem 3	Subsystem 4	Subsystem 5	Average Fitness of the whole system
No Design Rules	0.474	0.450	0.436	0.471	0.487	0.464
All Design Rules	0.374	0.365	0.410	0.404	0.434	0.397

Table 12: Average Number of Iterations with and without Design Rules

Average Number of Iterations of 100 runs (N=20):						
Number of Pairs of Developed Design Rules	Subsystem 1	Subsystem 2	Subsystem 3	Subsystem 4	Subsystem 5	Average Iterations of the whole system
No Design Rules	7.833	6.167	7.000	7.583	6.167	**34.750**
All Design Rules	0.000	0.000	0.000	0.000	0.594	**0.594**

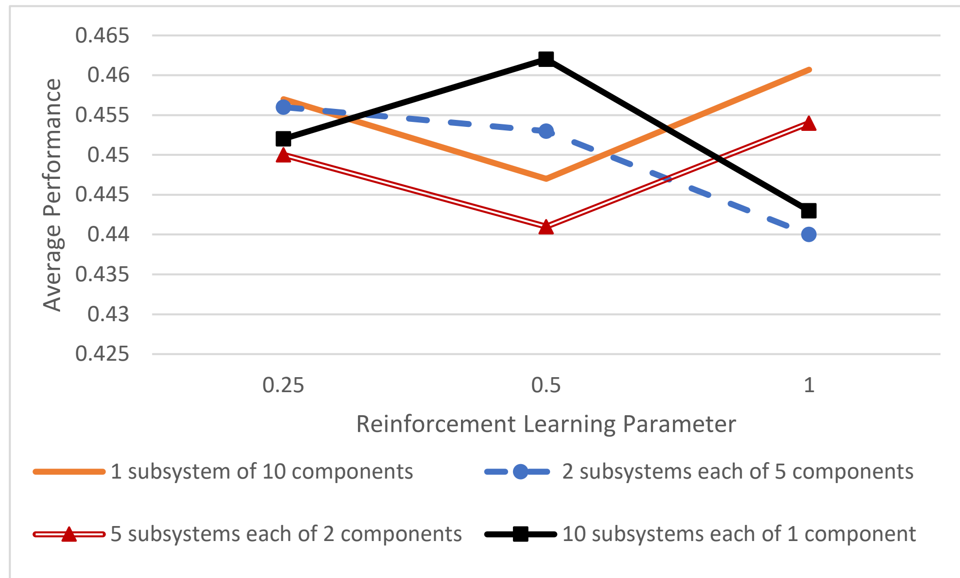
Tables 11 and 12 show the variation of the average fitness values and average number of iterations respectively in a N=20 system, with and without design rules. In Table 11, it can be shown that the average fitness value of the whole system, over 100 runs, before introducing design rules into the system was **0.464**. This value slightly decreases to become **0.397** when introducing design rules.

As for Table 12, we can observe that the average number of iterations decreases dramatically from **34.75** to **≈0.6**. From this, we can conclude that despite the design rules restrict the design freedom of the system's components, which may lead to a less fitness value; however, this occurs at the advantage of a huge reduction in the cost, i.e. needed number of iterations. It is noticed from Table 12 as well that the first four subsystems are completely resolved as everything is decided upon developing the design rules, unlike the case in subsystem 5 as it took ≈0.6 to become resolved.

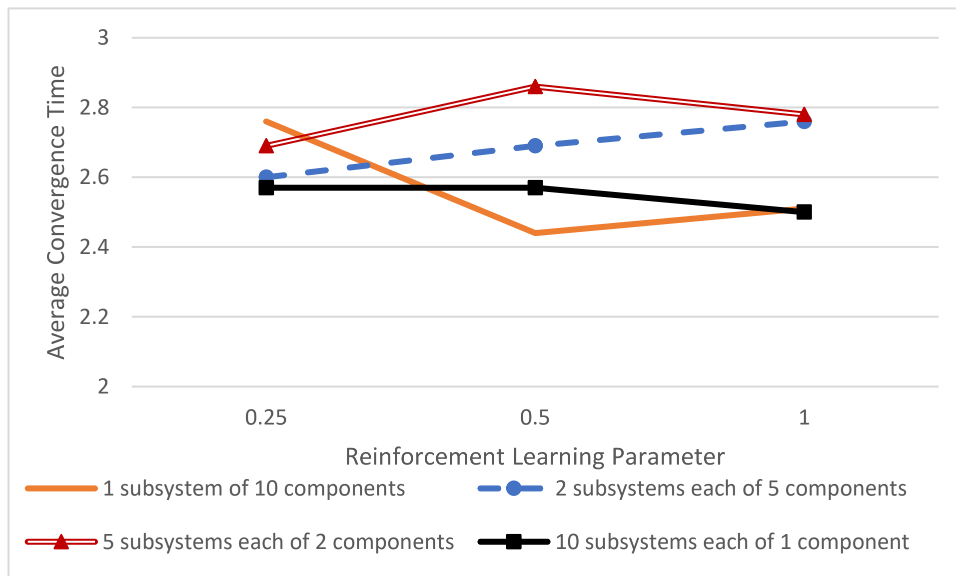
4.5 Investigation of product performance and convergence time and dynamics as a function of the reinforcement learning

In this section, we introduce reinforcement learning to investigate the variation of the product performance and convergence time as a function of the reinforcement learning parameter Φ. Similar to earlier sections, we perform this analysis on small, medium and large sized systems.

We fix the values of the completion rate $\mathbf{r_i}$ and $\bar{\mathbf{t}}$ to 1 in all our simulations, where the average is done over 100 runs. Results are shown below in Figures 43, 44 and 45.

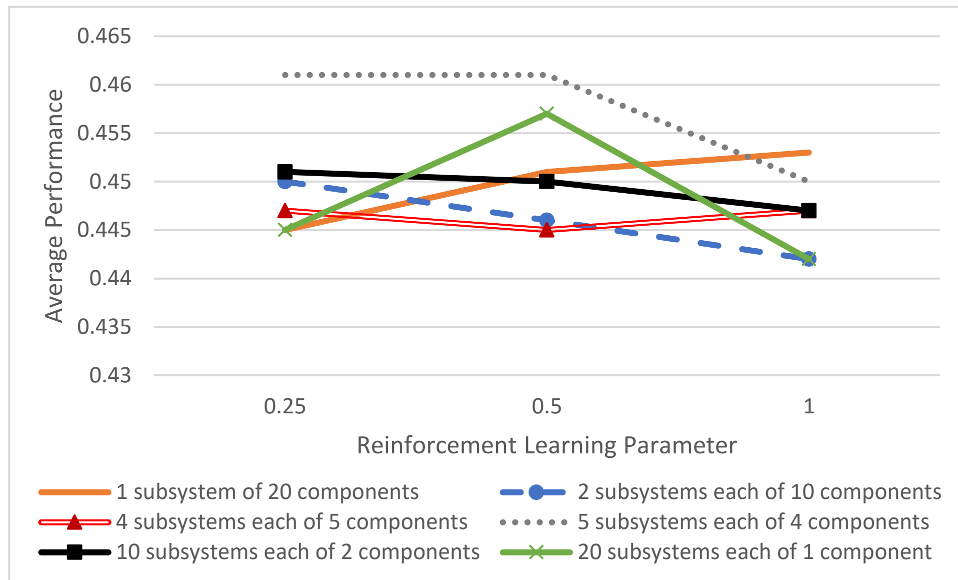


(a)Average Performance (i.e. fitness)

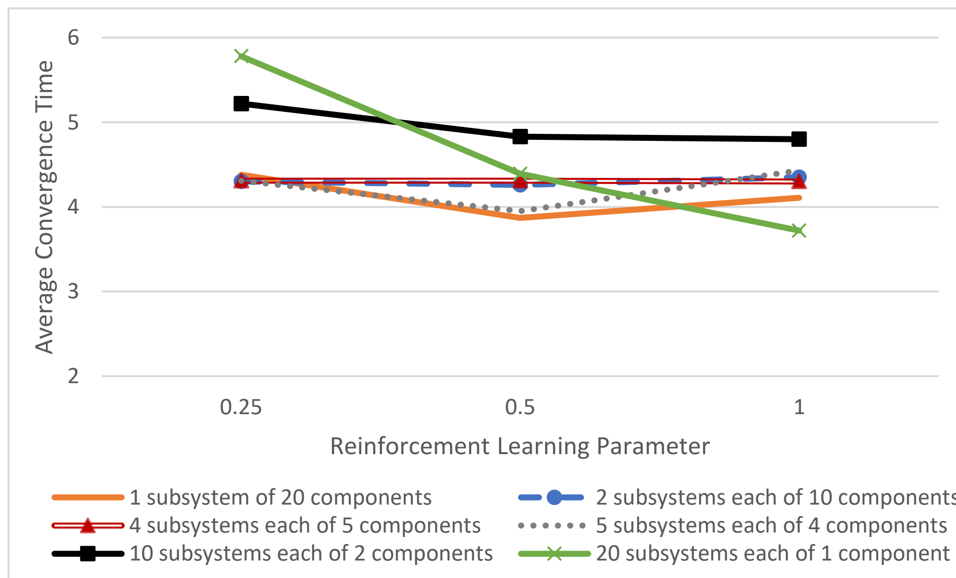


(b)Average Convergence Time

Figure 43: Variation of the average fitness values and average number of iterations as a function of Φ for N=10

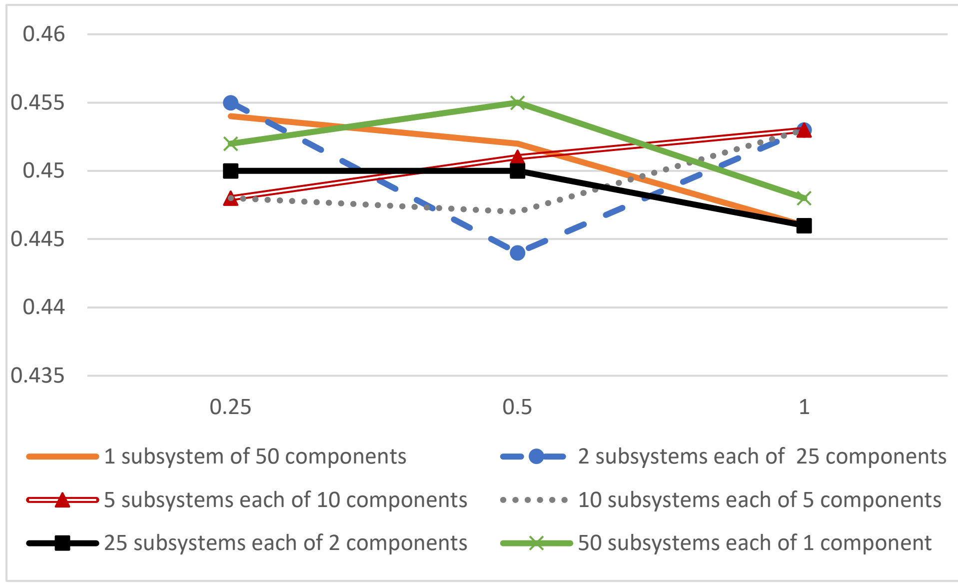


(a)Average Performance (i.e. fitness)

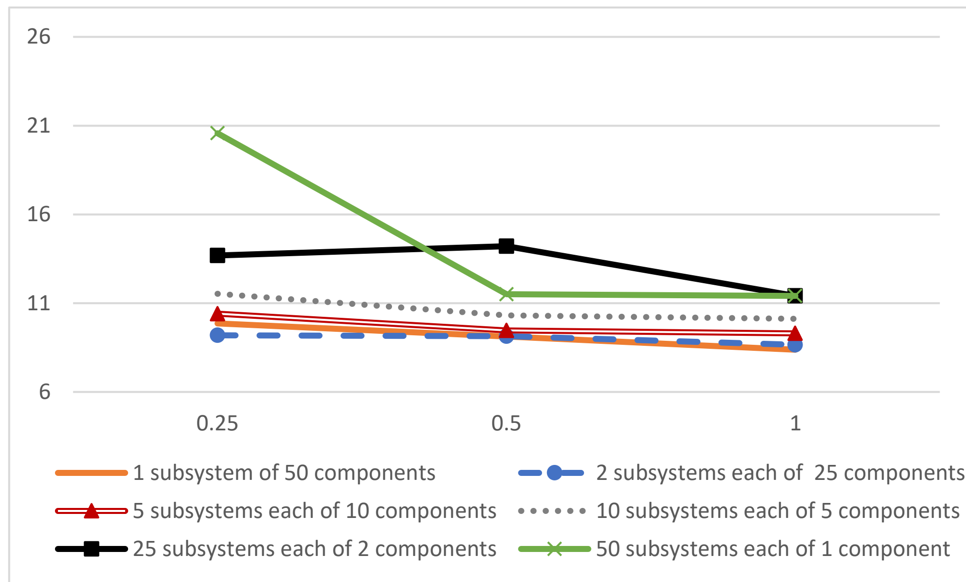


(b)Average Convergence Time

Figure 44: Variation of the average fitness values and average number of iterations as a function of Φ for N=20



(a)Average Performance (i.e. fitness)



(b)Average Convergence Time

Figure 45: Variation of the average fitness values and average number of iterations as a function of Φ for N=50

As shown from Figures 43, 44 and 45, we notice that the average convergence time to reach system resolution is decreasing as we increase the value of the reinforcement learning parameter Φ. In addition, when comparing the average convergence time for a fixed $\boldsymbol{\beta}$ value, (i.e. without

learning presented in section 4.3), and the average convergence time presented in this section after introducing the learning feature, we notice that the needed number of iterations to achieve system resolution has significantly decreased. For example, in large sized systems (N=50), when having a system divided into 50 subsystems with 1 component each, the average convergence time was around 76 iterations, at $\bar{\mathbf{t}}$ set to 1, as shown in Figure 39. However, when introducing learning, we see that this value decreased to vary between 21 and 8, according to the reinforcement learning parameter, as shown in Figure 45(b). This decrease in the average convergence time applies for all sized systems and their corresponding different decompositions. Moreover, it should be noted here that the impact of learning is more effective on the convergence time in large systems than in small ones, as the decrease in the convergence time, as shown in Figure 45(b), is more recognizable than in small and medium sized systems, represented in Figures 43(b) and 44(b) respectively. This is expected, as in large systems, and due to the big number of components and dependencies, complexity is more than in small systems, which makes the decrease in the convergence time, caused by learning, more effective. Comparing the average convergence time of different systems' decompositions, we recognize that as we decompose the system into more subsystems, the average convergence time is decreasing more with the increase of the reinforcement learning parameter. For example, in Figure 45(b), when the system was one big subsystem of fifty components, the average convergence time decreased from 9.86 to 8.38 iterations with the increase of Φ from 0.25 to 1. However, when the system is decomposed into fifty subsystems, each with one component, the average convergence time decreases significantly from 20.57 to 11.41 iterations.

As for the average performance values, they are roughly varying between 0.44 and 0.46, for all systems with different number of components $\boldsymbol{N}$ and different decompositions. Observing the

effect of system decomposition on the variation of the average performance, in Figures 43(a), 44(a) and 45(a), we can see that average fitness is not directly impacted by system decomposition as the values of highest and lowest performance in small, medium and large systems are sometimes achieved when the system is decomposed into a small number of subsystems. However, they are also noticed when we decompose the system into a larger number of subsystems. For this, we conclude that decomposition has no direct effect on the trend of average performance values.

CHAPTER 5

CASE STUDY

In this chapter, we apply our base and extended models, in sections 5.1 and 5.2 respectively, on a real-life system. This system is a gas turbine aero engine which is mainly composed of two building blocks: the high-speed spool and the low speed spool. In these two units, we have the following subunits: fan, low pressure compressor (LPC), high pressure compressor (HPC), the diffuser/combustor, high pressure turbine (HPT) and the low-pressure turbine (LPT). A simple representation of the system is shown in Figure 46 (Mascoli, 1999).

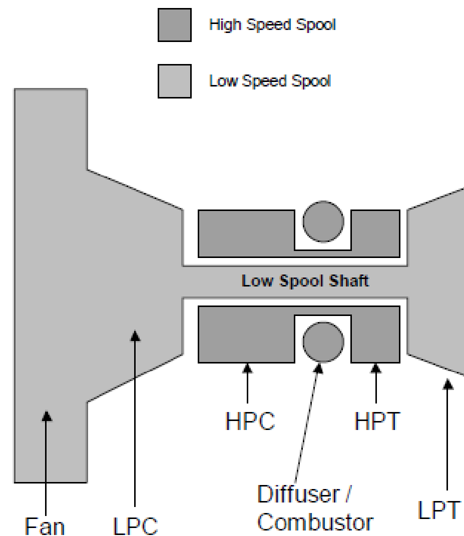


Figure 46: Representation of the gas turbine aero engine (Mascoli, 1999)

This engine performs three functions which allows it to provide propulsive thrust to the aircraft: compression of air, combustion of the compressed gas and exhaust of the combusted gas (Mascoli, 1999). A briefly description of this engine is as follows. The air (gas) is compressed in

the Compression Systems Module, which is composed of the fan, the LPC and the HPC. Then, the compressed air leaves the HPC to enter into the diffuser/combustor in which it is mixed with fuel and burned. Finally, the process of the combusted gas exhaust is accomplished in the Turbine Systems Module, which is composed of the HPT and the LPT.

To simulate the above gas turbine aero engine, we model it as a square (non-symmetric) matrix with N=81, which is divided into seven subsystems and each subsystem has its own number of components. Number of components per each subsystem are represented in Table 13 below.

Table 13: Distribution of the number of components in the subsystems

Subsystem S	Number of components in subsystem S
1-Fan	12
2-LPC	12
3-HPC	16
4-Diffuser/Combustor	7
5-HPT	13
6-LPT	14
7-Bearing Design and Rotor Support System	7

The above described system is as shown below, in Figure 47, however we didn't take into consideration the below highlighted part to ensure system decomposition.

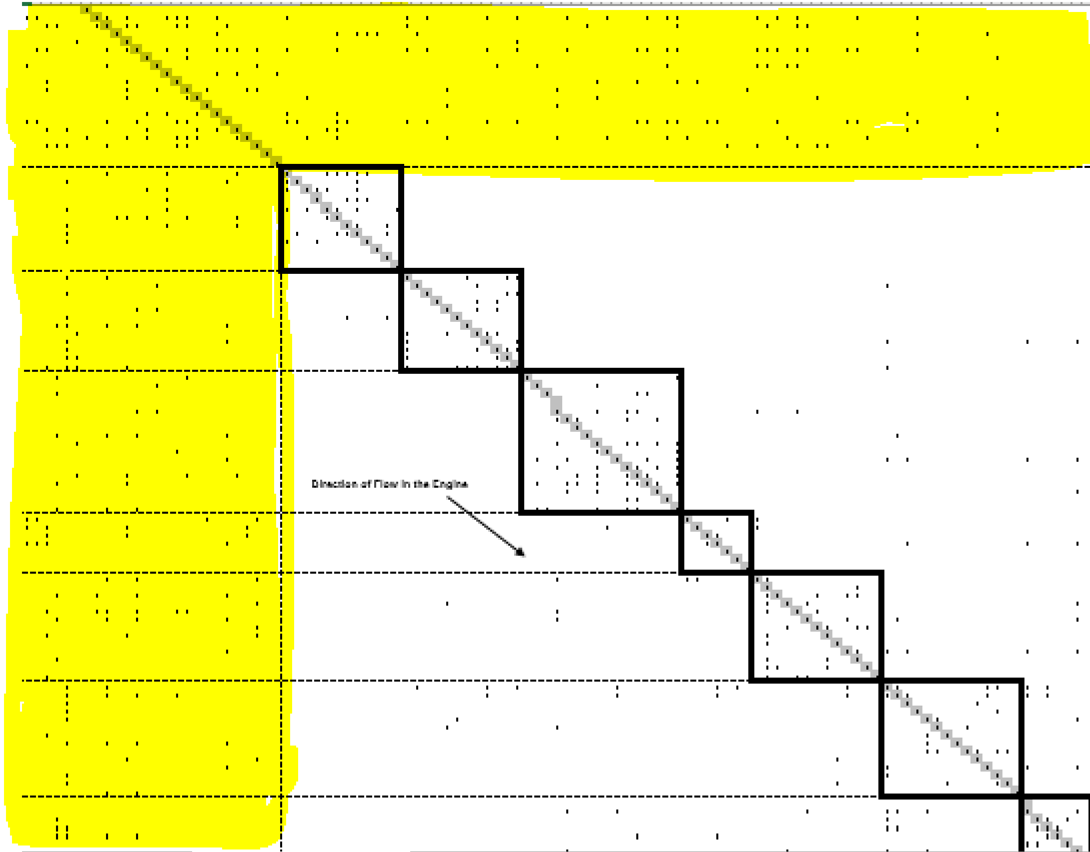


Figure 47: DSM of the Complete Gas Turbine Aero Engine

The unhighlighted system, taken into consideration in our case study, can be represented in the DSM shown in Figure 48 below.

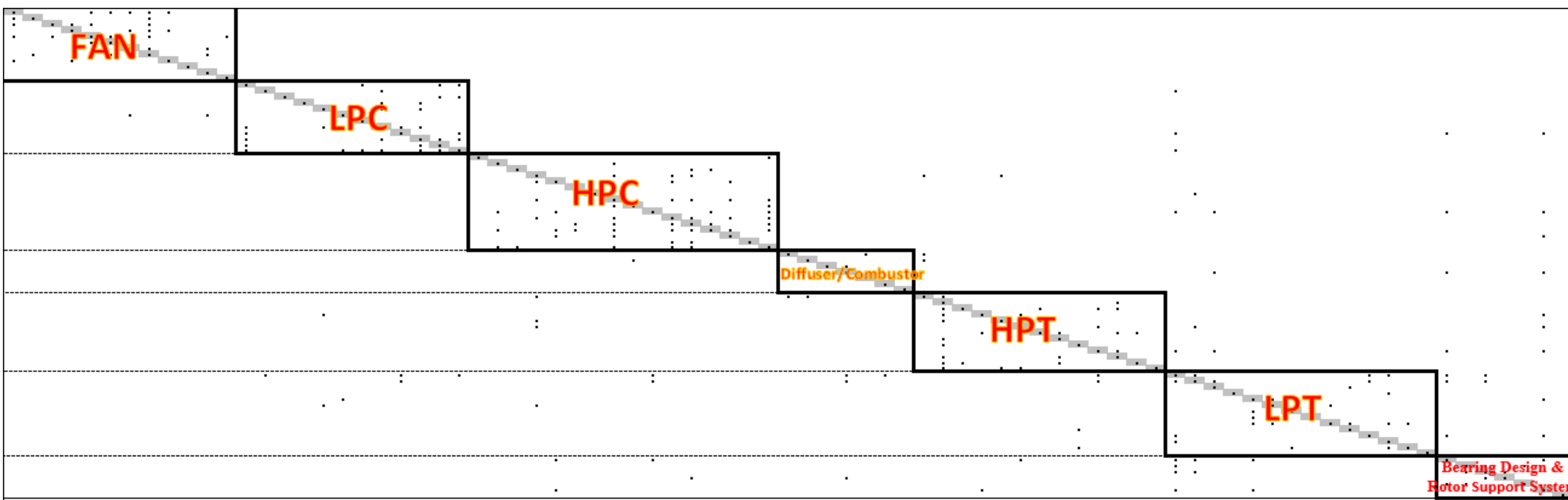


Figure 48: DSM of the Considered Gas Turbine Aero Engine

5.1 Case Study Simulation-Without Design Rules

In this section, we apply our base model on the system, described above, to check the convergence time of this system and the final performance value. We fix the value of the completion rate $\mathbf{r_i}$ to 1, that is each component has all the needed resource allocation intensity to be successfully resolved. As for the value of betta $\boldsymbol{\beta}$, it is variable since our model hosts the learning feature, however we set the initial value of the learning parameter $\boldsymbol{\beta}$, in this case study, to be 0.15. As for the reinforcement learning parameter $\mathbf{\Phi}$, we set it to be 1, that is the maximum value at which the system benefits from the learning process. Finally, we set the value of $\bar{\mathbf{t}}$ to 1, i.e. we always inform the external dependencies about the internal changes of the system. Noting that we set the value of $\bar{\mathbf{t}}$ to 1 to accommodate for the worst-case scenario, which is represented by the fact that the number of iterations will increase consequently when we deal with more external dependencies, the thing that increases the cost. We simulated the case study, over an average of 100 runs, and ended up with reaching system resolution after an average of **32.36** iterations. As for the average system performance value, it ended to be **0.4538.** In Figures 49 and 50, we show the variation of the performance value and number of unresolved tasks, respectively, over an average of 100 runs.

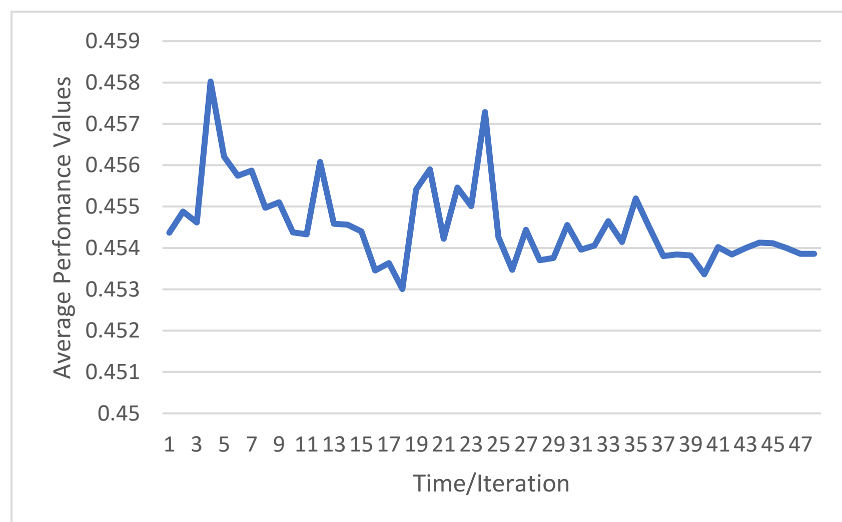


Figure 49: Variation of the average performance values of the case study system

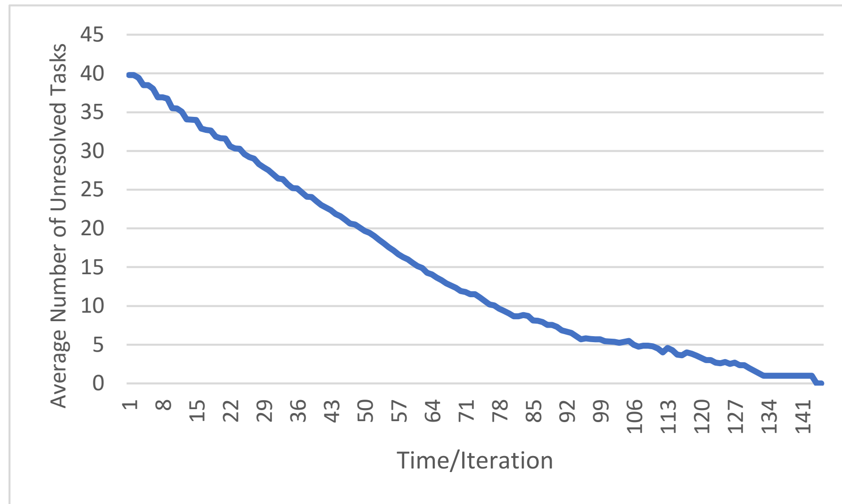


Figure 50: Variation of the average number of unresolved tasks of the case study system

We can observe from Figure 49 that the average performance of the gas turbine aero engine varies between 0.453 and 0.458 to finally get fixed at an average of 0.4538 after approximately 33 iterations. As for the convergence time, Figure 50 shows that the number of unresolved tasks in our case study decreases exponentially to reach complete system resolution after 33 iterations.

5.2 Case Study Simulation-With Design Rules

In this section, we apply our extended model, i.e. with design rules, on the system. In Table 14, below, we list all design rules, with all sized cycles, in our case study matrix, represented in Figure 48.

Table 14: All Cyclic Design Rules between Components

Design Rule #	Cycle Size	Components of the cyclic design rules
1	2	2 in subsystem 2 and 1 in subsystem 6
2	2	9 in subsystem 2 and 1 in subsystem 6
3	2	12 in subsystem 2 and 1 in subsystem 6
4	2	4 in subsystem 3 and 1 in subsystem 5
5	2	4 in subsystem 3 and 5 in subsystem 5
6	2	10 in subsystem 3 and 1 in subsystem 6
7	2	10 in subsystem 3 and 1 in subsystem 7
8	2	1 in subsystem 4 and 1 in subsystem 5
9	2	2 in subsystem 4 and 1 in subsystem 5
10	2	4 in subsystem 5 and 6 in subsystem 7
11	2	10 in subsystem 5 and 1 in subsystem 6
12	2	1 in subsystem 6 and 1 in subsystem 7
13	2	1 in subsystem 6 and 3 in subsystem 7
14	2	2 in subsystem 6 and 1 in subsystem 7
15	2	2 in subsystem 6 and 3 in subsystem 7
16	2	5 in subsystem 6 and 6 in subsystem 7
17	3	1 in subsystem 6, 1 in subsystem 7 and 10 in subsystem 3
18	5	10 in subsystem 3, 1 in subsystem 7, 2 in subsystem 6, 3 in subsystem 7 and 1 in subsystem 6

After listing the cyclic dependencies occurring in the case study, we need to decide which of these dependencies to select to be developed as design rules. This decision usually depends on

how subsystems interact with each other and on the importance and certainty of the interactions between components. Checking further the relation between the subsystems of the gas turbine aero engine, we notice that the dependency is highly dense and important between the HPC and HPT, that is subsystems 3 and 5, and between LPC and LPT as well, that is subsystems 2 and 6. This means that these subsystems are highly related to each other and that the role and function of components in one subsystem depend on the role and function of the components in the other subsystem. This is due to the fact that HPC and HPT are mechanically coupled through a shaft and HPT extracts enough energy to drive the HPC. As for the LPT, it is coupled with the LPC through the low speed shaft as well where the LPT extracts energy to drive the LPC (Mascoli, 1999). For this reason, we decided to select the cyclic dependencies which are between subsystems 2 and 6 and subsystems 3 and 5 respectively, represented by the first five listed design rules in Table A14.

As for the values of the system's parameters, we kept them constant as specified in section 5.1. When simulating the case study, considering the five selected design rules, we reach system convergence after **30** iterations with an average system performance of **0.412**.

Another approach we considered is to select all two sized cyclic dependencies as developed design rules, which resulted in reaching convergence time in 17 iterations with average fitness of 0.4.

Comparing results obtained in sections 5.1 and 5.2, we can see that design rules play an important role in decreasing the cost of system resolution. As managerial and design decisions are taken in the whole system, the design rules are developed and thus their corresponding components become standardized, that is their states become resolved and their fitness values are

constant. This process, as previously mentioned, has proven to save the system from running additional iterations as specifying a design rule will fix the role of the components, involved in the design rule, and will restrict any changes to be done on them, neither from components in external subsystems nor those in internal subsystems. To have a clearer view about the difference in the trend of system performance and number of unresolved components between the system with and without design rules, we present the results below in Figures 51 and 52 respectively.

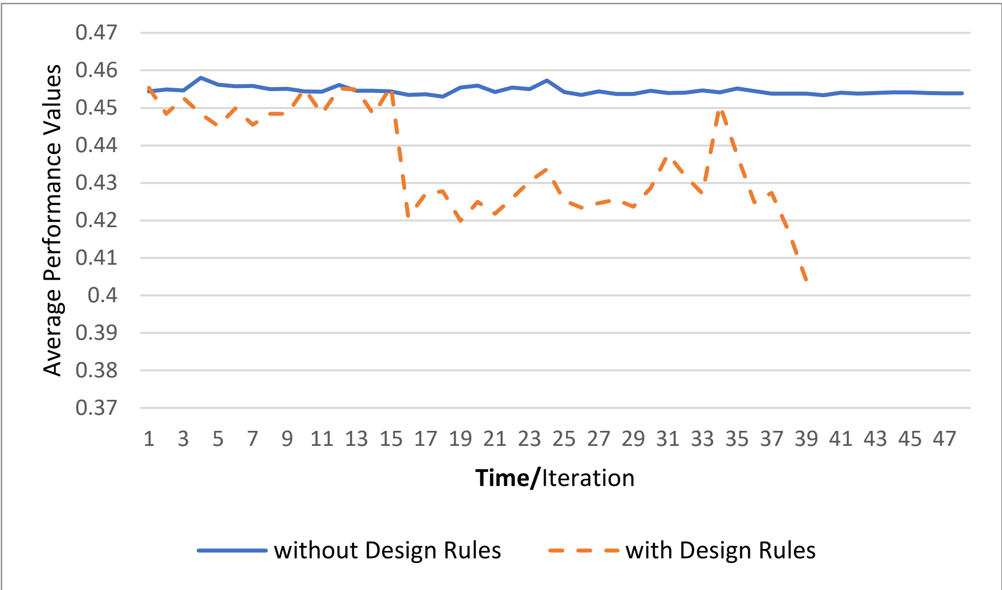


Figure 51: Variation of the average performance values with and without design rules

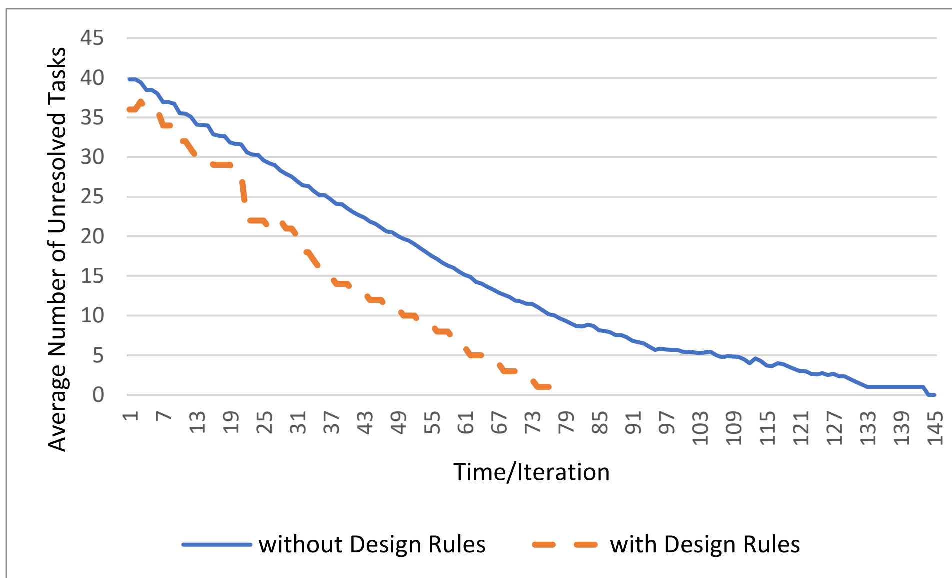


Figure 52: Variation of the average number of unresolved tasks with and without design rules

CHAPTER 6

CONCLUSION

The last decade witnessed a proliferation of product development (PD) research using complex systems and network theories. These studies aim to examine the nature of interactions between the systems' components and how they are affected by the system's properties, parameters and architecture. In this thesis, we focused on the effect of system's decomposition (i.e., the number of blocks to which the system is divided and the corresponding number of components in each of these subsystems). The overarching objective is to test how decomposition impacted the system's behavior which is mainly reflected by development process performance (i.e. the number of iterations to reach convergence), and on system or product performance (i.e. the overall system or product fitness value). Several sensitivity analyses and simulations were performed to check how different system decompositions are affected when varying the system parameters (the learning parameter $\boldsymbol{\beta}$, percentage of updating the external dependencies $\bar{\mathbf{t}}$, reinforcement learning parameter). We studied the importance of design rules as well as we presented the results of a system with and without design rules and compared the results in both cases. Furthermore, the presented case study demonstrated how the design rules can be effective in achieving system resolution faster, provided that these developed design rules are agreed upon. That is designers/managers must specify which interactions between subsystems can or should be standardized and developed as design rules.

PD managers can benefit from this model, as it is an efficient methodology to apply in systems, especially in large ones, such as companies, departments, etc. This model helps PD managers to

choose, experiment, and test the proper decomposition for their systems, which will consequently help in reaching system convergence faster. Moreover, product development managers can refer to the simulation results to generate (relative) metrics on the needed convergence time and expected system performance when reaching system resolution. Also, product development managers can benefit from the reinforcement learning feature in our model, which captures the increase in the amount of knowledge of the subsystems' components about each other and consequently result in lower cost to achieve system resolution. In addition, the design rules feature in our model introduces an additional leverage for organizations where managers and decision makers can enhance their system's convergence by standardizing the interdependencies between subsystems and thus decreasing the needed convergence time.

Our proposed model suffers from several limitations. One limitation to the model is the fact that the system can freely simulate without having a certain threshold on the number of iterations, unlike the case in real companies. In addition, in our model, design rules are based on random selection, rather than on certain finite selection. Selecting specific interdependencies as design rules, based on managers' knowledge and experience, may help in reaching system resolution more realistically than when being randomly selected. Another limitation of our model is that it disregards the effect of the type of dependencies between components, where each dependency between components might have either a positive or a negative effect on system performance, but this was not taken into consideration in our model.

For future work and research, the model can be extended to include additional parameters through introducing new restrictions to the system, such as limited budget and scheduled deadlines. For example, systems in real life cannot just evolve freely to reach resolution and convergence as they face specific limitations. Moreover, choice of the component to work on is

also not random but a deliberate choice is made by the development team. In addition, specifying the type of dependencies between the components; that is, when the performance of one component increases, the performance of its dependent components may increase or decrease depending on the nature of this dependency. Finally, we can study how we can achieve higher system performance when reaching system resolution, as the main focus in our model was more to resolve the system with an acceptable fitness value rather than reaching a high fitness.

REFERENCES

Baldwin, C. Y., Clark, K. B., & Clark, K. B. (2000). Design rules: The power of modularity (Vol. 1). MIT press.

Beesemyer, J. Clark, et al., 2011. Developing methods to design for evolvability: research approach and preliminary design principles, 9th Conference on Systems Engineering Research.

Brabazon, T., & Matthews, R. (2002). Product architecture, modularity and product design: A complexity perspective. tech. rep., University College Dublin.

Braha, D., & Bar-Yam, Y. (2007). The statistical mechanics of complex product development: Empirical and analytical results. Management Science, 53(7), 1127-1145.

Frenken, Koen, and Stefan Mendritzki., 2012. Optimal modularity: a demonstration of the evolutionary advantage of modular architectures. Journal of Evolutionary Economics 22, no. 5: 935-956.

Frenken, Koen., 2006. A fitness landscape approach to technological complexity, modularity, and vertical disintegration. Structural Change and Economic Dynamics17, no. 3: 288-305.

Hordijk, Wim, and Stuart A. Kauffman., 2005. Correlation analysis of coupled fitness landscapes. Complexity 10, no. 6: 41-49.

Kauffman, S.A., 1993. The Origins of Order: Self-Organization and Selection in Evolution. Oxford University Press.

Luo, Jianxi., 2015. A simulation-based method to evaluate the impact of product architecture on product evolvability. Research in Engineering Design 26, no. 4: 355-371.

Mascoli, G. J. (1999). A systems engineering approach to aero engine development in a highly distributed engineering and manufacturing environment (Doctoral dissertation, Massachusetts Institute of Technology).

McCarthy, I. P. (2008). Manufacturing fitness and NK models. Tackling Industrial Complexity, Institute for Manufacturing, Cambridge, UK [online] http://www. ifm. eng. cam. ac. uk/mcn/proceedings. htm (accessed January 2008).

Morin, P. (2013). Open Data Structures: An Introduction (Vol. 2). Athabasca University Press.

Oyama, Kyle, Gerard Learmonth, and Raul Chao., 2015. Applying complexity science to new product development: modeling considerations, extensions, and implications. Journal of Engineering and Technology Management 35: 1-24.

Rivkin, Jan W., and Nicolaj Siggelkow., 2007. Patterned interactions in complex systems: Implications for exploration. Management Science 53, no. 7: 1068-1085.

Simon, H. A. (2019). The sciences of the artificial. MIT press.

Solow, D., Burnetas, A., Tsai, M. C., & Greenspan, N. S., 2000. On the expected performance of systems with complex interactions among components. Complex Systems, 12(4), 423-456.

Songhori, M. J., & Nasiry, J. Organizational Structure, Subsystem Centrality, and Misalignments in Complex NPD Projects.

Woodard, C. J., & Clemons, E. K. (2014). Managing Complexity Through Selective Decoupling.

Yassine, Ali, and Dan Braha., 2003. Complex concurrent engineering and the design structure matrix method. Concurrent Engineering 11, no. 3: 165-176.

Yassine, A. (2004). An introduction to modeling and analyzing complex product development processes using the design structure matrix (DSM) method. Urbana, 51(9), 1-17.

Yuan, Y., & McKelvey, B. (2004). Situated learning theory: Adding rate and complexity effects via Kauffman's NK model. Nonlinear Dynamics, Psychology, and Life Sciences, 8(1), 65-101.

APPENDIX 1

Figure A1 (a): Flowchart of the Base Model

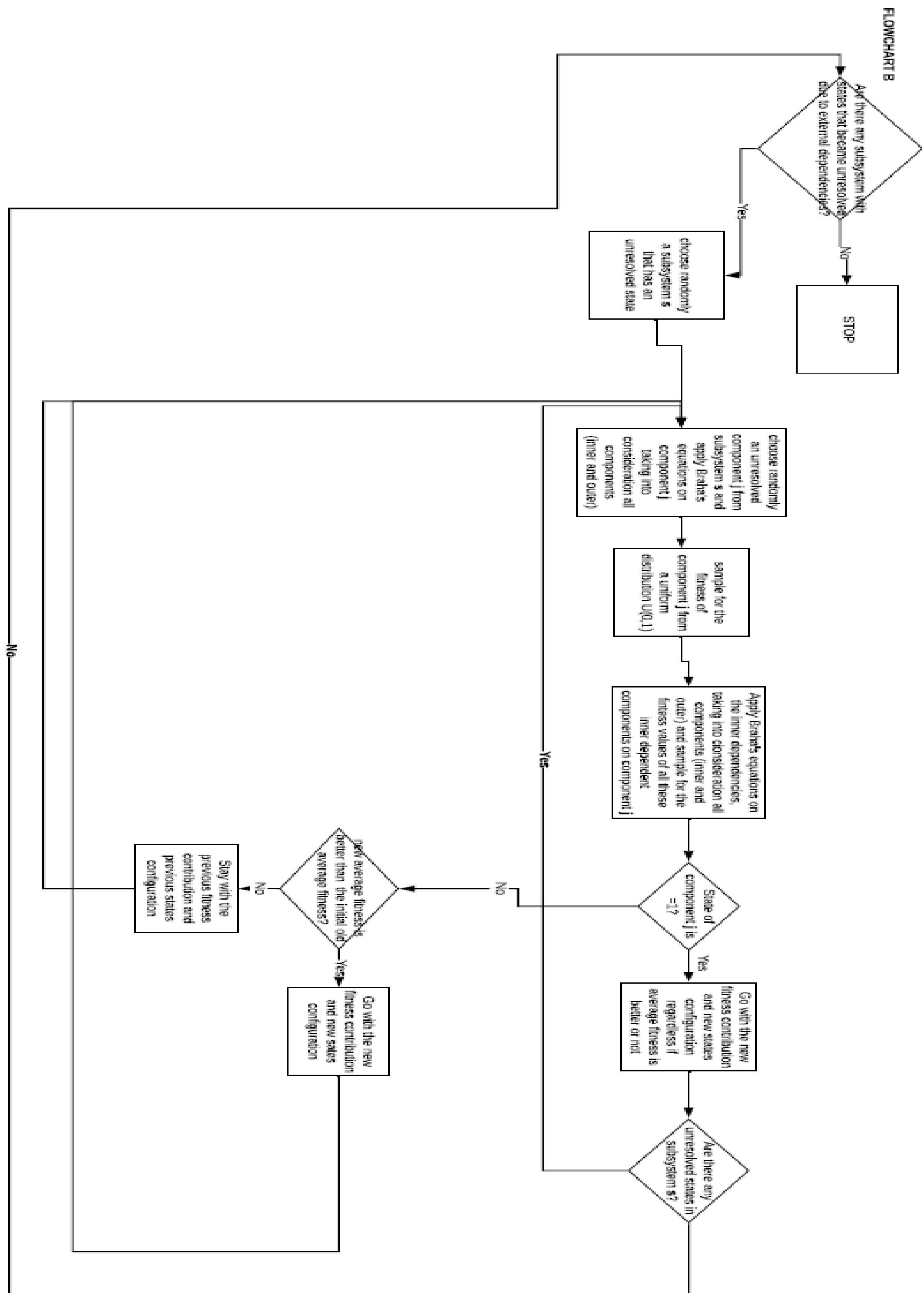


Figure A1 (b) : Flowchart of the Base Model-Continued

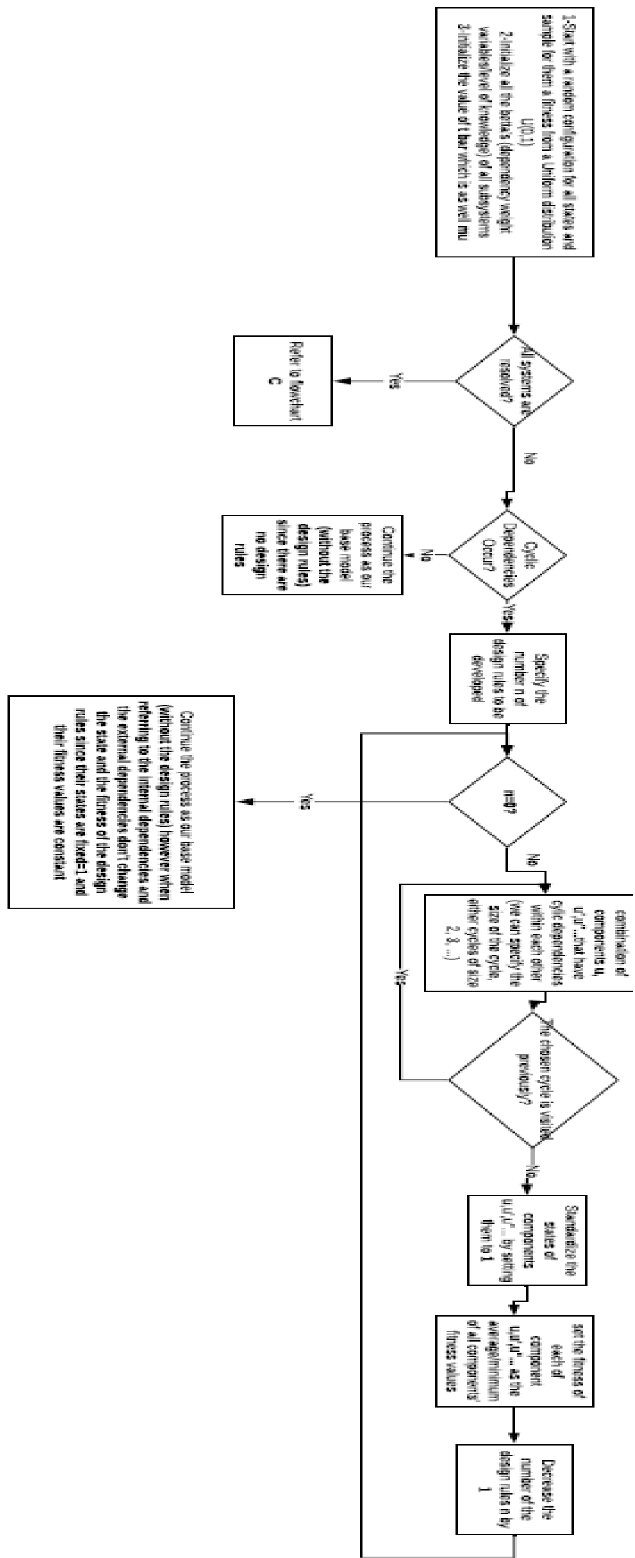


Figure A2 (a): Flowchart of the Extended Model

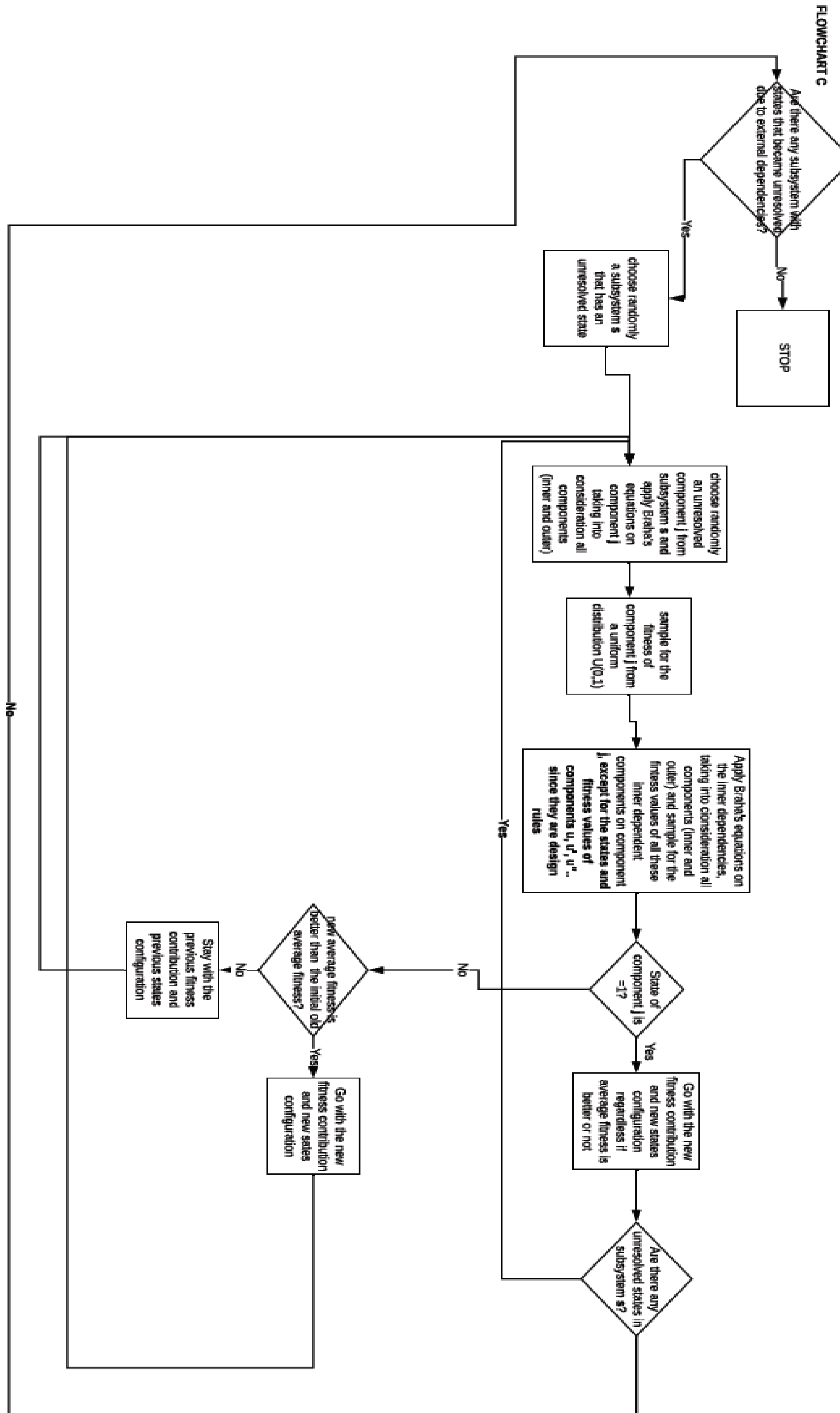


Figure A2 (b): Flowchart of the Extended Model-Continued