EVOLUTION STRATEGIES BASED AUTOMATED SOFTWARE CLUSTERING APPROACH

By

BILAL KHAN

(2006-NUST-MS PhD-CSE (E)-24)

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

Master of Science
in
Computer Software Engineering

Thesis Advisor
Dr. Shaleeza Sohail

MS-CSE-6

College of Electrical & Mechanical Engineering
National University of Sciences and Technology
2009

Acknowledgements

I thank Almighty Allah for the successful completion of my thesis. I gratefully acknowledge the support and guidance of my advisor Dr. Shaleeza Sohail. She has been a great mentor always providing me with the much needed encouragement and thoughtful direction. I am also thankful to my advisory committee for their feedback and valuable suggestions.

During my research I also worked full-time as a software engineer. Special thanks to my manager, Muhammad Nadeem, who provided me with flexibility and guidance.

The nexus of my family and friends was also instrumental in enabling me to finish this project, bearing my late working hours taken from their time.

I offer best regards to all those who were helpful in any way during this research.

Dedication

Dedicated to my family and friends who always supported me and prayed for my success.

Abstract

Maintenance is one of the key phases of software development life cycle, for long term effective use of any software. It can become very lengthy and costly for large software systems, especially when subsystem boundaries are not clearly defined. System evolution, lack of up to date documentation and high turnover rate of software professionals (leading to non availability of original designers of the software systems) can complicate the system structure many folds by making the subsystem boundaries ambiguous. Automated software module clustering helps software professionals to recover high-level structure of the system by decomposing the system into smaller manageable subsystems, containing interdependent modules. We treat software clustering as an optimization problem and propose a technique to get near optimal decompositions of relatively independent subsystems, containing interdependent modules. We propose the use of self adaptive Evolution Strategies to search a large solution space consisting of modules and their relationships. We compare our proposed approach with a widely used genetic algorithm based approach on a number of test systems. Our proposed approach shows considerable improvement in terms of quality and effectiveness and consistency of the solutions for all tests cases.

Table of Contents

List of Tables

List of Figures

Chapter One: Introduction

In today's advanced world, software has gained a large share in the expenses and gains of market shares not only for purely software dominated domains such as communications and management information systems, but also in other conventional technology realms like aviations, electronics engineering, astronomy and media industry. The share of software in production for these conventional technology domains measured up to 30 to 50 percent. The average fortune-100 company has 35 millions lines of code in operation with a growth of 10 percent per year [1].

Case studies dealing with software costs reveal that software evolution takes 60-80% of the total cost of a software product [2]. Interestingly though, the academia and the industry have made insufficient efforts to deal with the problems of software evolution as compared to efforts put in to the software development area. It was the "Y2K problem" that brought software evolution into the limelight. However, even this example of a mass change has not changed the situation very much [3]. More than 50% of the time needed for program evolution is spent in understanding the program before the actual change can be designed and realized, as several case studies have shown [4]. The main reason behind this is the unavailability of the complete and meaningful documentation required for the task. In the absences of correct information, the developers responsible for the maintenance, who are already under

strict time constraints, have to fix the problem locally rather than finding the origin of the problem and fixing it at the root. These local code fixes that treat the problem only phenotypically instead of being the real solution, not only disrespect the intended design but also become source of error at other sites of the software systems. This also complicates the understanding of the software system in the future. Such circle continues, and unless appropriate deterrent policies are adopted, the software system becomes virtually impossible to maintain.

1.1 The Importance of Software Architecture

Large software systems are compromised of many subsystems. The architecture of a software system is composed of these subsystems, sometimes referred to as components, and the dependencies existing among these components. Most of the attributes of a software system depend upon the architecture of the system. Hence software architecture is key asset. If the architecture is inappropriate or it gets deteriorated, one way or another, it can have catastrophic effects on the maintainability of the software system. The software architecture can have major impact on the following aspects of a system as describe by Garlan and Perry [5]. They have described the impact with a development point of view. We present these aspects from the view point of a person responsible for maintaining a software system.

***Understandability*:** The software architecture presents a system at a higher level of abstraction. This representation depicts the high-level constraints on system design that a maintainer has to observe. This also helps more focused searching on

architectural information. Many design decisions and the penalties on their violation become only clear at this level.

Reuse: It is through the architecture that a maintainer can identify the reusable components and their dependencies to other parts of the system that need to be handled before the components can be reused. Current work on reuse is generally limited to component libraries. Architectural design supports not only the reuse of large components but also frameworks into which components can be integrated.
Architecture recovery is also vital for product line scenario where common architectural components of a class of systems are integrated and generalized into a generic architectural framework for a particular domain; the architectures of the actual systems in this domain can then be realized as instantiations of the general framework [6].

Evolution: The software architecture is the skeleton of the system. The software architecture description empowers the maintainer to identify the bottle-neck and potentially weak parts that need to be carefully tackled as the system goes through evolution phase. Also, the availability of information of a component's dependencies enables the maintainer to modify the component in a manner that does not affect other parts of the system. It also helps to modify the dependencies in order to deal with concerns regarding reuse, performance and interoperability. The architectural information enables the maintainer to fix the errors where they were cause rather than where they appear, through the identification of responsible components or the undocumented dependencies and constraints.

Analysis: Documenting the recovered architecture provides new opportunities for analysis. This includes analysis of high-level system consistency, abidance to the pre-agreed architectural style and conformance to quality attributes. In addition, the architectural description can be used to keep the check that changes to the system do not violate the design principles of the architecture.

Management: Maintenance tasks can be assigned and managed made on the basis of subsystems. Furthermore, the software architecture allows relatively precise risk and cost estimation of a modification. The quality of a system can be evaluated by estimating the stress-bearing capacity of its architecture. Weak parts can be isolated and procedures to overcome these weaknesses can be chalked out and examined.
Components with many problems may have to be reengineered. Reengineering of large systems is a viable option if it is done at subsystem basis. For such an approach,
The information of component dependencies and the plan for packaging of not yet reengineered components must be available. As all the above mentioned factors play an important role during the evolution of the system, software architecture recovery becomes an important task. Once the architecture is recovered, the documentation should be kept up-to-date with future changes and the need for recovery should never arise again in the future. However, it may still be important to examine the architecture to identify and analyze differences from the documented architecture. Moreover, the maintainer may need to explore the architecture if its description has been abstracted from certain details. Recovering and exploring the software

architecture is costly and the only available automated support in practice often a simple debugger to trace the system at a low level.

1.2 Research in Architecture Recovery

Architecture recovery includes the detection of **components** (the computational parts) and **connectors** (the means and points of communication) of systems. It is aimed at supporting the process of program comprehension for software maintenance and evolution.

Component Recovery: A major research area in component recovery is detection of subsystems [7]; another one is recovery of objects and abstract data types. Though abstract data type and object detection is commonly driven by reuse or object-orientation, it does support architecture recovery at a lower level of components.

Subprograms, types, and global variables are the bases elements. With these base elements we can form the architectural concepts of abstract data types and objects. Other examples can be hybrid components or collection of related subprograms. Such low-level components solely built from types, variables, and subprograms as referred to as **atomic components**.

Connector Recovery: Connectors for concurrent and distributed systems have been the primary target of connector recovery [8], [9]. Nevertheless, most software systems, especially the legacy software systems, are sequential and monolithic. A primitive connector for such software systems is the Function Call. Shared global variables are another means of communication among different components. At the

next higher level of connectors, we come across atomic components. For instance, a pipe may be the means of communicate between two architectural components, where the pipe is as an abstract data type. This implies that atomic components themselves can be connectors at a higher level of architecture.

Hence, their detection can assist in understanding the communication among larger components

The focus of this thesis is to developing an automated technique to facilitate the task of recovering the structure of software systems. Before describing our work in detail we first present an introduction to the software clustering problem.

1.3 The Software Clustering Problem

The large size and complexity of industrial software systems make the understanding of their structure a difficult task. A typical software system usually consists of thousands of entities (procedures, classes, modules) that are integrated in various ways (procedure calls, inheritance relations). In practice, usually the documentation is either obsolete or non-existent. This means that the software system's architecture is limited to the system architects and developers involved. This knowledge is often lost when these knowledgeable people switch to another project or another company. For many of the legacy software systems currently deployed around the world, this knowledge has been lost years ago.

The fact that the need to modify existing industrial software systems is quire frequent is backed by the number of software reengineering projects. A variety of reasons exist

that require such modifications. These include, but are not limited to, migration to new hardware platforms or operating systems, compliance with new industry standards, or change in user requirements.

The software reengineering research is aimed at dealing with the difficult problems of understanding, re-documenting, and modifying software systems. A main factor that makes these problems tough to address is the large size of these software systems. Researchers in other realms of science and technology have also been faced with large amounts of data. The approach they usually adopt to address this complexity is to develop a taxonomy, i.e. categorize objects that exhibit similar features or properties. A variety of techniques have been proposed in the literature to discover such categories (commonly referred to as *clusters).* The field of *cluster analysis* [10, 11, 12, 13, 14] emerged from the study of these techniques.

In software context, several software researchers have developed similar techniques either by adapting existing cluster analysis techniques, or by proposing new ones. The objective of all the approaches presented in the literature is to decompose large software systems into smaller and manageable subsystems that are easier to understand. Such techniques are collectively referred to as *software clustering* techniques.

Numerous software clustering approaches exist in the software engineering literature. All these approaches deal with the software clustering problem from a different perspective, either by trying to compute a similarity measure between software entities[15]; deducing clusters from file and procedure names [16]; utilizing the

connectivity between software objects[17, 18, 19] or looking at the problem at hand as an optimization problem [20].

1.4 Our Approach for Software Clustering

Based on the goals they try to achieve, the software clustering approaches can be categorized in three different classes:

1. **Re-modularization Approaches**. These approaches strive to re-modularize (group the system's resources in a different way) in order to improve certain attributes of the software system such as its maintainability or evolvability.

2. **Objectification Approaches.** Many of the software reengineering projects are aimed at the migration of the source code from a procedural language to an object-oriented one. The objectification approaches attempt to identify the entities that would be candidates for classes.

3. **Comprehension approaches**. The goal of these approaches is to decompose the software system into smaller, manageable subsystems in a way that will assist in understanding the architecture of the software system.

The thesis focuses on a software clustering approaches directed towards re-modularization. For convenience, the term software clustering approach will be used to refer to an approach directed towards re-modularization.

Software clustering is aimed at categorizing large systems into smaller manageable subsystems containing modules of similar features. Thus clustering facilitates better comprehension of the system. The decomposition is based on the relationships among the modules. These relationships are usually represented in the form of module dependency graph where modules are represented as nodes and the relationships as the edges between these nodes. The software clustering problem can be seen as partitioning of this graph into clusters containing interdependent modules. However, the number of possible partitions can be very large even for a small number of nodes [20]. Moreover, the fact that even small differences between two partitions can generate quite different results, enhances the problem domain. Hence finding the best clustering for a given set of modules has been proved to be a NP-hard problem [21].

In this thesis, we propose Evolution Strategy Based Automated Software Clustering Approach (ESBASCA) that treats the clustering problem as an optimization problem with the goal of finding near optimal partitions. We define the criteria for near optimal partitioning in Chapter 5. Our approach searches the large solution space that consists of all the possible partitions and after a number of iterations finds the near optimal partitioning for the given system. The inherent quality of Evolution Strategies (ES)s [22], [23] is the self adaptability which makes sure that as the number of iterations are increased ESBASCA always gets same or better result than before and never lose local optimal value during the execution. To show the effectiveness of our approach we have compared it with Genetic Algorithm (GA) [24], [25] based clustering approach and results show considerable improvement by ESBASCA. The improvement is due to two main factors that GA suffers from when compared to ES:

- Reproduction can eliminate good solutions in GAs while good solutions always survive into the next generation in ES.
- In GA the strategy parameters (e.g., mutation strength) remain constant so it may remain stuck at local optima. Self adaptive ES on the other hand promises better results because self adaptation helps faster convergence and fine tuning of the search along the fitness landscape.

Figure 1.1 presents the MDG of a small software system and Figure 1.2 shows a sample decomposition generated by our clustering approach.

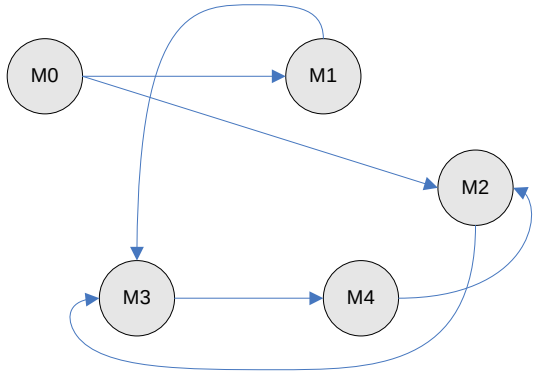

Figure 1.1: An Example MDG

For any optimization algorithms we cannot achieve exactly similar result in different executions of the algorithm on same data. However, the results should be close to each other without any major variations. One of the most desirable properties of any working algorithm is the consistency in its results. Hence, an important measure while evaluating the performance of any clustering algorithm is the consistency of the results produced by it. Keeping this in mind we conducted a comparative study on the consistency in results produced by ESBASCA and GA based approaches. We found

that ESBASCA gives far more consistent results as compared to the GA based approach.

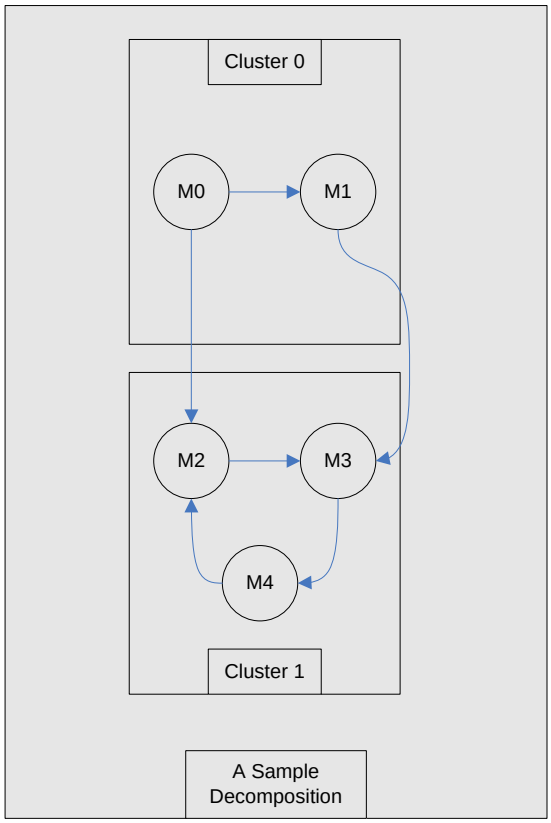

Figure 1.2: A Sample Decomposition of the Example MDG

1.5 Thesis Outline

In this chapter we introduced the software clustering problem, and presented an overview of our approach to addressing this problem. The remainder of this section provides an outline of rest of the chapters in this thesis.

Chapter 2- Literature Survey

This chapter surveys the software clustering techniques. It examines clustering approaches in the software domain. The chapter concludes with a discussion of

research challenges related to software clustering, providing motivation and perspective for our research.

Chapter 3- Overview of Software Clustering

This chapter presents and overview of the software clustering process. It discusses the sourced code entity representation, similarity measures and clustering algorithms. It also encompasses the topics of source code analysis and graph visualizations.

Chapter 4- Evolution Strategies

This chapter introduces the evolution strategies. It discusses the basic operators involved in evolution strategies. The important aspect of self adaptation and a generic evolution strategies algorithm is also presented in this chapter.

Chapter 5- ES based Software Clustering Approach

This chapter presents the instantiation of ES Algorithm for the software clustering problem. Besides the ideas of bringing software clustering problem into a representation on which evolution strategies can be applied, an objective function for the same is also defined and explained in this chapter.

Chapter 6- Implementation

This chapter is dedicated to the implementation of our software clustering tool. It presents the working environment and the architectural design of our tool.

Chapter 7- Experimental Results and Analysis

This chapter presents the experimental settings, results obtained and a detailed analysis of the results.

Chapter 8- Conclusion and Future Research Direction

This chapter concludes the thesis and points our future research directions. It also presents our research contributions.

Chapter Two: Literature Survey

There has been considerable research activity in the field of reverse engineering over the past few years. Several reasons account for this activity. First, immediately prior to the year 2000, information technology professionals spent a great deal of effort and time verifying that their software will work into the new century. Many software developers, who had no access to the original designers of the system under their consideration, had to remedy the software so that it would work after the year 2000. Second, the software development processes have gone through rapid changes. Software systems have migrated from two-tiered, to three-tiered, to n-tiered client/server architectural models in the past few years. Development approaches have varied from procedural, to object oriented, to component based. Developers have to strive to port the earlier version of their software to fit the new one each time the system architecture goes through a change. In these situations, software professionals are being forced to understand, and in some cases, re-modularize huge code bases. Tools for software clustering can help these professionals by providing automated support for recovering the abstract structure of such large and complex systems.

In the earliest days of computing, the need for clustering procedures into modules was identified. The popular work of David Parnas [26] first suggested that the “secrets” of a program should be hidden behind module interfaces. The information hiding principles based on this concept advocated that modules should be designed in a manner that design decisions of the module are hidden from all other modules. Parnas

suggested that module interfaces should be created in order to provide a well-defined mechanism for communicating with the modules' internal logic. Parnas proposed that procedures acting on common data structures should be grouped (clustered) into common modules. Parnas's ideas were a foundation for object-oriented design techniques.

Objected-oriented techniques provide a primitive clustering by grouping related data, and functions that operate on the data, into classes. Booch [27] suggests that during the design process a system should be decomposed into autonomous objects that collaborate with one another to provide higher-level system behavior. Booch emphasizes the importance of abstraction, encapsulation, modularization and organizing design abstractions into hierarchies. Abstraction focuses on the similarities between related entities. Encapsulation provides information hiding. Organizing design abstractions into hierarchies is aimed at simplifying program understanding. Modularization promotes strong cohesion and loose coupling between classes. Almost all research in software clustering concentrates on one or more of these concepts.

Given the importance of recovering and understanding the architecture of source code, the remainder of this chapter will present work performed by researchers in the area of software clustering. The research work on software clustering is classified into bottom-up, top-down, data mining, and concept analysis clustering techniques. In the end of this chapter, we present miscellaneous approaches that have been applied by researchers to the software clustering problem.

2.1 Bottom-Up Software Clustering Techniques

Following sections present some important bottom-up clustering approaches.

2.1.1 Data Binding

In an early paper on software clustering, Hutchens and Basili [17] have presented the concept of a data binding. A data binding categorizes the similarity of two procedures on the basis of common variables in the static scope of these procedures. Because of their ability to cluster the procedures and variables into modules, data bindings are useful for modularizing software systems (e.g., helpful for migrating a program from COBOL to C++). The authors of this paper have presented several remarkable aspects of software clustering. First, they identify the importance of maintaining a consistency between the systems' reverse engineered model and the designer's mental model of the system's structure. The authors also claim that software systems are best considered as a hierarchy of modules and they have focused on clustering methods that demonstrate their results in this fashion. Finally, the paper addresses the software maintenance problem by presenting the benefits of clustering technologies to verify how the structure of a software system deteriorates over time.

2.1.2 Semi Automated Clustering

1. ***ARCH.*** A semi-automatic approach to software clustering is provides in Robert Schwanke's tool *Arch* [7]. Arch is intended to help software professionals to understand, reorganize and document system structure, integrate system architects' opinion and to monitor compliance with the recovered architecture. Schwanke's clustering heuristics are based on the

software engineering principles of cohesion and coupling. They are based on maximizing the cohesion of procedures present in the same module while minimizing the coupling between procedures that reside in different modules. Arch also provides an innovative feature called *maverick analysis* which redefines modules by locating misplaced procedures and placing them in most appropriate modules. Schwanke also explored the use of neural networks and classifier systems to modularize software systems [28].

2. ***RIGI.*** Hausi Muller in his work adopted a more abstract software structure i.e. the subsystem as the basic building block of a cluster rather than a module. Muller's tool, *Rigi* [29], implements many heuristics that guide software engineers during the clustering process. The heuristics presented and discussed by Muller vastly focus on measuring the “strength" of the interfaces between subsystems. *Omnipresent module* is an exclusive aspect of Muller's work. While examining the structure of a software system, often modules acting like libraries or drivers are found. These modules either provide services to other subsystems, or consume the services of other subsystems. Muller classifies these modules as omnipresent and stressed that they should be ignored during cluster analysis because they add ambiguity to the system's structure. Another feature of his work is the concept that the module names themselves can be used as a clustering criterion. Later in this section we discuss a paper by Anquetil and Lethbridge [30], which investigates this technique at length.

The research of Schwanke and Muller resulted in semi-automatic clustering tools that require significant user input and feedback to obtain meaningful results.

2.1.3 Resource Based Clustering

Choi and Scacchi's [19] paper presents a fully-automatic clustering technique based on maximizing the cohesion of subsystems. Their clustering algorithm starts with a *resource flow diagram* (RFD) that actually is a directed graph and forms a hierarchy of subsystems using the articulation points of the RFD. If a module A provides one or more resources to module B, an arc is placed from A to B in the RFD. Articulation points are nodes in the RFD that divide the RFD graph into two or more connected components. Their algorithm searches for these articulation points. Each articulation point and connected component of the RFD is used as the starting point of forming subsystems. Choi and Scacchi specified resultant design of the system using the NuMIL [31] architectural description language.

2.1.4 Optimization Techniques

Mancoridis, Mitchell et al. [20, 32, 33, 34] treat the software clustering problem as a search problem. An important aspect of their clustering technique is that they do not try to cluster the native source code entities directly into subsystems. Instead, they start by generating a random subsystem decomposition of the software entities. Then they apply heuristic searching techniques to shift software entities either between the randomly generated clusters or in some cases they even create new clusters, to produce improved subsystem decomposition. This process is iterated until no further improvement is possible. The search is guided by an objective function based on software engineering concepts of cohesion and coupling. Their algorithm rewards

high cohesion and low coupling. They have used several heuristic search approaches based on hill-climbing and genetic algorithms.

Their tool, Bunch, clusters source code into subsystems automatically. The fully automatic capability of Bunch that distinguishes it from related tools that require significant user input to guide the clustering process. However, they have extended Bunch over the past few years to integrate other useful features that have been described and/or implemented by other researchers. For instance they added Orphan Adoption techniques [35] for incremental structure maintenance, Omnipresent Module support [29], and user directed clustering to complement Bunch's automatic clustering engine.

2.2 Top-Down Clustering Techniques

In the following subsections we present some key top down clustering techniques.

2.2.1 Software Reflexion Model

Most software clustering techniques work in a bottom-up manner. These techniques provide high-level architectural views of a software system from system's source code. Murphy's work with Software *Reflexion Models* [43] works in a top-down manner. The goal of the Software *Reflexion Model* is to recognize the differences between a designer's mental model of the system structure and the actual organization of the source code. Once these differences are identified and understood, either the designer can update his model or the source code can be modified to comply with the designer’s understanding. This technique is valuable to prevent the system to drift away from the intended structure as it undergoes maintenance.

2.2.2 Static and Dynamic Analysis

Eisenbarth, Koschke and Simon [44] used concept analysis to develop a technique to map system's externally visible behavior to relevant parts of the source code. Their technique uses static and dynamic analysis to enable the users to understand a system's implementation without much knowledge about its source code. Profiling is done to collect data while the program is execution. Concept analysis is then used to process this data in order to identify a minimal set of feature-specific modules that participated in the execution of the feature. Static analysis is then performed against the results of the concept analysis to identify further feature-specific modules. The goal is to reduce the set of modules that participated in the execution of the feature to a small set of the most relevant modules in order to simplify program understanding. Their case study was based on two open source web browsers whose various features were investigated which were then mapped to a small fraction of software system modules.

2.3 Concept Analysis Clustering Techniques

Concept Analysis Clustering can be classified into two main categories namely Modularization Concept Analysis and Objectification Concept Analysis. We discuss each of them in the following subsections.

2.3.1 Modularization Concept Analysis

Lindig and Snelting [26] used mathematical concept analysis [37] to develop a software modularization technique. Conceptually, their work resembles that of

Hutchens and Basili [17] i.e. the goal is to cluster procedures and variables into modules based on variable dependencies between procedures. They first generate a variable usage table. This table captures the shared variables that are used by each procedure in the system. The authors then use a technique for converting the table into a concept lattice which is a convenient way to visualize the variable relationships between the procedures in the system. Their technique then methodically modifies the procedure interfaces to remove global data dependencies. This is done by passing the variable through the procedure's interface. This is aimed at transforming the concept lattice into a tree-like structure. As soon as this transformation is achieved, the modules are realized from the concept lattice. Modularization by *interface resolution* and modularization by *block relation* are the two techniques used by the authors to achieve this transformation: Due to performance problems, the authors failed to modularize two large systems as part of their case study. Furthermore, their technique is only useful for analyzing systems that are developed using programming languages liked COBOL and FORTRAN that rely on global data as a means for information sharing. Thus, their technique is not for object-oriented programming languages, based on the concept of encapsulation.

2.3.2 Objectification Concept Analysis

Van Deursen and Kuipers [38] investigated the use of clustering and concept analysis techniques to identify objects from COBOL code automatically. Their approach to object comprises the following steps:

1. Categorize the COBOL records as objects,
2. Categorize the procedures or programs as methods

3. Use clustering technique to decide the best object for each method

Their algorithm starts by creating a usage matrix that cross references the relations between modules and variables. Once this matrix is created, a hierarchical agglomerative clustering algorithm [39] similar to *ARCH* [7] is used. Euclidian distance between the variables in the usage matrix is used to calculate dissimilarity measurement. Clusters are formed on the basis of this dissimilarity measurement.

The authors also explored the use of concept analysis to determine clusters from the variables in the usage table. In this technique, the usage table is transformed into a concept table by considering the items (variable names) and features (usage of variables in modules). After identifying the items and features, they locate the maximal collection of items sharing common features which determines the concepts.

Similar to the Lindig and Snelting's approach, the concepts can be represented as a lattice. Clusters can be determined at various granularity levels by moving from the bottom to the top of the lattice.

2.4 Data Mining Clustering Techniques

Visual Representation Model and Graph Annotation are the main clustering techniques based on data mining. Each one of them is described in the following subsections.

2.4.1 Visual Representation Model

Montes de Oca and Carver [40] present a formal visual representation model for deriving and presenting subsystem structures. Their work uses data mining techniques to form subsystems. They claim that data mining techniques are complementary to the software clustering problem. More importantly:

1. In database management, data mining has been used to find non-trivial relationships between elements. Software clustering in a similar manner forms subsystem relationships based on non-obvious relationships between the source code entities.

2. Data mining can discover interesting relationships in databases without much knowledge of the objects being studied. One of the salient features of software clustering is that it can be used to promote program understanding.

3. Data mining techniques operate on a large amount of information. Thus, the study of data mining techniques may advance the state of current software clustering tools that usually suffer from performance problems due to the large amount of data that needs to be processed.

The authors form subsystems on the basis of dependencies of procedures on shared files. They have not explained the working of their similarity mechanism because their research was primarily aimed at developing a formal visualization model. However, they presented a good set of requirements for visualizing software clustering results. For example, their visualization approach supports *hierarchical*

nesting, and *singular programs*, which are programs that do not belong to a subsystem. The concept of *singular programs* is very similar in concept to Muller's *omnipresent modules* [29].

2.4.2 Graph Annotation

Sartipi et al. [41, 42] also explored the software clustering problem through the data mining approach. Their technique used data mining to annotate nodes in a graph representing the structure of a software system with association strength values. These association strength values are then used to partition the graph into clusters.

2.5 Other Software Clustering Techniques

Besides the techniques discussed earlier in this chapter, the techniques presented in the following subsections are also significant.

2.5.1 Clustering Based on Naming Conventions

Anquetil, Fourrier and Lethbridge [45], investigated several hierarchal clustering algorithms. The primary objective of their research was to evaluate the effects of varying the clustering parameters while applying clustering algorithms to software re-modularization problems. They presented three quality measures to compare the results of their experiments. The most important measure is the *precision and recall* measure that computes the difference between two clustering results. This is typically used to match the decompositions produced by a clustering algorithm with the decompositions produced by some expert (e.g. the original designer of the system). *Precision* measures how much the clustering results agree with the expert

decomposition. While *Recall* measures the agreement between the expert and the clustering method. The authors found that many clustering algorithms yielded good precision and poor recall.

Anquetil et al. also state that clustering algorithms impose the system structure rather that discovering it. They further state that an important decision is to select a clustering technique that best suits a particular system. Their statement is generally believed to be correct. In case of software maintenance, where the objective is to understand a software system's structure, it is usually desirable to impose a structure conforming to the information hiding principle of software engineering. This is the reason that many similarity measures are based on maximizing cohesion and minimizing coupling.

Anquetil and Lethbridge [30] also proposed a clustering technique based on naming conventions. Such techniques cluster entities with similar source code file names and procedure names. They claimed that this technique often produces better results as compared to the techniques based on extracting information from the source code. They presented case studies based on name similarity as the clustering criterion that showed promising results (high precision and high recall).

This technique, however, is very subjective. If developers organize their source files into directories, and name source code files that perform a related function in a similar way, then this technique will show good results. However when there are inconsistencies in naming, e.g. when a system has undergone maintenance by developers who did not understand the system's structure thoroughly, this technique

might not work well. In contrast, clustering based on source code always provide accurate information as this information is directly extracted from the source code.

2.5.2 Comprehension Driven Clustering

Tzerpos' and Holt's ACDC clustering algorithm [46] uses patterns having good program comprehension properties to determine the system decomposition. They have presented seven subsystem patterns and their clustering algorithm that applies these patterns to the software structure. This places most but not all of the modules into the subsystems. ACDC then uses orphan adoption [35] to assign the remaining modules to appropriate subsystems.

2.5.3 Koschke's Clustering Research

Koschke's Ph.D. thesis [47] presents 23 different clustering techniques and classifies them into following categories

1. connection-based
2. metric-based
3. graph-based
4. concept-based

16 out of the 23 techniques in his work are fully automatic while 7 are semi-automatic. The author also developed a semi-automatic clustering framework based on modified versions of the fully-automatic techniques. This framework enables a mutual session with the user. The clustering algorithm does the processing, and the user validates the results.

2.6 Consistency of Clustering Techniques

We were hardly able to find any research works in Software Clustering literature that has formally compared the consistency of the results generated by different software clustering approaches. In this manner, our effort is one of the very first ones in this domain that compares consistency in results, an important and desirable property of any algorithm. To show the importance of our work we present some earlier work from Software Engineering field that highlights the significance of consistency in results.

Consistency is an important step toward stability of the clustering algorithm. Tzerpos and Holt [48] defined a stable clustering algorithm as one whose output does not change significantly when its input software system is slightly modified. From this definition it is clear that an inconsistent algorithm, that is showing large variations even for the same input in different runs, can not be stable; hence, consistency is important for stability. The results presented in this thesis have shown the consistency of our approach, ESBASCA.

Olson and Wolform [49] explained the importance of consistency in Information Architecture. They presented an approach to indexing that selects names and topic in manner that gives consistent and effective retrieval.

Monge, Marco and Cervigón [50] discuss the significance of consistency in context of Software Measurement Methods. They have defined a homogeneous statistic that

indicates how consistent a software measurement method is. They also provided a statistical analysis that compares given measurement methods and tells which one is more consistent.

Chapter Three: Overview of Software Clustering Process

This chapter presents an overview of software clustering techniques based on source code. Using source code as input to the clustering process is a good idea because source code is usually the most up-to-date documentation of a software system.

A survey paper by Wiggerts [51] describes three fundamental issues that need to be addressed while designing clustering techniques:

1. Representation of the entities.
2. Criteria for measuring the similarity between the entities.
3. Clustering algorithms.

3.1 Representation of Source Code Entities

While clustering software systems, a variety of decisions have to be taken in order to determine the representation of entities and relationships in the software system. First, one has to decide the granularity level of the recovered system design i.e. whether the entities would be procedures (methods) or modules (classes). Next decision is whether the relationships among the entities should be weight or not. Weights are helpful to signify certain special types of dependencies among the software system entities.

For instance, whether two entities are more related if they use a common global variable? How should the weight of such relationship compare to a pair of entities that

have a relationship that is based on one entity using the public interface of another entity?

In the *RIGI* tool, the user has the option to lay down the criteria to calculate the weight of the relationships among entities.

3.2 Similarity Measurements

After establishing the type of entities and relationships of the software system, the next step is to determine the similarity criteria among the entities. For this purpose similarity measures are used. Large values of these similarity measures depict a stronger similarity between the entities.

Based on the in their input, the similarity measures can be categorized in two groups:

1. ***Object Relationships***. In this case, graph representation can be used, where the nodes are the objects and the edges are the relations between the objects. In case of more than one relation the graph will have multiple kinds of edges. Generally the similarity measures dealing with such cases are the number of edges in the path between two objects, the length of shortest path between the objects or the weights of different types of edges. Another important factor is whether the graph is directed or undirected.

2. ***Score of objects on different Edges***. Similarity in this case, is commonly usually measured by means of *association coefficients.* Number of features available for each object is used to specify these association coefficients. That is why these coefficients are of binary type i.e. they reflect whether a feature is available or not.

Table 3.1 is used to compute various coefficients between two objects ***i*** *and* ***j***:

	Object *j* **1**	Object *j* **0**
Object *i* **1**	a	b
Object *i* **0**	*c*	*d*

Table 3.1: Classification of Software Features

In the table 3.1, the variable ***a,*** specifies the count of features present for both objects, ***b*** denotes the number of features present only for object ***i*** and so on. Different coefficients deal with 0—0 matches (whose number is given by ***d)*** in *different* manner.

Similarly different coefficients use different weights for of the four entries of the table. The most common coefficients are:

1. Simple Matching Coefficient**: (a+d) / (a+b+c+d)**
2. Jacard Coefficient: **a / (a+b+c)**

An extensive study of *coefficients* can he found elsewhere [10].

Other similarity measurements can be categorized as:

- **Distance Measures:** determine the dissimilarity between two entities.
- **Correlation Coefficients:** use statistical correlation to determine the similarity between two entities.
- **Probabilistic Measures:** assign significance to rare features shared among entities

3.3 Clustering Algorithms

After discussing and the representation and similarity measures in the previous sections, we focus or attention, in this section, on describing some common software clustering algorithms.

Most of the software clustering algorithms found in literature can be classified in the following three classes:

1. Hierarchical Algorithms.
2. Partitional Algorithms.
3. Graph-based Algorithms.

In the following subsections we present each class in detail.

3.3.1 Hierarchical Algorithms

The hierarchical algorithms yield a nested sequence of partitions. One extreme of this sequence is the partition where every entity lies is in a different cluster and at the other extreme is the partition where all the entities are placed in the same cluster. Starting from the first extreme, at each step clusters are joined together until the other extreme is reached. Figure 3.1 shows an example partition sequence for four entities M0, M1, M2 and M3.

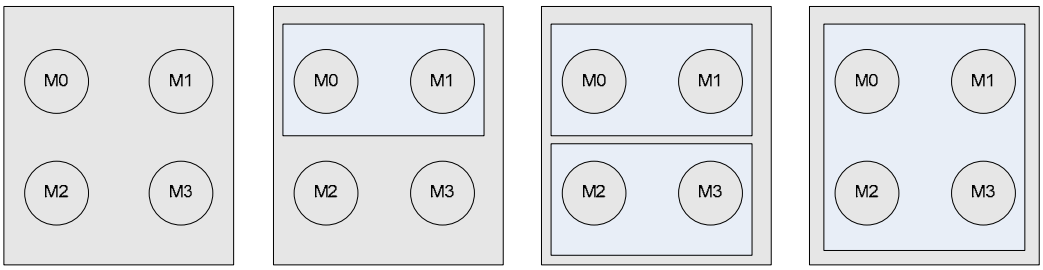

Figure 3.1: An Example Partition Sequence

The hierarchical structure is usually represented by a *dendrogram*. The example partition sequence is represented in a *dendrogram* in Figure 3.2.

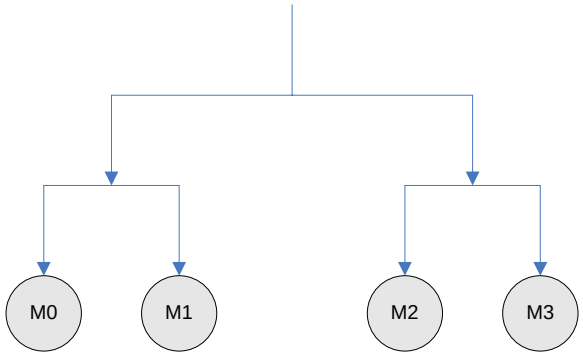

Figure 3.2: Dendrogram for the Example Partition Sequence

All the partitions in a sequence do not have equal significance. Rather, only a few of them (maybe only one) are of use. Finding the desired partitions is commonly termed as finding *cut points* of the dendrogram. Prior knowledge of the structure and parameters driving the clustering such as maximum number of clusters or the number of entities in a cluster, are the factors the impact the selection of *cut point*.

Two subclasses of hierarchical algorithms are:

1. Agglomerative Algorithms.
2. Divisive Algorithms.

1. ***Agglomerative Algorithms***. These algorithms start with all entities as separate partitions and then iteratively keep on join the most similar clusters based on the similarity measure. A point of debate in such

algorithms is about evaluating the similarity between a newly formed cluster and the rest of the already formed clusters. This is known as the update rule problem. Researchers have proposed different solutions for this problem.

Single Link Update Rule states that the similarity of the newly formed cluster to an existing cluster C is the *maximum* of the similarities of its constituents to *C.*

Complete Link Update Rule states that the similarity of the newly formed cluster to an existing cluster C is the minimum of the similarities of its constituents to C.

2. ***Divisive Algorithms.*** These start with all entities in a singular partition and try to iteratively split the partition until all entities are placed as different partitions. Computational performance is a major concern for these algorithms as exponential numbers of partitions are possible at every step. For this reason these algorithms have failed to achieve much popularity.

3.3.2 Partitional Algorithms

Partitional algorithms generally start with an initial partition and try to modify it in a bid to optimize the quality of a given partition. Different criteria can be used to define the quality of a partition and this is highly subjective. It usually is domain dependant. Example of such a criterion is maximization mathematical expression depending upon the maximization of cohesion and minimization coupling among the clusters. A major challenge faced by these algorithms is that the number of possible partitions is very

large. For example, there are 34,105 partitions of ten objects into four clusters, but this number explodes to approximately 11,259,666,000 if the number of objects is increased to 19 [21].

The common workaround to this problem is to heuristically select an initial partition and attempt to optimize the quality criterion by modifying that partition in an appropriate way. Such hill-climbing algorithms [10] do converge to the local optima. Therefore, the choice of the initial partition is vital for the success of the algorithm.

ISODATA [10] is popular partitional algorithm. Well calculated initial choice of value for seven parameters, controlling the factors such as the number of expected clusters, the number of entities in a cluster, is the basis of its effectiveness. Then, depending on closeness of the actual values of the current partition to the chosen parameters, the algorithm iteratively improves the initial partition by operations such as joining or splitting the clusters. Software clustering literature contains several variations of this method.

3.3.3 Graph-based Algorithms

Another important class of software clustering algorithms is of those based on the ideas of graph theory. Several categories of such techniques exist depending on the perspective [51, 52]. Some of them are presented here:

1. ***Minimum Spanning Tree Algorithms.*** These algorithms start by computing the minimum spanning tree (MST) of the given graph. Then, they either iteratively join the two closest nodes into a cluster or split the graph into clusters by removing inconsistent edges. Researchers have differences over the

definition of an inconsistent edge. But it is generally agreed that they usually carry considerably larger weight than the rest of the edges of the MST.

2. ***Clique Algorithms***. These algorithms either treat the maximal complete sub graphs (cliques) of the given graph as clusters or use them as the basis for other algorithms.

3. ***Local Connectivity Algorithms***. These algorithms use the number of edge or vertex disjoint paths of a specified length between two points as the criterion to decide which entities (represented as nodes in the graph) to place in the same cluster. For instance as described in [53], rather than only using single edges (paths of length 1) as a measure of similarity, multiple edges (e.g. paths of length 2) can also be used.

4. ***Aggregation Algorithms.*** These algorithms join nodes into aggregate nodes which can either be used as clusters or can be used as input for a new iteration to find higher level aggregates. Gregor in [54] presents Graph reduction, an aggregation technique, based on the concept of node's neighbourhood. Bi-components and strongly connected components have also been used for this purpose [55].

5. ***Heuristic Approaches*.** As already discussed in earlier sections, the large number of possible partitions means the graph partitioning problem is almost impossible to solve optimally. This is where heuristic approaches come into play. They attempt to cleverly search the possible solution space to come as close to the optimal solution as possible within the time constraints. The

Kernighan-Lin method [56] tries to overcome the local optima problem of hill-climbing algorithms by opting to go *downhill* for a while in an attempt to find a *taller hill* in the next few steps.

3.4 Observations

So far we have presented significant software clustering research and discussed several well-known clustering techniques. We have made the following observations:

- Earlier software clustering research work was directed at a low granularity level i.e. clustering procedures into modules. Along with the advancement in software engineering the research focus has changed to clustering modules and into higher-level abstractions such as subsystems.

- One of the important issues in software clustering is the selection of appropriate algorithms. A potential research initiative is a comparative study based on a number of software systems. Some algorithms may be found to It is suit particular type of software systems. A classification of the algorithms and the types of software for which they work best would be beneficial to the software clustering community**.**

- Majority of clustering algorithms have performance issues due to their computational intensive nature. Partitioning the modules dependency graph has been proved to be a NP-hard problem [21]. In order to achieve results in polynomial time, most researchers have adopted the use of heuristics that reduce this computation complexity.

- Much of the clustering research uses the software engineering concepts of coupling and cohesion as the criteria to compute the quality of the decompositions produced by the clustering process. Low coupling and high cohesion [74] are generally recognized properties of well-designed software systems.

- Software clustering researchers generally have resorted to use expert opinion in order to validate the results of their algorithms. This approach is quite subjective and though such assessment provides insight into the quality of software clustering results, formal methods are required to validate these results. Antquetil and Lethbridge [30, 45] have addressed this problem by defining the *precision and recall* measurement, which investigates the clustering results against expert decompositions of the software system. This approach is much better than the expert opinion the software clustering results. Another such approach has been presented by Tzerpos and Holt [57]. They have presented a distance metric, *MoJo*, to evaluate the similarity of two decompositions of a software system.

- Most of the researchers admit that software system deterioration, as it goes through maintenance, is a fundamental software engineering problem. To address this problem Tzerpos and Holt [35] have proposed an *orphan adoption technique* to incrementally update existing system decomposition by evaluating the impact of source code changes on the software system structure.

A problem with this technique is that it only investigates the impact of the changed modules on the existing software system structure. As the remaining modules and relationships of the system are ignored, the software system structure can deteriorate repeated *orphan adaptation*. The solution to this periodical re-modularization of the complete system.

- Most of the clustering techniques have not been tested on large systems. Testing on large software systems is vital for the validation of software clustering approaches. Open source systems are good option for this as access to large industrial systems is not easy.

In the next section of this chapter we introduce source code analysis and software visualization, which are two important bodies of work that are related to software clustering.

3.5 Source Code Analysis and Visualization

Clustering tools typically rely on source code analysis and visualization tools as shown in Figure 3.3.

Clustering tools enable users to provide software systems' source code as input to the clustering process. Generally these systems are large and complex and it is virtually impossible possible to manually transform the source code into the representation (e.g., MDG) required by the clustering tool. Manual transformation is tedious and prone to errors. Furthermore, the researchers have to cluster the same software system

iteratively by varying the clustering parameters without having to revisit the original source code.

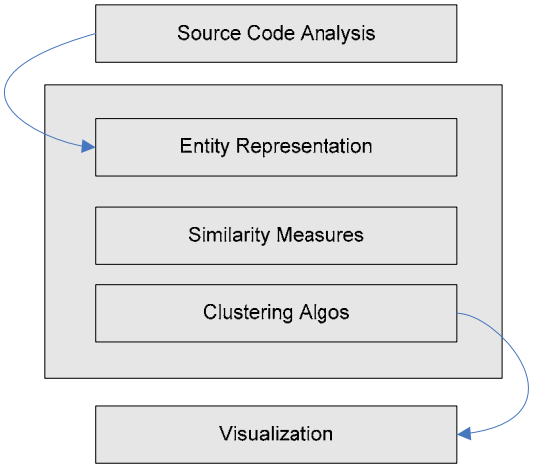

Figure 3.3: The Relationship between Source Code Analysis and Visualization

Visualization is another supporting technology needed by the software clustering process. The clustering results usually contain a large amount of data to be presented to users. Effective visualization techniques assist to present the results of the clustering process in a useful way.

3.5.1 Source Code Analysis

Source code analysis has been an active area of research for a long time. The primary reason for this is that is useful to consolidate a system's source code into a single repository that can be later be used for a variety of purposes like program comprehension and reverse engineering. Repositories allow the researchers to explore

the program structure by navigating through the complex dependencies existing among software system modules. Source code analysis tools have traditionally been used in tasks like dead code detection, program slicing, and reachability analysis. Researches in the area of software clustering have also found these tools to be helpful in their work.

Source code analysis tools parse the source code and populate their repository. Then, the repository can be queried to extract structural information. Internal structure of the repository can be organized either as a relational database or by storing the abstract syntax tree of the software system Source code analysis tools support a variety of programming languages like COBOL, C, C++, Java and Smalltalk.

We used the source code fact extractor developed by [58]. It extracts relationships and their weights. The relationships taken into account are those based on inheritance, containment, genericity and member access. The fact extractor provides MDG information in a matrix form. For example, the MDG of a small software system is shown in Figure 5.1 and the information extracted by the fact extractor is shown in Table 5.1.

3.5.2 Visualization

Graph visualization tools present graphical results in an organized manner. Software clustering systems usually process large and complex systems. The clustering results of such systems contain a large amount of data. So graphical representation of these results is an easily interpretable manner is a difficult task.

AT&T's *DOT* [59] is a powerful graph visualization tool that has been used by researchers. Dot has its own graph description language. Users specify the nodes, the edges between them and the attributes to steer appearance of the resultant graph in a text file. The *DOT* documentation [59] contains the description of different options that can be included in the input file. It is a command line tool that accepts a description file as input, and produces output in the specified format. It supports twelve output file formats (e.g., GIF, JPEG, PostScript, etc.). It has a powerful layout engine that increases the clarity of the resultant graph by edge routing and edge crossing minimization operations.

Another tool by AT&T, called *DOTTY* [59], is an online graph viewer, that can visualize and edit a graphs specified in the dot description file. It supports many visualization functions such as *bird eye views* and *zooming*.

We use this tool to present our results visually in this thesis.

Chapter Four: Evolution Strategies

Before we present our evolution strategies based software clustering approach we find it important for the reader to have an overview of Evolution Strategies, the operators involved and a generic ES algorithm in this chapter.

Evolution Strategies is a specialization of evolution algorithms. These are nature inspired optimization methods that apply selection and genetic operators to a population of individuals to evolve better solutions in an iterative manner. Every individual in the search space represents a potential solution. Each iteration is called a generation and in each generation a new population is created using the fittest individuals in the preceding generation. The operators, the idea of self adaptation and the generic ES Algorithm is presented in the following subsections.

4.1 Objective Function

The quality of solution is calculated using a problem dependent objective function that defines the fitness value (quality) of each member of the population. The function is designed in such a manner that an individual with higher fitness represents a better solution than an individual with a lower fitness. We define the objective function for our software clustering problem in the next chapter.

4.2 Operators

ES typically uses selection, mutation and recombination operators to guide the search. We present each of them briefly in the following subsections:

4.2.1 Selection

Evolutionary algorithms need objective oriented selection operator to steer the search into promising regions of the object parameter space. Selection is, therefore, unlike the genetic operators of mutation and recombination. It guides the evolution. Selection in ES follows the phenomenon that only the individuals with promising properties, i.e. individuals with high objective function values get a chance to breed. This truncation selection in ES guarantees that only the μ best individuals from the current generation *(g)* are transferred to the next population at *(g + 1)* by means of a *deterministic* process. Beyer [22] represents the population at (g+1) is represented as

$$\beta_{p}^{(g+1)} := \{ \alpha_{1;\gamma}, \ldots, \alpha_{\mu;\gamma} \} \qquad (4.1)$$

where

$\alpha_{\mathbf{m;\gamma}}$:= "*m*th best individual from γ individuals."

Based on whether or not the parent solutions compete for inclusion into the next generation, the selection technique has two versions:

1. *Comma* Selection *(μ, λ)*.
2. *Plus* Selection *(μ + λ)*

In comma selection, only the λ newly generated offspring individuals define the selection pool. In other words, the parents from generation *(g)* are ignored by definition even if they are fitter than all offspring.

The Plus selection, on the other hand, takes the old parents into account. This means that *both* the parents *and* the offspring are copied into the selection pool which is therefore of size $\gamma = \mu+\lambda$.

Plus selection guarantees that fittest individual found so far, survives. For this reason such selection technique is termed as *elitist*. For the ES to converge globally, elitism is a sufficient condition Elitism allows the parents to survive an infinitely long time-span.

Both versions of the selection technique have their different application areas. While the *comma selection* is recommended for unbounded search spaces Y, especially $Y = R^N$ [60], the *plus selection* should be used in discrete finite size search spaces, e.g. in combinatorial optimization problems [61, 62].

4.2.2 Mutation

The mutation operator is the primary variation operator in ES. That is, it is the main source of genetic variation. The design of mutation operators depends on the problem, domain. Although there are not any design principles but [63] has proposed some rules based on theoretical considerations and investigation of successful ES implementations. They are:

1. Reachability.
2. Unbiasedness.
3. Scalability.

Reachability: This requirement states that any finite state in the search space can be reached within a finite number of mutation steps or generations. This is also a necessary condition for global convergence.

Unbiasedness: This requirement has been derived from Darwin's theory of evolution. Selection and mutation have different purposes. Selection uses the fitness values in order to guide the search into promising areas in the search space. Variation, on the other hand, *explores* the search space, i.e. it is based on search space information of the parent population rather than using any fitness information. This means that variation operator should not incorporate any bias by giving preference to any selected individual. This is a basic design rules for the variation operators. This ends up at the *maximum entropy principle* whose application leads immediately to the normal (Gaussian) distribution in the case of *unconstrained real-valued search spaces* R^N. Rudolph [64] has shown that this principle suggests the geometrical distribution in case of *unconstrained integer search spaces* Z^N. Other cases have not been investigated so far.

Scalability: The *scalability requirement* states that the mutation strength should be tunable in order to adapt to the fitness landscape. Adaptation ensures the *evolvability* of the ES algorithm along with the objective function. *Evolvability* expresses that the variations should be generated in such a way that improvement steps are possible thus building a *smooth* evolutionary random path through the fitness landscape toward the optimum solution [64]. As the objective function along with variation operators defines the properties of the fitness landscape, the *smoothness* of the fitness landscape

becomes a prerequisite of efficient evolutionary optimization. The smoothness assumption is sometimes expressed in terms of the *causality concept* [66] stating that small changes on the genetic level should result on average in small changes in the fitness values.

As the evolvability can not be ensures independently, we have to rely on scalability that can be guaranteed in real-valued search spaces.

4.2.3 Recombination

While mutation performs search steps based on the information of only one parent, recombination shares the information from up to p parent individuals [66, 67, 68]. $p > 2$ means multi-recombination.

The recombination operator in ES produces only one offspring from a family of size p. This is in contrast to the crossover operator in GA [24] that produces two offspring from two parents. Generally two types of are recombination used in ES:

1. Discrete (or dominant) recombination.
2. Intermediate recombination.

Discrete Recombination: Suppose $\mathbf{a} = (a_1, \ldots, a_D)$ is the parent vector (object or strategy parameter vector), the discrete recombination produces a recombinant vector $\mathbf{r} = (r_1, \ldots, r_D)$ by coordinate-wise random selection from the p corresponding coordinate values of the parent family [22]

$$(\mathbf{r})_k = (\mathbf{a}_{mk})_k \qquad (4.2)$$

where

$$mk := \text{Random}\{1, \ldots, p\}$$

This means that the *k*th component of the recombinant is determined exclusively by the *k*th component of the randomly (uniformly) chosen parent individual *mk.*

Intermediate Recombination: In contrast to discrete (dominant) recombination the intermediate recombination takes all *p* parents equally into account. It simply calculates the center of mass (centroid) of the *p* parent vectors $\mathbf{a}_m$ [22]

$$(\mathbf{r})_k := \sum_{m=1}^{p} (\mathbf{a}_{mk})_k \qquad (4.3)$$

The procedure defined in Eq. 4.3 is for real-valued state spaces. Supporting procedures like probabilistic rounding are require, for application in discrete spaces, in order to map back onto the discrete domain.

4.3 Self Adaptation

Self Adaptation is an important feature of Evolution Strategies. We explain this in detail.

4.3.1 Introduction

Evolutionary algorithms operate on basis of population of individual solutions. They are highly dependant on the characteristics of the population distribution in order to perform well. The objective of self adaptation is to bias the distribution towards

promising regions of the search space. This is achieved by introducing enough diversity among individuals to facilitate further evolvability.

Generally, diversity is introduced by adjusting the values of the control parameters. Control parameters can be the mutation rates, recombination probabilities, or the population size.

The goal is to efficiently find suitable adjustments. This is further complicated due to the dynamic nature of evolutionary algorithms. A parameter setting suitable at the beginning may become sub optimal during the iterations. That is why adaptation of the control parameters, during the iterative run of an evolutionary algorithm, is required.

Population individuals represent possible solutions. These are represented as sets of object parameters that can be interpreted as the genome of the individual. The basic idea of explicit self-adaptation is that the strategy parameters themselves are evolved along with the object parameters.

4.3.2 Self Adaptation in Evolution Strategies

As far as the evolution strategies are concerned, the need to adapt the mutation strength during the evolutionary process was recognized 1973 in Rechenberg's book *Evolutionsstrategie*[66].

He introduced the famous 1/5th rule, which was originally developed for (1 + 1)-ES. For a certain number of generations, it keeps track of the mutations that results the

improvement in fitness value, termed as successful mutations. If more than 1/5th of mutations are successful, then the mutation strength is increased, otherwise it is decreased. The aim was to keep within the *evolution window* where optimal progress is ensured.

Besides the 1/5th rule, Rechenberg [66] also proposed to couple the evolution of the strategy parameters with that of the object parameters. This gave birth to the idea of explicit or self adaptation. Rechenberg conducted experiments on sphere and corridor model to compare the performance of self adaptation with the 1/5th rule. Self adaptation not only demonstrated higher convergence rates but also proved to be applicable in scenarios where it was not possible to use the 1/5th rule. Hence, self adaptation emerged as a more universally usable method.

Self Adaptation of Strategy Parameters: In the paradigm of evolution strategies, the technique most commonly associated with the term self adaptation was introduced by Rechenberg [68] and Schwefel [69, 70]. The strategy parameters considered in this technique apply to the mutation process and parameterize the mutation distribution.

4.4 The Generic ES Algorithm

ES applies the above defined operators to a population in an iterative process. The generic algorithm is outlined here:

1. Take an initial population of x individuals.
2. Generate y offspring, where each offspring is generated in the following manner:

   a. Select z parents from x (z is a subset of x).
   b. Recombine the z selected parents to form a new individual i.
   c. Mutate the strategy parameter (adaptation).
   d. Mutate the individual i using the mutated strategy parameter.
3. Select new parent population consisting of x best individuals (based on objective function) from the pool of x and y.
4. Go to 2, until termination condition occurs.

Chapter Five: ES Based Automated Software Clustering

In this chapter, we present the instantiation of ES Algorithm for the software clustering problem.

Figure 5.1 shows an example weighted MDG of small system that we will use to explain the concepts throughout this chapter.

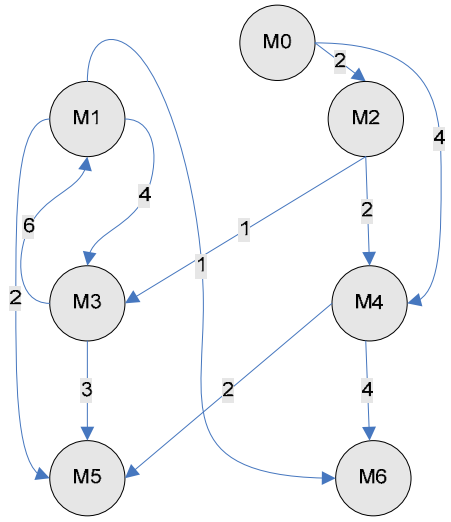

Figure 5.1: An Example Weighted MDG

Figure 5.2 is a decomposition of the MDG presented in Figure 5.1

5.1 Variable Selection

Software Clustering Algorithms need to be independent of the programming language syntax. For this purpose, source code analysis tools are used to transform the source

code of the software system under study into a language independent format. This representation contains all of the entities and the relationships/dependencies among the entities.

Evolution Strategies work on population of individuals. In software clustering problem we have three types of variables which affect resolution of the problem. These variables are discussed below.

Entities: First, the entities involved, which in this case are modules of the system. We represent these modules with indices from 0 to n-1.

Relationships: Second variable is the set of relationships among these modules. We used a third party fact extractor [58] that provided us with these relationships among modules and their weights. The relationships taken into account are those based on inheritance, containment, genericity and member access. Figure 5.1 shows the modules and relationships of a small software system extracted by [58]. Table 5.1 shows the relationship matrix, which is in fact used as input to the clustering algorithm, for the same software system.

Subsystems: Third variable is the subsystems (clusters) which comprise of these modules. These subsystems are represented by 0 based indices. Therefore a system can have minimum one cluster and maximum n (equal to total number of modules in systems) subsystems.

	M0	M1	M2	M3	M4	M5	M6
M0	0	0	2	0	4	0	0
M1	0	0	0	4	0	2	2
M2	0	0	0	1	2	0	0
M3	0	6	0	0	0	3	0
M4	0	0	0	0	0	2	4
M5	0	0	0	0	0	0	0
M6	0	0	0	0	0	0	0

Table 5.1: Relationship Matrix for the Example MDG

To solve the software clustering problem using Evolution Strategies, we use the variables defined above, to represent the search space population.

5.2 Population Representation

Every individual solution of the software clustering problem is represented by an encoded string of integers. This encodes string is generated by assigning a cluster number to each entity. For instance the decomposition shown in Figure 5.2 can be represented as [1 0 1 0 2 0 2]. This encoding means that

- Cluster 0 contains modules 1,3 and 5;
- Cluster 1 contains modules 0 and 2;
- Cluster 2 contains modules 4 and 6.

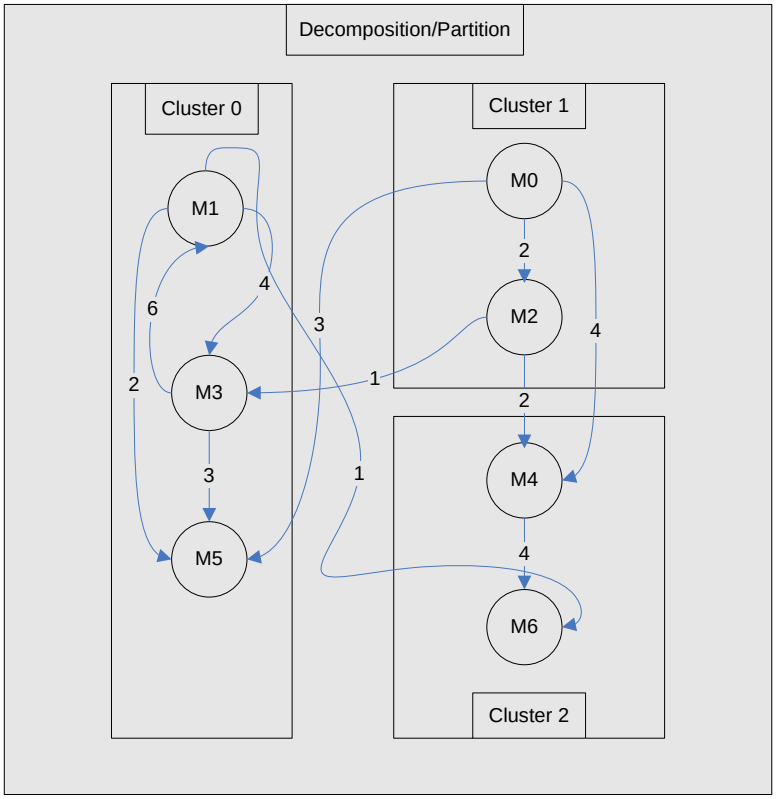

Figure 5.2: A Sample Decomposition for the Example Weighted MDG

5.3 Objective Function

The major goal of software clustering algorithms is to find a partition/decomposition of software systems in the form of subsystems that expose abstractions of the software system structure. Finding *good* partitions involves navigating through all the possible partitions of the search space. To achieve this, we treat software clustering as an optimization problem based on maximizing the value of an objective function.

The objective function is derived using the variables involved in the system. We are proposing the use of operators and algorithm presented in Chapter 4 on an objective function based on software engineering concepts of coupling and cohesion.

Cohesion measures that how deeply-related and focused the various responsibilities of a software subsystem are. Subsystems with high cohesion are preferable because high cohesion is has several attractive features of software including reusability, reliability, understandability and robustness. On the other hand, low cohesion is associated with undesirable qualities such as raising difficulties in maintaining, testing, understanding and reuse.

Coupling is the measure to which extent each subsystem relies on the other subsystems. Coupling is usually disparate with cohesion. High coupling usually correlates with low cohesion, and vice versa.

It is generally considered that subsystems exhibiting high cohesion and low coupling form well designed systems. Hence, the resulting decompositions should have more intra-cluster relationships and less number of inter-cluster relationships. To achieve this property we use the objective function *Turbo MQ*, used and defined in [21].

For each cluster we calculate two quantities: intra-connectivity and inter-connectivity. μ_i, which refers to the *Intra-connectivity* of a cluster i is the weighted sum of all relationships (provided by the fact extractor) that exist between modules in that cluster i. A higher value of *Intra-connectivity* corresponds to high cohesion. ϵ_{ij}, that

refers to inter-connectivity is the weighted sum of all relationships (provided by the fact extractor) that exist between modules in two distinct clusters i and j. This quantity can have values between 0 (when there are no subsystem level relations between subsystem i and subsystem j) and 1 (when all modules in subsystem i are related to all modules in subsystem j and vice-versa). A low value for inter-connectivity means low coupling.

Using these two quantities, a cluster factor $\boldsymbol{CF_i}$ is calculated for each cluster i and total fitness of the system is given by the sum of **CF** for all clusters. The cluster factor is calculated as:

$$CF_i = \begin{cases} 0 & \text{if } \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum_{j=1, i \neq j}^{k} (\epsilon_{i,j} + \epsilon_{j,i})} & \text{otherwise} \end{cases}$$

Total Fitness is given by:

$$TurboMQ = \sum_{i=1}^{n} CF_i$$

For the decomposition given in Figure 5.2, we present the calculation of the individual cluster factors and that of TurboMQ, as an example.

Number of Modules in Cluster 0 = N_0 = 3

Number of Modules in Cluster 0 = N_1 = 2

Number of Modules in Cluster 0 = N_2 = 2

Calculations for Cluster 0:

μ_0 = 6+4+3+2 =15

$\in_{01}$ = 0 *No edge from any module in Cluster 0 towards any module in Cluster 1*

$\in_{10}$ = 4

$N_0 * N_1 = 6$

$\in_{02}$ = 1

$\in_{20}$ = 0 *No edge from any module in Cluster 2 towards any module in Cluster 0*

$N_0 * N_2 = 6$

CF_0 = (2*15)/(2*15)+[{(0+4)/6}+{(1+0)/6}] = **0.97**

Calculations for Cluster 1:

μ_1 = 2

$\in_{10}$ = 4

$\in_{01}$ = 0 *No edge from any module in Cluster 0 towards any module in Cluster 1*

$N_1 * N_0 = 6$

$\in_{12}$ = 6

$\in_{21}$ = 0 *No edge from any module in Cluster 2 towards any module in Cluster 1*

$N_1 * N_2 = 4$

CF_1 = (2*2)/(2*2)+[{(4+0)/6}+{(6+0)/4}] = **0.65**

Calculations for Cluster 2:

μ_2 = 4

$\epsilon_{20} = 0$ *No edge from any module in Cluster 2 towards any module in Cluster 0*

$\epsilon_{02} = 1$

$N_0 * N_2 = 6$

$\epsilon_{21} = 0$ *No edge from any module in Cluster 2 towards any module in Cluster 1*

$\epsilon_{12} = 6$

$N_2 * N_1 = 4$

CF_1 = (4*2)/(4*2)+[{(0+1)/6}+{(0+6)/4}] = **0.82**

TURBO MQ = $CF_0 + CF_1 + CF_2$ = 0.97+0.65+0.82 = **2.44**

Chapter Six: Implementation

We have developed a tool for automated software clustering using Evolution Strategies. As we also wanted to compare our approach some well known approach so our tool also provides Genetic Algorithms based software clustering. We wanted to use *BUNCH* tool [21] for GA based approach but neither we could get hold of *ACACIA* [72], the tool that is needed to generate input for Bunch, nor we were able to find any helpful documentation regarding the input format to generate the input for Bunch by ourselves. So we decided to implement the GA based approach as well.

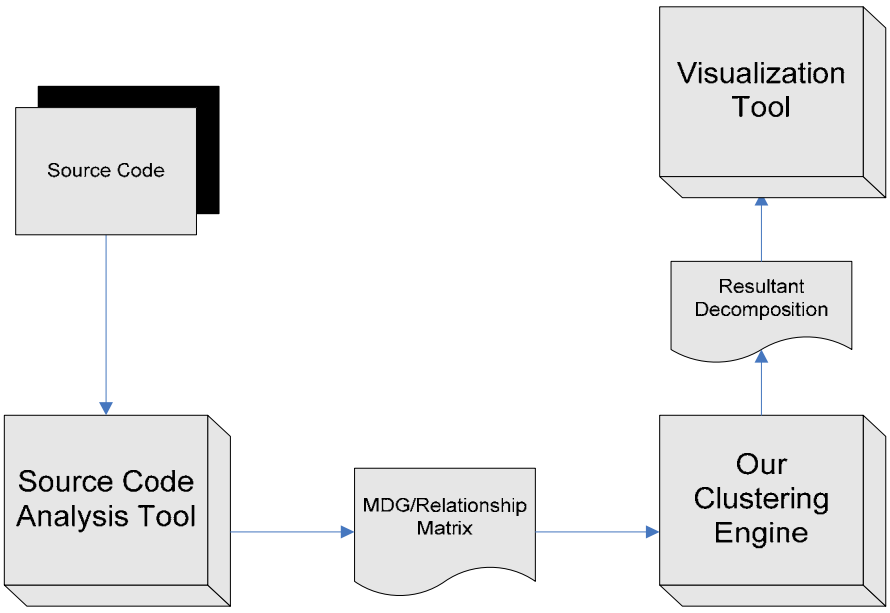

Figure 6.1: The Working Environment of Our Software Clustering Tool

6.1 Automated Clustering Using Our Tool

The first step in our automatic clustering approach involves converting the source code into programming language independent format e.g. module dependency graph (MDG). This MDG is then fed to our clustering engine in the form of module relationship matrix. The clustering engine then performs software clustering such that the clusters represent meaningful subsystems. The resultant decomposition of the clusters found in the previous step is then viewed using a visualization tool. Figure 6.1 shows the clustering environment of our tool.

6.2 Architecture of Our Tool

Figure 6.2 shows the architecture of our tool. We discuss these components one by one.

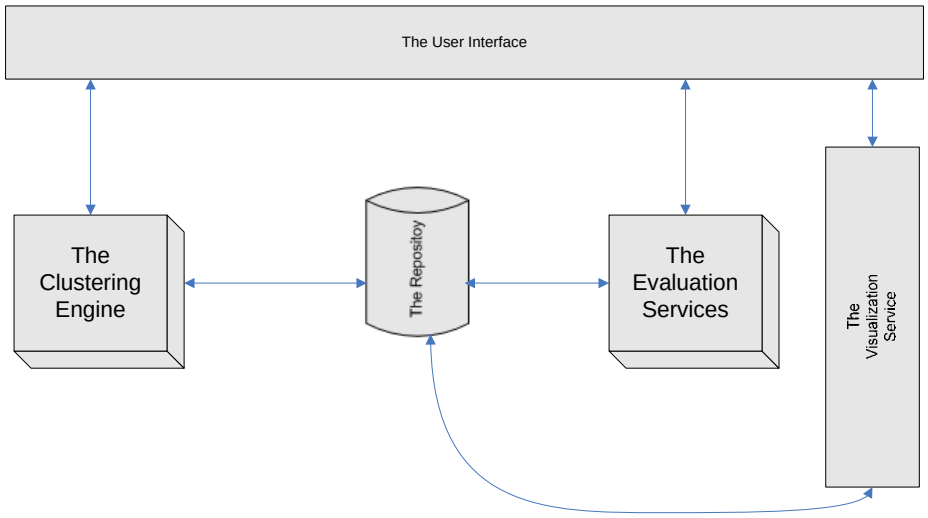

Figure 6.2: Architecture of Our Software Clustering Tool

6.2.1 The User Interface

The user interface of our tool, which is shown in Figure 6.3, collects information that is necessary to perform the clustering. The key information collected on the user interface is the path of the folder containing the project of software system to be clustered; the clustering approach to be used i.e. genetic algorithm or evolution strategy; clustering options like initial population size, maximum number of generations and number of clusters. We will discuss these options in detail in the next

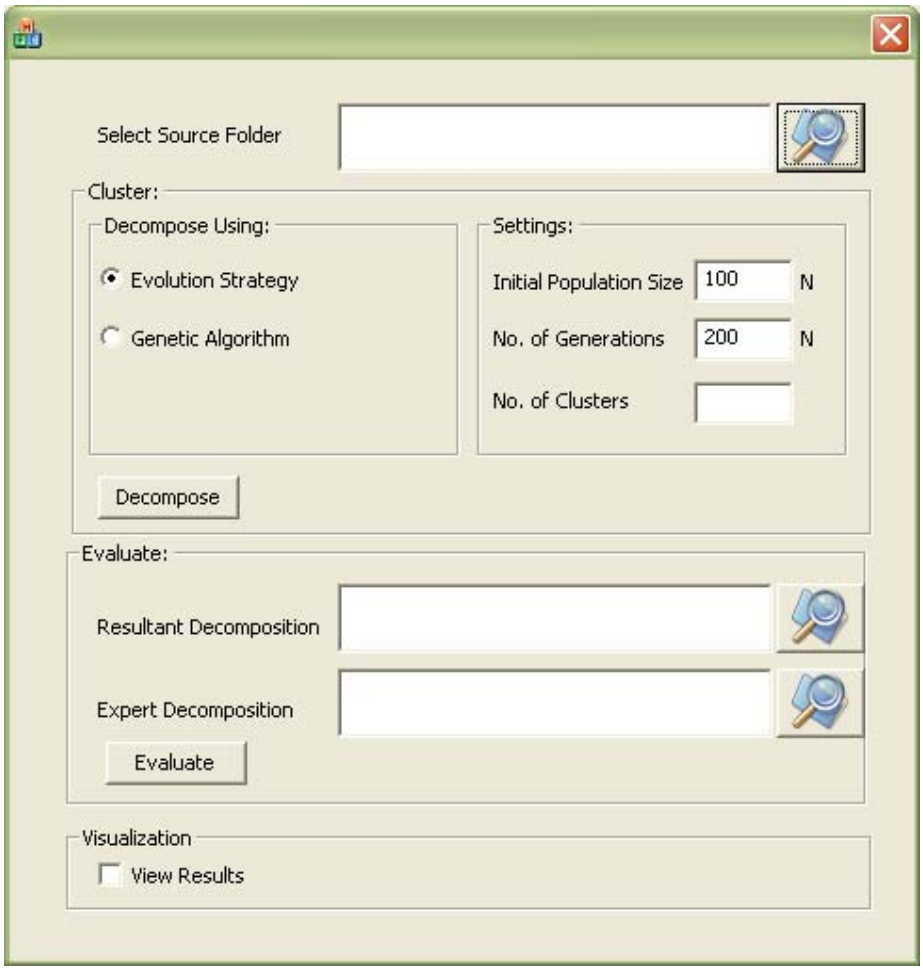

Figure 6.3: The User Interface of Our Software Clustering Tool

chapter. The user interface also provides options like the text file containing the expert decomposition, to evaluate the clustering results.

6.2.2 The Clustering Engine

Clustering Engine is the main component that provides the software clustering services. Clustering engine is composed of classes that provide methods such as *selection*, *mutation*, *recombination*, *cross-over* in order to support software clustering using the Evolution Strategies and the Genetic Algorithms. The flow chart in Figure 6.4 explains the working logic of our clustering engine.

We now explain the steps shown in the flow chart for both the clustering approaches.

Evolution Strategy Approach:

The following steps explain the Evolution Strategy approach towards software clustering.

1. Depending upon the *initial population size* specified by the user at user interface, an initial population of potential solutions is randomly generated which is saved in a *parent pool*.

2. Using the *objective function*[1], a specific number of fittest individuals, say ***x***, from the initial population are selected using *deterministic selection*[2]. The following procedure is then repeated until a specific number of offspring, say ***y***, are generated.

---

[1] *Defined and explained in Section 5.3.*
[2] *Defined and explained in Section 4.2.*

3. Using the *recombination operator*[3] a new individual is generated. The *mutation operator*[4] is then used to mutate the strategy parameter, which in this case is the *mutation strength*.

4. The mutated mutation strength is then used to *mutate* the newly generate individual. The result is then saved to an *offspring pool*.

5. When the offspring pool contains the ***y*** offspring, ***x*** individuals of the parent pool are replaced by ***x*** fittest individuals in the offspring pool.

6. Steps 2 to 4 are repeated until the *stopping criteria,* specified at the user interface, is fulfilled.

Genetic Algorithms Approach:

The following steps explain the Genetic Algorithms approach towards software clustering.

1. Depending upon the *initial population size* specified by the user at user interface, an initial population of potential solutions is randomly generated.

---

[3] *Defined and explained in Section 4.2.3.*
[4] *Defined and explained in Section 4.2.2.*

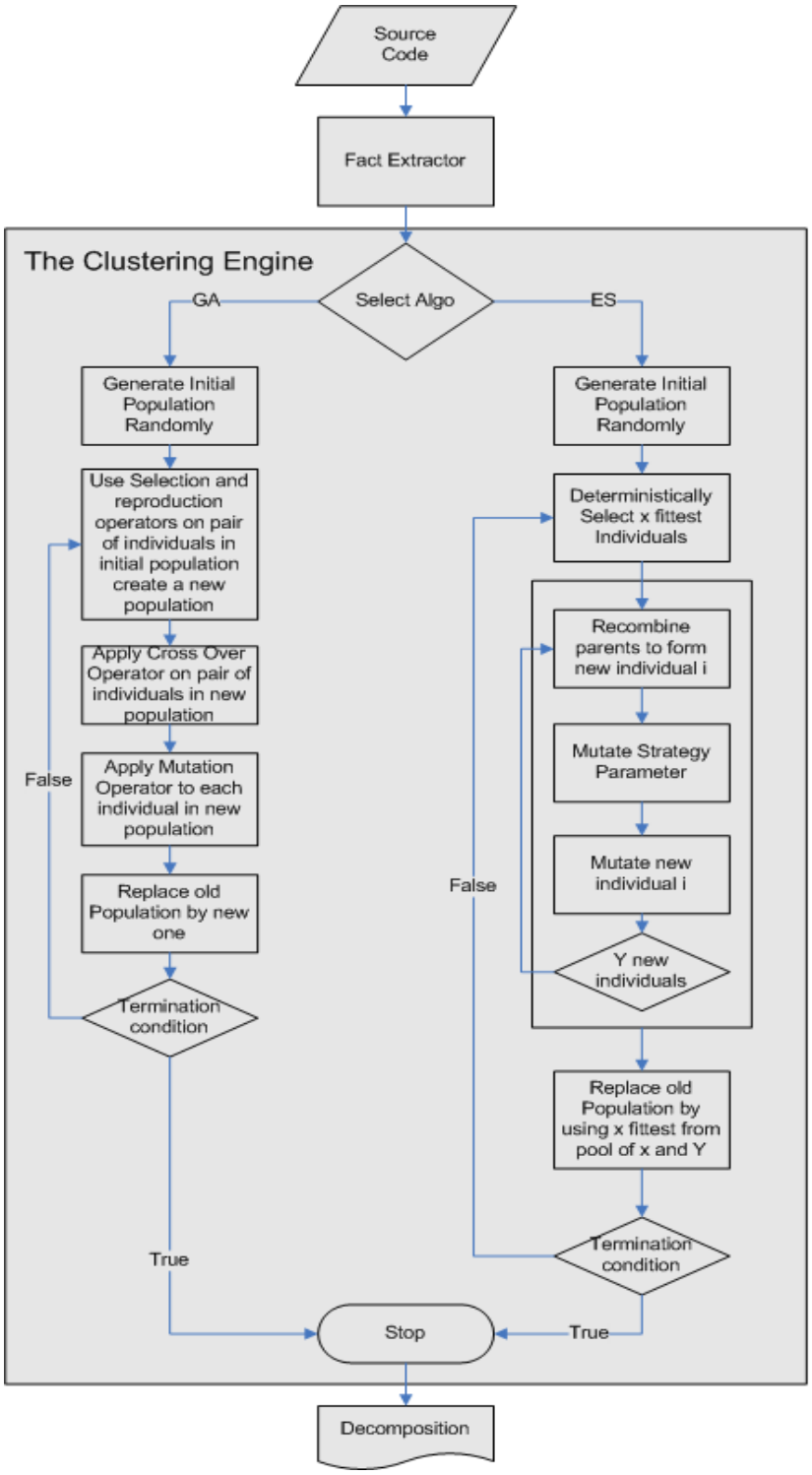

Figure 6.4: Working Logic of Clustering Engine

2. Using the *cross-over operator* [24] on pair of individuals to create new individuals.

3. Apply mutation operator on the new individuals

4. Replace the old parent population with the new one.

6.2.3 The Evaluation Services

This component provides the services to evaluate the clustering results. The decomposition produced by the clustering engine is compared against the expert/benchmark decomposition usually provided by the original designer of the system. We have implemented a similarity measure *Precision & Recall* [30, 45] for this purpose. We discuss this similarity measure in the next chapter.

6.2.4 The Repository

It is used to store the clustering results for later reference. It also contains expert decompositions. The major advantage of this repository is that the once a software system is clustered, its results are available for use in future without having to perform the clustering again. Also the repository is handy in situations where the user just wants to cluster the system and wants to defer the evaluation of the clustering results for future. So whenever evaluation is required, user can pick the clustering results and can evaluate them against corresponding expert decompositions which are also present in the repository.

6.3 Output Structure

The output of the clustering engine (i.e. a resulting decomposition) is actually an array of cluster with each cluster having three elements:

1. The list of modules that the cluster contains.
2. The sum of the weights of relationships that exist between the modules contained in the cluster.
3. The cluster factor computed for the cluster.

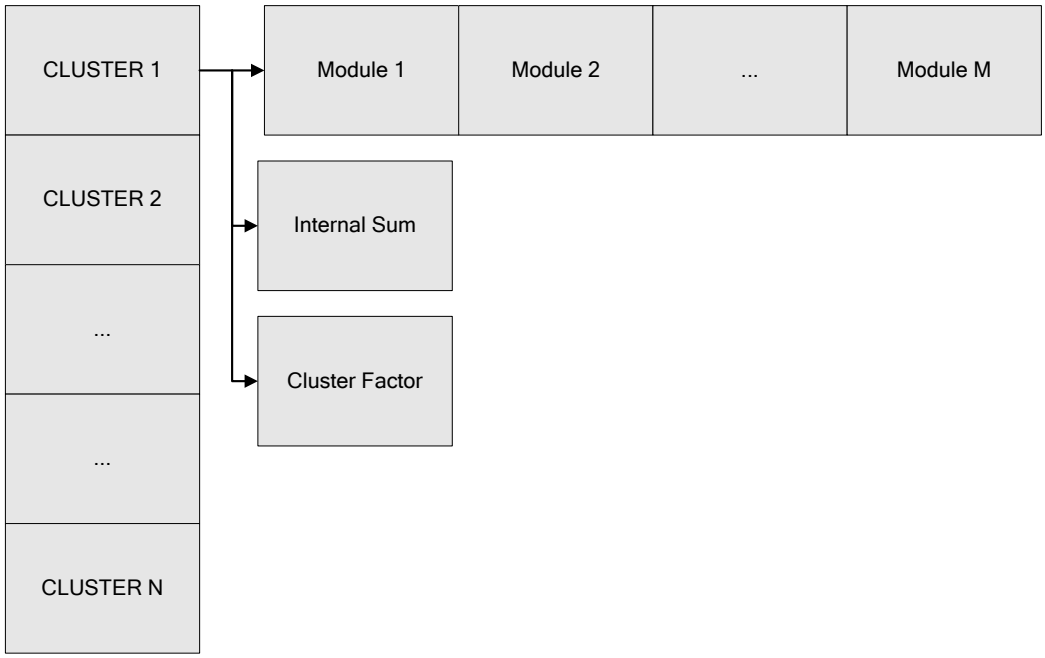

Figure 6.5: Output Structure

Chapter Seven: Experimental Results and Analysis

In the previous chapter we discussed the implementation of our software clustering tool. To establish the correctness of our approach we tested it on several test systems. We also wanted to compare our approach against a good and widely used GA based approach. This chapter presents our research results and the comparative study of our self adaptive evolution strategies based automated software clustering approach ESBASCA and the genetic algorithms based software clustering approach.

This section will first describe the interfaces exposed by the PDF parser that will be used by Client to get PDF objects. Second part will present complete guidelines for those who want to convert PDF into their own format.

7.1 Test Systems

This section presents the test system that we used for the verification of approach and the comparative study.

We used four medium sized industrial software systems in our study. We will use test system IDs in this thesis, instead of their names. These are object oriented systems implemented in C++. Implementation of these systems also involves software libraries *MFC*, *ATL* and *STL*. The classes residing in these libraries are not included in clustering process. However, they are used in fact extraction process and support building relationships among other software entities. For instance, say a software system contains two classes named X and Y which are derived from *CEdit* (which is

MFC based user interface related class). *CEdit* may be used to find relationship among X and Y, as both classes have same inheritance hierarchy. But *CEdit* will not be used in the clustering process and set of subsystems will not contain this class.

Each test system is introduced briefly in the following sections, each followed by a statistical summary. Details of the test systems and their module relationships can be found in [58]. The relationships taken into account are those based on *inheritance*, *containment*, *genericity* and *member access*.

7.1.1 Test System 1

Test System 1, from now onwards, referred to as TS-1, is a component of a large software system. It provides conversion support from intermediate data structures to a well known document format. Table 7.1 shows the entity related statistics for TS-1 whereas Table 7.2 shows relationship statistics for TS-1.

No. of Lines of Code	45582
No. of Header Files	53
No. of Source Files	39
No. of Modules	36
No. of Relationships	817

Table 7.1: Entity Related Information for TS-1

Relationships based on Inheritance	373
Relationships based on Containment	298
Relationships based on Generecity	27
Relationships based on Member Access	32
Other Relationships	87

Table 7.2: Relationships Information for TS-1

7.1.2 Test System 2

Test System 2, from now onwards, referred to as TS-1, is a software system solves economic power dispatch problem using conventional and evolutionary computing techniques. It uses MFC document view architecture and implements conventional and genetic algorithms. Table 7.3 presents the entity related statistics for TS-2 whereas Table 7.4 presents relationship statistics for TS-2.

No. of Lines of Code	16360
No. of Header Files	31
No. of Source Files	27
No. of Modules	41
No. of Relationships	473

Table 7.3: Entity Related Information for TS-2

Relationships based on Inheritance	102
Relationships based on Containment	166
Relationships based on Generecity	6
Relationships based on Member Access	127
Other Relationships	72

Table 7.4: Relationships Information for TS-2

7.1.3 Test System 3

Test System 3, from now onwards, referred to as TS-3, is a component of a large software system. It provides conversion support from intermediate data structures to a well known printer language. Table 7.5 summarises the entity related statistics for TS-3 whereas Table 7.6 shows summarises statistics for TS-3.

No. of Lines of Code	51768
No. of Header Files	27
No. of Source Files	27
No. of Modules	69
No. of Relationships	4973

Table 7.5: Entity Related Information for TS-3

Relationships based on Inheritance	251
Relationships based on Containment	379
Relationships based on Generecity	465
Relationships based on Member Access	254
Other Relationships	3624

Table 7.6: Relationships Information for TS-3

7.1.4 Test System 4

Test System 4 from now onwards, referred to as TS-4 is a software system for design document layout and composition. It provides visual support to define document layout and complete saving and loading mechanism for designed applications. Entity related statistics for TS-1 are given in Table 7.7 whereas relationship statistics for TS-4 are given in Table 7.2.

No. of Lines of Code	82877
No. of Header Files	74
No. of Source Files	68
No. of Modules	80
No. of Relationships	4886

Table 7.7: Entity Related Information for TS-4

Relationships based on Inheritance	151
Relationships based on Containment	774
Relationships based on Generecity	59
Relationships based on Member Access	1174
Other Relationships	2728

Table 7.8: Relationships Information for TS-4

7.2 Testing Environment

We performed our testing on Win-XP platform on a machine with 3GHz Intel Pentium IV processor and 2GB RAM. Table 7.9 shows the parameters common to both ES and GA.

Parameter	Value
Initial Population Size	300
No. of Clusters	±2 of that proposed in ex pert decomposition
Termination Condition	3000 Generations or No Improvement in Fitness Value since last 300 Generations

Table 7.9: Common Parameters for ES and GA

Here, we find it important to discuss the common features i.e. *initial population size*, *the number of clusters* and the *termination criteria*.

Initial Population Size: The larger the initial population size the better is the chance of finding a near optimal solution. But due to computation intensive nature of these approaches we have to make trade-off between the initial population size and execution performance. So for our test systems we empirically found out 300 to be a good option.

Number of Clusters: It is not feasible to check all decompositions containing 1 to n clusters where n is the number of modules in the test system. So we adopted a strategy based on the checking the range of ±2 number of clusters proposed by the benchmark decompositions, provided by the designers of the test system. So we have five decompositions in all and we select the decomposition with the highest fitness as the final solution.

Termination Criteria: Another important decision is to chalk out an efficient termination criteria where again a trade-off has to be made between a good solution and execution performance. This also depends on the number of modules in the system and their relationships. We empirically found out that for the test systems used in this study, 3000 iterations is a good criterion as both ESBASCA and GA based approach converged within this limit. Rather ESBASCA converged well before this limit but we wanted to match our approach to the best possible results of GA based approach so we adopted this limit that favours GA based approach. The second

criterion is simply to stop the process when no improvement has been made for a long time.

It should be noted that all these parameters that guide our search can be changed by the user of our application. Table 7.9 shows the values that we empirically found after experimentation with the test systems under study.

Parameters specific to GA used for our tests are presented in Table 7.10. The available options and details for these GA specific parameters are in [24, 25].

Parameter	Value
Selection Method	Rank based Selection
Cross-over Probability	0.6
Mutation Probability	0.2

Table 7.10: GA-Specific Parameters

Parameters specific to ES used in our tests are presented in Table 7.11. The options and details of these parameters are in [22, 23].

7.3 Results and Discussion

We have compared the fitness value of the resulting decomposition of each test system by both ESBASCA and GA based clustering approach. The collected results are also compared with reference decompositions provided by the original designers

of the systems. The following sub sections present and discuss each category of these results.

Parameter	Value
Mutation Type	Mutation by Geometric Distribution
Exponent for the Geometric Distribution	2
Recombination Type	Discrete

Table 7.11: ES-Specific Parameters

7.3.1 Quality

Fitness value gives us the idea of how good is the decomposition according to a predefined objective function. Using the cohesion and coupling criteria given in Chapter 5, the Fitness values of the best decomposition found by both GA based clustering approach and ESBASCA for each test system was computed. Table 7.12 presents the fitness results for the four test systems using both the approaches.

A comparison of the fitness results of both approaches for all test systems averaged over ten runs is presented in Figure 7.1. It can be seen that ESBASCA yields much better results for all test systems; the improvement is in the range of 20-50%. The improvement in fitness value by ESBASCA as compared to GA based approach calculated for each test system is given in Table 7.13.

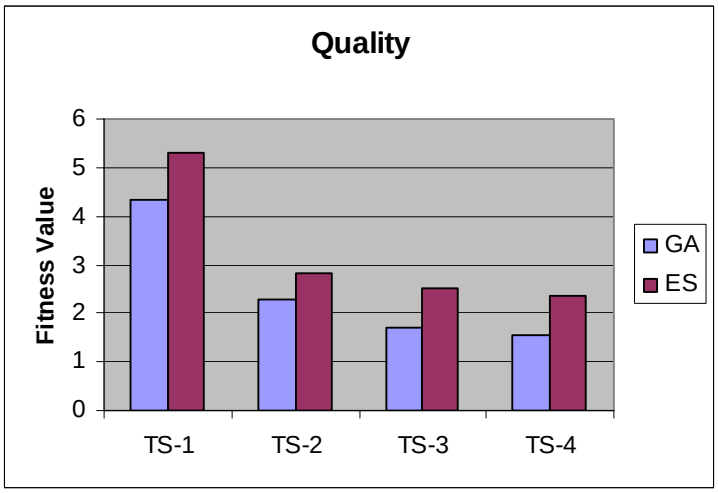

Figure 7.1: Comparison-Quality

Run#	TS-1		TS-2		TS-3		TS-4	
	GA	ES	GA	ES	GA	ES	GA	ES
1	4.89524	5.1129	1.81979	2.83333	1.57621	2.38468	1.61919	2.37529
2	4.08602	5.13399	2.61199	2.89983	1.94534	2.66388	1.36839	2.42168
3	4.69636	5.2451	2.70386	2.89983	1.54118	2.43608	1.27642	2.32985
4	3.88172	5.2451	2.23882	2.73909	1.94742	2.52381	1.68663	2.22072
5	3.64762	5.39524	2.39226	2.62195	1.31854	2.42272	1.26835	2.5128
6	4.8172	5.39524	2.99429	2.84726	1.38764	2.31689	1.9552	2.22072
7	4.5625	5.3172	1.9142	2.57889	1.72435	3.24058	1.35205	2.32985
8	4.77083	5.39524	1.83557	3.10021	1.10844	2.40316	2.11174	2.22072
9	3.9697	5.39524	2.568	2.782	1.44422	2.27648	1.51686	2.5128
10	3.90972	5.39524	1.93804	2.89983	3.20124	2.40316	1.50222	2.32985

Table 7.12: Fitness Values of Resultant Decompositions by Both Approaches

This improvement in quality of results through ESBASCA is due to the absence of two inherent features of the GA based approach as mentioned in Section 1.4.

Reproduction can abolish good solutions in GAs, while ESs ensure that good solutions always survive into the next generation. The design of GA is such that parents do not survive in to the next generation and are replaced by the offspring, irrelevant of the fitness values. The result of such design is that the fitness value may suffer degradation if the offspring resulting from the cross over operator have less fitness than the parents. Hence, not only the convergence speed is affected but the solution may remain get stuck at local optima, if such situation continues to prevail through generations. A technique called Elitism [73] has been proposed that tries to minimize this loss over a number of generations.

This is not the case in ES where both parents and offspring compete to survive into the next generation and only the fittest survive; see details in Chapter 4. This means that fitness value can either remain unchanged or improve in ES.

To show this quality of ESBASCA, we have monitored and recorded the fitness values of each test system over 500 generations for both ESBASCA and GA based approach. From the results it is clear that the fitness value either increases or remains constant over the generations in case of ESBASCA. However, it may suffer degradation in case of GA based approach. This is shown in figures Figure 7.2 to 7.9.

Test System	GA-Based	ESBASCA	Percent Improvement
TS-1	4.323691	5.303035	~23%
TS-2	2.301682	2.820222	~22.5%
TS-3	1.719458	2.507144	~46%
TS-4	1.565705	2.347428	~50%

Table 7.13: Improvement in Quality through ESBASCA

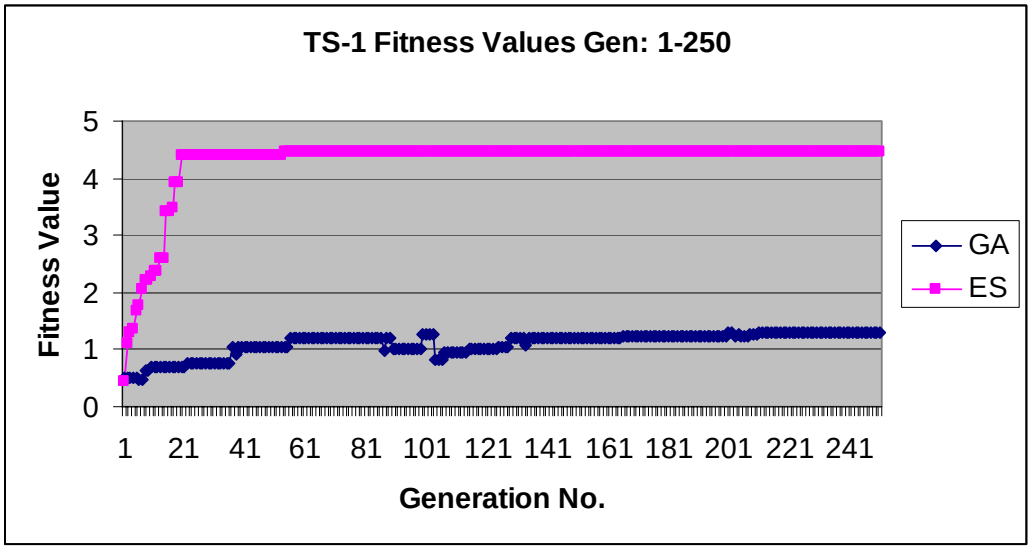

Figure 7.2: Fitness Values by Both Approaches of TS-1; generations 1-250

Self adaptation of strategy parameters is the second feature that resulted in improved results for ESBASCA. GA may remain stuck at local optima due to the fixed mutation rate throughout the evolution. Self adaptive ES, on the other hand, adapts the mutation rate along the course of evolution that helps in fine tuning the search. For this, mutation rate is also evolved by applying the mutation operator in the same way as it is applied to the individual solutions. The evolution process keeps monitoring whether

or not the change of mutation rate was advantageous according to it impact on the fitness of the individual solutions, and based on this information the mutation strength is modified.

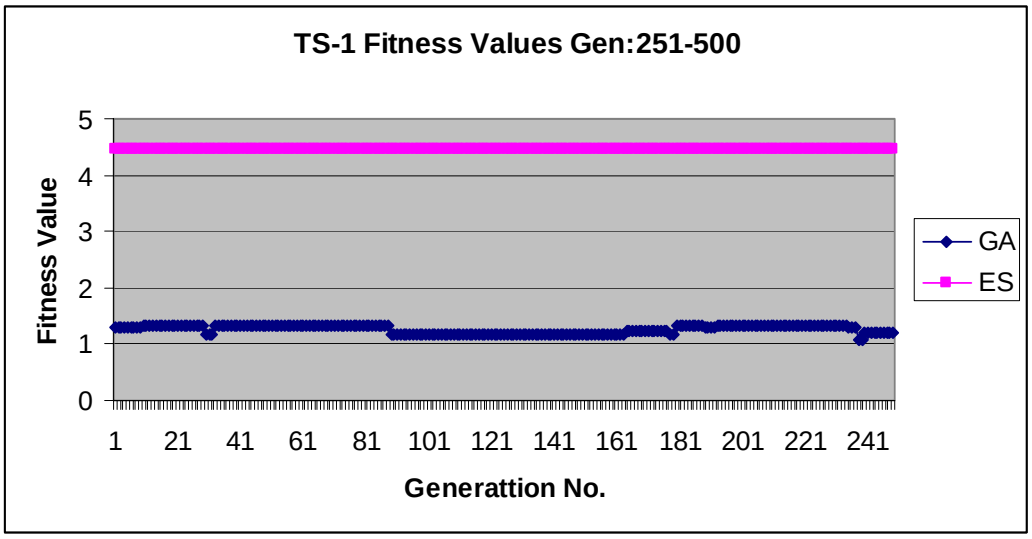

Figure 7.3: Fitness Values by Both Approaches of TS-1; generations 251-500

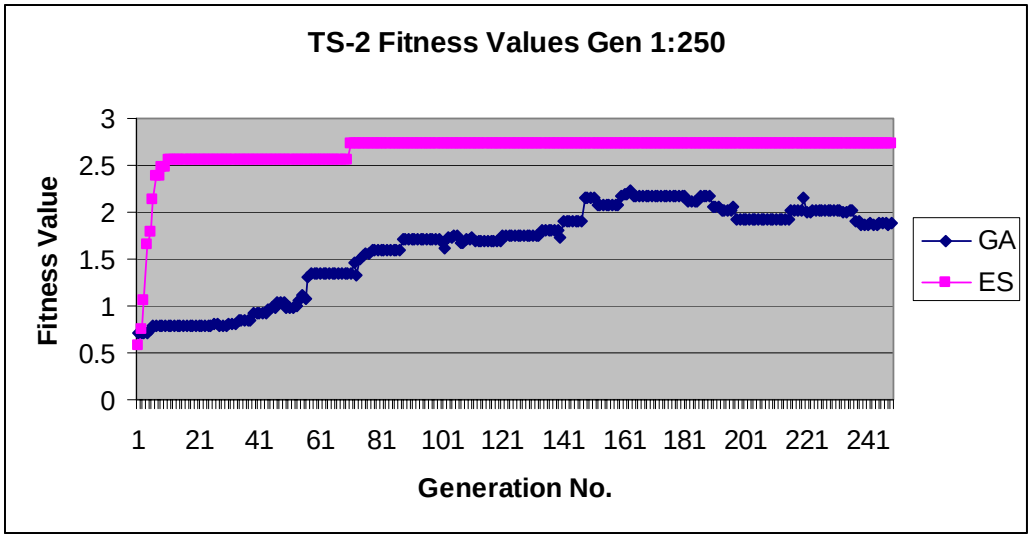

Figure 7.4: Fitness Values by Both Approaches of TS-2; generations 1-250

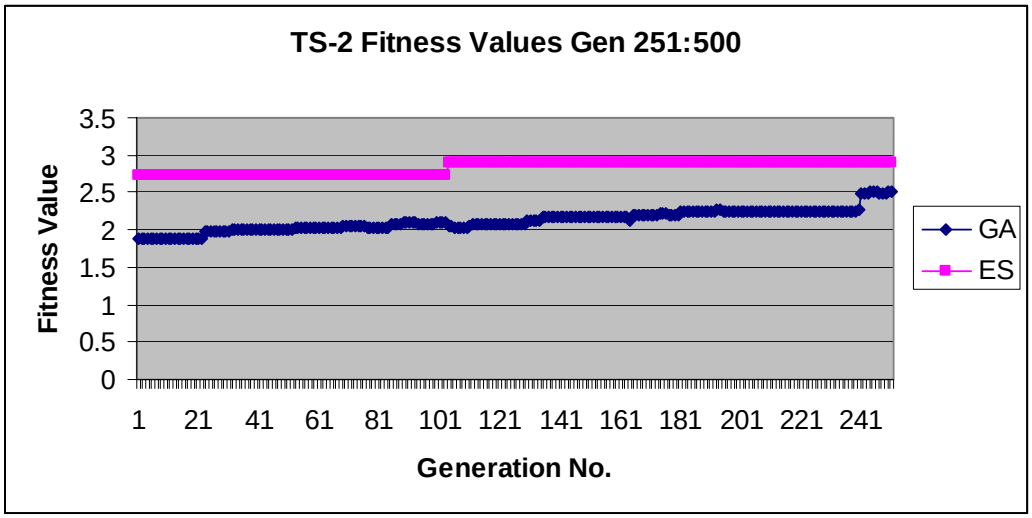

Figure 7.5: Fitness Values by Both Approaches of TS-2; generations 251-500

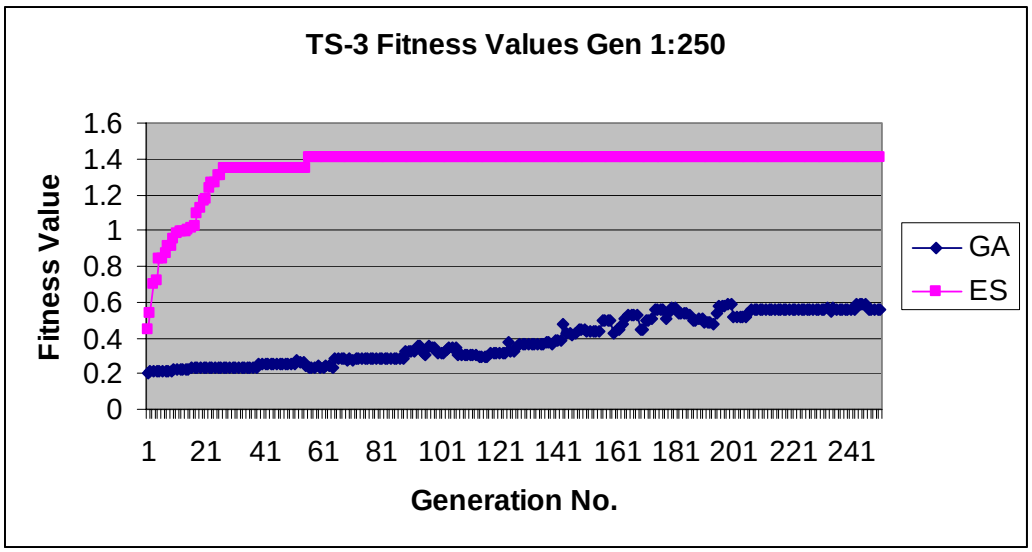

Figure 7.6: Fitness Values by Both Approaches of TS-3; generations 1-250

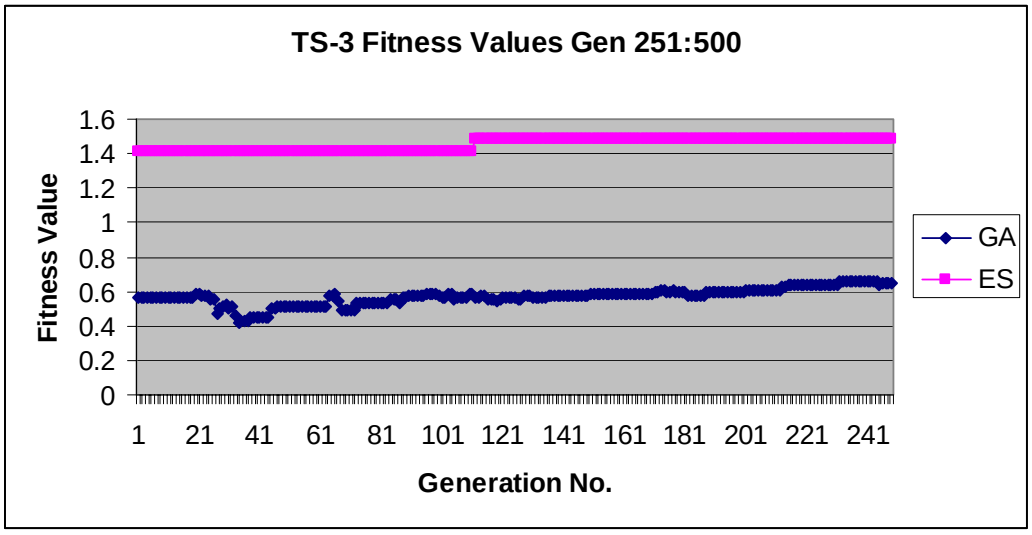

Figure 7.7: Fitness Values by Both Approaches of TS-3; generations 251-500

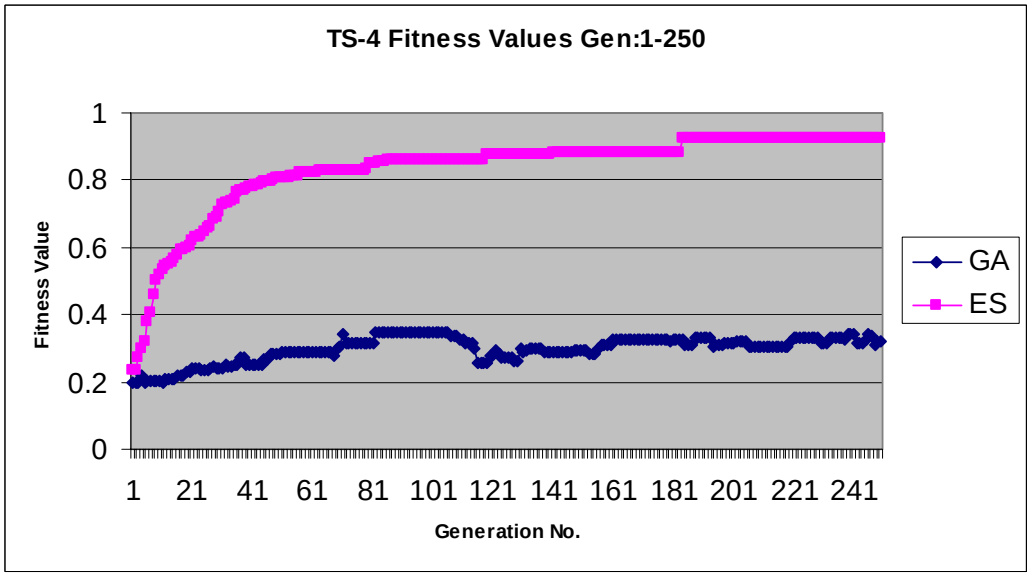

Figure 7.8: Fitness Values by Both Approaches of TS-4; generations 1-250

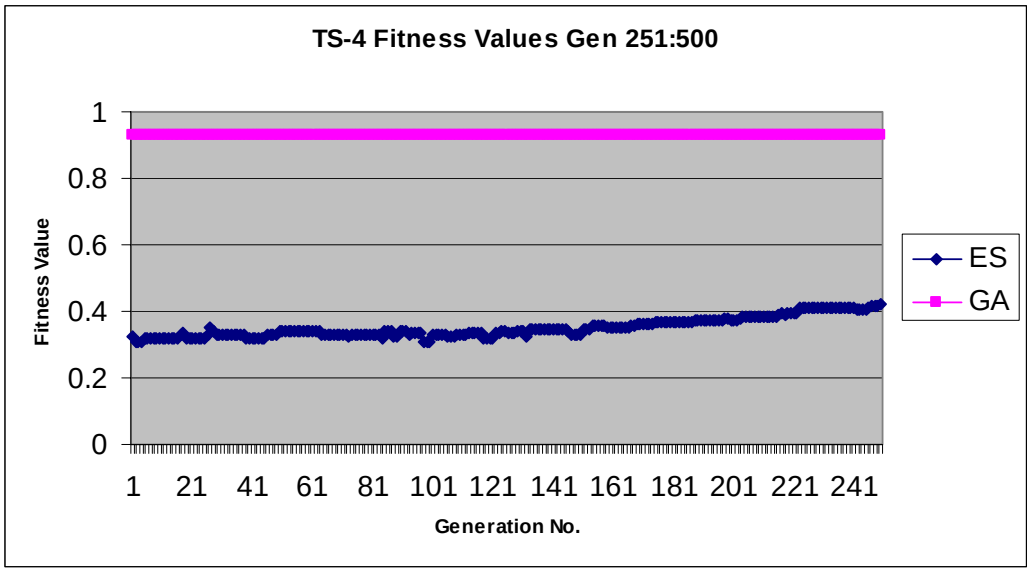

Figure 7.9: Fitness Values by Both Approaches of TS-4; generations 251-500

7.3.2 Effectiveness

Similarity Measure gives us the idea of how good (effective) a decomposition is, by comparing the decomposition produced by the clustering algorithm against the benchmark/expert decomposition. For obtaining the expert decompositions we approached the original designers of the test systems used in our study. Based on their knowledge of the system, source code, class listings and partial documentation of their corresponding systems, the designers provided us with the expert decompositions.

We have used the similarity measure *Precision and Recall* [30, 45]. Precision and Recall checks the correctness of our results on the basis of inter and intra cluster relations. Two entities in the same cluster are termed as *Intra pair* while two entities in different clusters are termed as *Inter pair*. *Precision* gives the percentage of intra

pairs proposed by the clustering algorithm which are also intra in the expert decomposition. *Recall* gives the percentage of intra pairs in the expert decomposition which were found by the clustering algorithm.

The higher these percentages are, better is the decomposition produced by the clustering algorithm. The *precision and recall* percentages of the decompositions produced by GA based approach and ESBASCA for ten runs for each test system is given in Table 7.14 and 7.15 respectively.

Figures 7.10 and 7.11 compare these resulting precision and recall percentages of the decompositions produced by GA based approach and ESBASCA averaged over ten runs for each test system. Again we can see that ESBASCA significantly outperforms GA based approach as it shows better precision and recall for all test systems. The percentage improvement in the precision and recall values by our approach as compared to GA based approach for each test system is provided in Table 7.16.

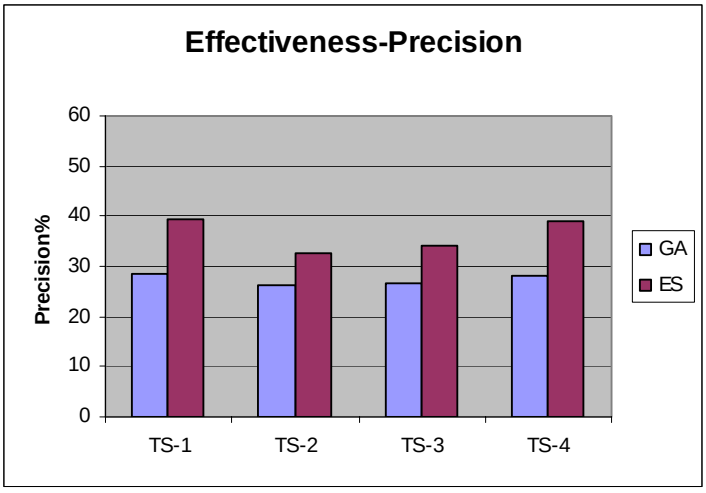

Figure 7.10: Comparison- Effectiveness (Precision)

Run#	TS-1		TS-2		TS-3		TS-4	
	GA Based	ESBASCA	GA Based	ESBASCA	GA Based	ESBASCA	GA Based	ESBASCA
1	25.93	41.57	23.7569	33.27	29.1457	34.57	29.7407	43.1871
2	23.12	47.24	26.9231	32.3	25.5486	34.4	29.3248	39.0635
3	30.8333	44.71	26.3158	31.95	24.5971	34.08	27.5511	35.2917
4	25	43.32	29.3839	32.3	26.2243	34.08	28.1946	37.8784
5	25.23	39.29	23.3202	31.38	24.2472	34.08	27.3734	36.4273
6	30.08	37.96	27.6498	32.84	29.4833	34.08	24.2174	36.8788
7	41.96	36.74	26.776	33.31	24.4842	34.08	27.4137	40.6693
8	31.43	28.66	27.8607	34.77	27.2727	34.08	26.8738	47.9478
9	25.4902	34.71	24.0964	32.11	26.676	34.08	25.9301	35.3961
10	26.21	39.85	25.5814	32.45	26.755	34.08	32.765	35.9522

Table 7.14: Precision % of Resultant Decompositions by Both Approaches

	Precision			Recall		
	GA Based	ESBASCA	Percent Improvement	GA Based	ESBASCA	Percent Improvement
TS-1	28.53	39.4	~38%	24.496	33.334	~36%
TS2-	26.17	32.67	~25%	26.96	42.646	~58%
TS-3	26.44	34.16	~29%	34.066	41.116	~21%
TS-4	27.94	38.87	~39%	42.208	57.136	~35%

Table 7.16: Improvement in Effectiveness through ESBASCA

Run#	TS-1		TS-2		TS-3		TS-4	
	GA Based	ESBASCA	GA Based	ESBASCA	GA Based	ES	GA Based	ESBASCA
1	21.71	52.71	21.08	39.71	34.38	38.58	49.26	54.78
2	17.05	37.21	27.45	42.16	35.82	38.22	34.61	56.26
3	28.68	35.66	24.51	43.63	35.82	47.96	61.57	57.54
4	22.48	35.66	30.39	37.25	35.1	43.63	35.24	57.54
5	20.93	27.91	28.92	32.84	40.26	43.02	32.48	57.54
6	31.01	27.91	29.41	41.18	29.2	39.54	41.19	57.54
7	36.43	28.68	24.02	36.27	29.6	40.62	37.79	57.54
8	25.58	27.91	27.45	59.8	27.4	40.5	46.5	57.54
9	20.16	30.23	29.41	49.02	29.21	37.98	40.55	57.54
10	20.93	29.46	26.96	44.6	43.87	41.11	42.89	57.54

Table 7.15: Recall % of Resultant Decomposition of Both Approaches

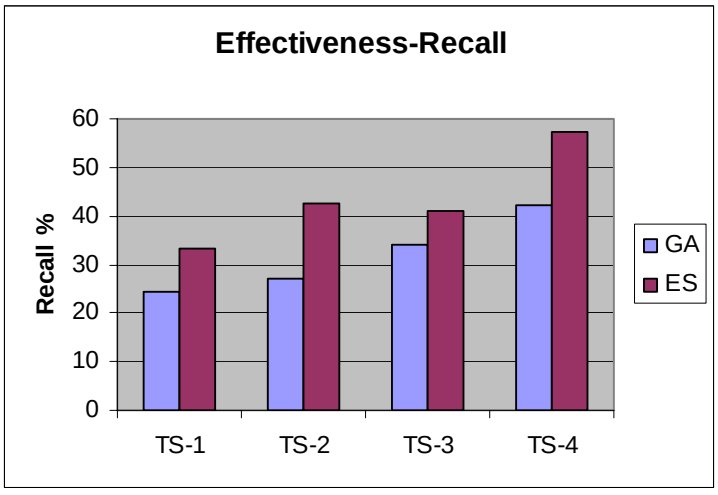

Figure 7.11: Comparison- Effectiveness (Recall)

7.3.3 Consistency

A comparison of the Fitness values of the best decomposition found by both GA based approach and ESBASCA for each test system in ten runs is presented in Figures 7.12 to 71.5. From these figures it is clear that

- ESBASCA yields much better results than the GA based approach. The improvement in fitness value by ES as compared to GA calculated for each test system over ten runs is given in Figure 7.1.
- ESBASCA produces much consistent decompositions as compared to the GA based approach.

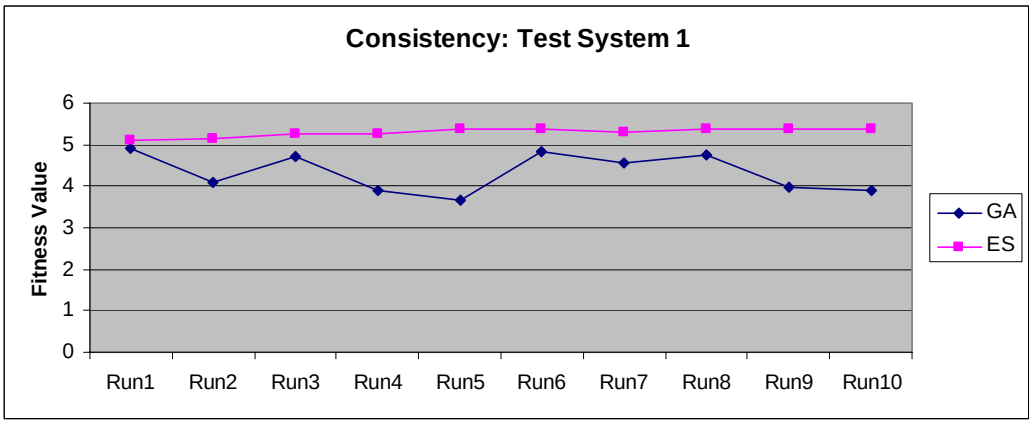

Figure 7.12: Consistency Comparison for TS-1

To highlight the second point mentioned above, standard deviation of the fitness values using both ESBASCA and GA based approach over 10 runs for each test system was computed. The results are shown in Figure 7.15. The figure shows that results with GA based approach have a standard deviation in the range of 0.29 to 0.59 for the four test systems while the results with ESBASCA have a standard deviation in the range of 0.11 to 0.28 for the same test systems. This means that even the

maximum deviation in ESBASCA's results is less than the minimum deviation of GA based results. This much less deviation by ESBASCA as compared to the GA based approach clearly indicates that our approach performs consistently without any major variations in results.

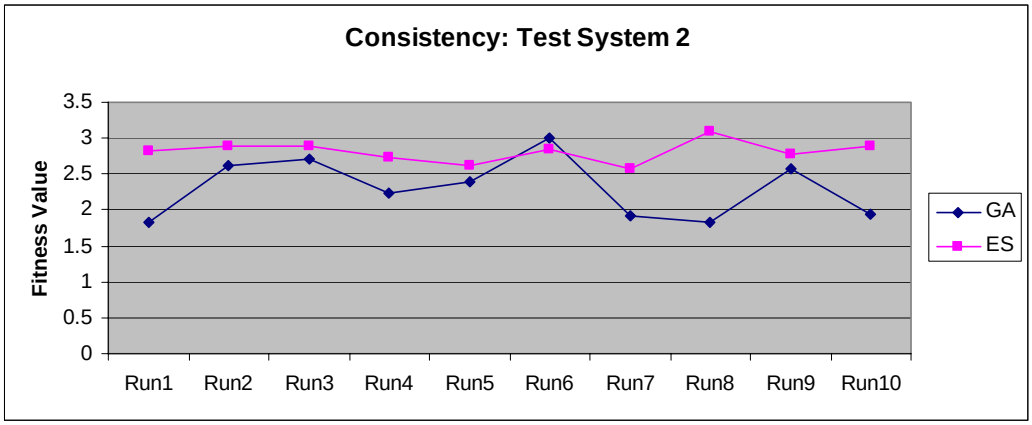

Figure 7.13: Consistency Comparison for TS-3

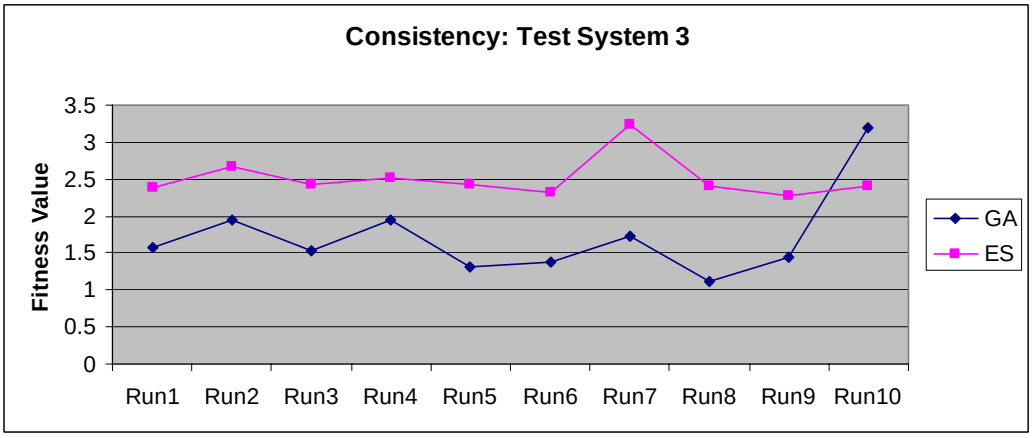

Figure 7.14: Consistency Comparison for TS-4

We would like to elaborate that the main reason for this difference in the consistency of ESBASCA and GA approaches can be attributed to the primary operators involved

in the two schemes. Mutation is the basic operator that provides genetic variation in ES. This operator helps in ensuring that the search is not stuck at local optima by adding variations in a manner that helps in exploring new possibilities in the search space without destroying the current high fitness values. Each individual has a probability of going through a small change when mutation is applied.

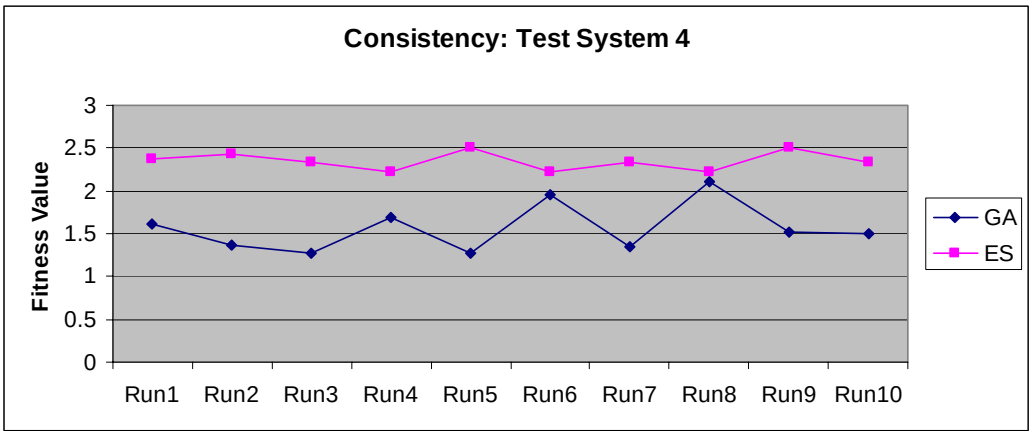

Figure 7.14: Consistency Comparison for TS-4

For example,

$$10\mathbf{0}01000 \rightarrow 10\mathbf{1}01000$$

The main operator in GA is the cross over operator. In contrast to mutation, cross over works with two individuals. Cross over operator combines parents (the individuals selected using the selection operator) to create offspring, in a bid to find individuals that have higher fitness values than either of the parents. In a single point cross over (the commonly used type of cross over) each individual is split at a point:

1 **<=** ***j*** **<=** ***L*** where L is the length of the individual.

By swapping the parts of the parents between j+1 and L, two new individuals are created:

$$100\ \mathbf{110} \rightarrow 100\ \mathbf{001}$$

$$001\ \mathbf{001} \rightarrow 001\ \mathbf{110}$$

Hence, the variation achieved by the cross over operator is higher than that of mutation. While this variation is the main driving force of GAs, it brings inconsistency for a non uniform population like the one in software clustering problem.

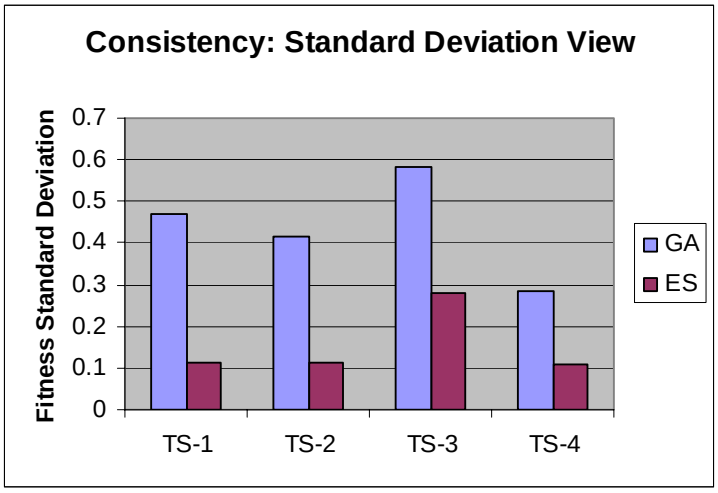

Figure 7.15: Comparison-Standard Deviation of Fitness Values

It is customary to present the decomposition visually, we also performed the visual comparison of the decompositions of TS-3 achieved by ESBASCA and GA based approaches with the actual decomposition of the system by original designer; using DOT [59]. It was clear from the visual comparison that ESBASCA based decomposition reduces the inter cluster edges to a small number, hence, achieving low

coupling and high cohesion which is a desirable quality. Due to resolution issues, it is not possible to present the visual comparison in this report.

7.3.4 Comparison with other Approaches

Comparison with other approaches is not required because of their obvious disadvantages as shown below. These techniques have been discussed in detail in Chapter 2 of this thesis report. Also, the implementation of these approaches is not publicly available and implementing them is beyond the scope of this thesis.

Approach	**Disadvantage**
Documentation Based	Updated documentation is not always available.
Data Bindings	Different level of granularity (clusters procedures into modules).
Semi Automated	Not feasible for large systems as they require too much user interaction.
Resource Name Based and Comprehension Driven	Too much subjective in nature

7.4 Validation of Test Results

The results were then shown to the original designers of the systems for validation. The designers of two test systems could spare time for this purpose. For this validation, architectures extracted through both techniques were given to different coders of same calibre who previously had no knowledge about these test systems. The coders were then asked to fix a problem in the code based on their understanding of the architecture. The coders acknowledged that the architecture extracted by ESBASCA was relatively more meaningful and it easily mapped to the source code. Here it must be made clear that IDs were assigned to the architectures and it was not known to the persons validating the results that which architecture was obtained using what technique.

Chapter Eight: Conclusion and Future Research Directions

8.1 Conclusion

Maintaining and understanding large software systems from source code or module dependency graph is a difficult task. Partitioning the graph can help but the number of possible partitions is quite large even for small systems. We have presented a self adaptive Evolution Strategies based approach that explores this large solution space to find an effective decomposition of the system. To study the effectiveness of our proposed approach, we have compared it against GA based approach using industrial systems of different sizes. The encouraging results showing the quality and effectiveness of our approach are presented for a number of test systems. In addition, the standard deviation among the achieved results by ESBASCA is much less than that of the GA based approach, highlighting the consistency in results of our approach. The encouraging consistent results make our approach more stable as well.

8.2 Future Directions

- In future we want to establish the stability of our approach using the stability measure defined in [48].
- We also want to develop a new similarity measure to remove a flaw in EdgeSim [74]. EdgeSim gives same results for two decompositions if all edges in both decompositions are of same type. It is possible that a module moves from one cluster to another cluster in a manner that edge types remain the same. EdgeSim will not point out this difference. Our similarity measure will incorporate this movement of modules between clusters.

- We have plans to conduct comparative studies with other clustering approaches as well.
- Another task that we have on the list is to explore other clustering algorithms using larges systems of magnitude as that of those used in our study.

8.3 Research Contributions

The following contributions have been made during our research.

8.3.1 A New Approach for Automated Software Clustering

We have proposed a new approach for automated software clustering. Our approach is based on using self adaptive evolution strategies. We have successfully verified our approach on sufficiently large and complex industrial systems. The approach yielded encouraging results.

8.3.2 Comparative Study of Software Clustering Approaches

We carried out a comparative study of our proposed approach and the widely used genetic algorithms based software clustering approach. Our study was based on comparison of the resultant decompositions of the two approaches in terms of fitness values (quality), precision & recall measure, (effectiveness) and consistency in results. In future we want to conduct comparative studies with other clustering approaches as well.

8.3.3 Consistency of Software Clustering Approaches

We were hardly able to find the research works in Software Clustering literature that has formally compared the consistency of the results generated by different software

clustering approaches. In this manner, our effort is one of the very first ones in this domain that compares consistency in results, an important and desirable property of any algorithm.

8.3.4 Empirical Study on Industrial Systems

We conducted our study on large and complex software systems. Lately, the researchers have been pointing towards the need to test software clustering techniques on large and complex systems. But there has been hardly any work in this regard. For example, B.S. Mitchell [21] presented his work on Genetic Algorithms using just one test system and that too consisting of 20 modules. We could only find the work of Jingwei Wu, Ahmed E. Hassan, Richard C. Holt [75] who conducted their study on sufficiently large open source systems. We have conducted our study on large industrial systems. In future we want to check other clustering algorithms on systems of this magnitude.

REFERENCES

[1] Buss, E., De Mori, R., Gentleman, W. et al., “Investigating Reverse Engineering Technologies for the CAS Program Understanding Project”, IBM Systems Journal, vol. 33, no. 3, pp. 477-500, February, 1994.

[2] Nosek, J.T. and Palvia, P., “Software Maintenance Management: Changes in the Last Decade”, Journal of Software Maintenance, 2(3), pp. 157-174, Sept, 1990.

[3] McCabe, T., “Keynote address at the Working Conference on Reverse Engineering”, Hawaii, October, 1998.

[4] Fjeldstadt, R.K., and Hamlen, W.T., “Application Program Maintenance Study: Report to Our Respondents”, Proc. GUIDE 48, IEEE Computer Society Press, April, 1984.

[5] Garlan, D., Perry D.E., “Introduction to the Special Issue on Software Architecture”, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 269-274, April, 1995

[6] Bayer, J., Girard, J.-F., Würthner, M, Apel, M., and DeBaud, J.-M., “Transitioning Legacy Assets - a Product Line Approach”, Proceedings of the SIGSOFT Foundations of Software Engineering, Toulouse, pp. 446-463, Association of Computing Machinery, 1999.

[7] Schwanke, R. W., “An intelligent tool for Re-engineering Software Modularity”, International Conference on Software Engineering, pp. 83–92, May, 1991.

[8] Fiutem, R., Tonella, P., Antoniol, G., and Merlo, E. (1996), “A Cliché-based Environment to Support Architectural Reverse Engineering”, pp. 319-328, Proc. of the Int. Conf. on Software Maintenance, 1996.

[9] Harris, D.R., Reubenstein, H.B., and Yeh, A.S, “Recognizers for Extracting Architectural Features from Source Code”, Proceedings of the Working Conference on Reverse Engineering, pp. 227-236, Toronto, IEEE Computer Society Press, 1995.

[10] M. R. Anderberg, “Cluster Analysis for Applications”, Academic Press Inc., 1973

[11] Brian S. Everitt, “Cluster Analysis”, John Wiley & Sons, 1993.

[12] John A. Hartigan, “Clustering Algorithms”, John Wiley & Sons, 1975.

[13]. A. Jain and R. Dubes, “Algorithms for Clustering Data”, Prentice-Hall, 1998.

[14] J. Zupan, “Clustering of Large Data Sets”, Research Studies Press, England, 1982.

[15] Robert W. Schwanke and Michael A. Platoff. “Cross References are Features”, In Second International Workshop on Software Configuration Management, pages 86-95, ACM Press, 1989.

[16] Nicolas Anquetil and Timothy Lethbridge, “File Clustering Using Naming Conventions for Legacy Systems”, In Proceedings of CASCON 1997, pages 184-195, November, 1997.

[17] David H. Hutchens and Victor R. Basili, “System Structure Analysis: Clustering with Data Bindings”, IEEE Transactions on Software Engineering, 11(8):749-757, August, 1985.

[18] Jams M. Neighbors, “Finding Reusable Software Components in Large Systems”, In Proceedings of the Third Working Conference on Reverse Engineering, pages 2-10, IEEE Computer Society Press, November, 1996.

[19] Song C. Choi and Walt Scacchi, “Extracting and Restructuring the Design of Large Systems”, IEEE Software, pages 66-71, January, 1990.

[20] S.~Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using Automatic Clustering to Produce High-Level System Organizations of Source Code", In Proceedings of the International Workshop on Program Understanding, 1998.

[21] B. S. Mitchell, "A Heuristic Search Approach to Solving the Software Clustering Problem", PhD Thesis, Drexel University, Philadelphia, PA, Jan. 2002.

[22] Hans-George Beyer, Hans-Paul Schwefel, "Evolution Strategies -A Comprehensive Introduction", Natural Computing: An International Journal, Vol 1 No 1, pages 3-52, May 2002.

[23] Hans-George Beyer, "The Theory of Evolution Strategies", Springer, April 27, 2001.

[24] D. E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, New York, 1989.

[25] M. Mitchell, "An Introduction to Genetic Algorithms", The MIT Press, Cambridge, Massachusetts, 1997.

[26] D. Parnas, “On the Criteria to be used in Decomposing Systems into Modules “Communications of the ACM, pages 1053-1058, 1972.

[27] G. Booch, “Object Oriented Analysis and Design with Applications”, The Benjamin Cummings Publishing Company Incorporated, 2nd edition, 1994.

[28] R. Schwanke and S. Hanson, “Using Neural Networks to Modularize Software”, Machine Learning, 15:137-168, 1998.

[29] H. Muller, M. Orgun, S. Tilley, and J. Uhl., “Discovering and Reconstructing Subsystem structures through reverse engineering”. Technical Report DCS-201-IR, Department of Computer Science, University of Victoria, August, 1992.

[30] N. Anquetil and T. Lethbridge, "Recovering Software Architecture from the Names of Source files", In Proceedings of Working Conference on Reverse Engineering, October, 1999.

[31] K. Narayanaswamy and W. Scacchi, “Maintaining Configurations of Evolving Software Systems”. IEEE Transactions on Software Engineering, SE-13(3):324-334, March, 1987.

[32] S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner, “Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In Proceedings of International Conference of Software Maintenance, pages 50-59, August, 1999.

[33] B. S. Mitchell, M. Traverso, and S. Mancoridis, “An Architecture for Distributing the Computation of Software Clustering Algorithms”. In The Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), August, 2001.

[34] D. Doval, S. Mancoridis, and B.S. Mitchell, “Automatic Clustering of Software Systems using a genetic algorithm”, In Proceedings of Software Technology and Engineering Practice, August, 1999.

[35] V. Tzerpos and R.C. Holt, “The Orphan Adoption Problem in Architecture Maintenance”, In Proc. Working Conf. on Reverse Engineering, October, 1997.

[36] C. Lindig and G. Snelting, “Assessing Modular Structure of Legacy Code based on Mathematical Concept Analysis”, In Proc. International Conference on Software Engineering, May, 1997.

[37] G. Snelting, “Concept Analysis: A New Framework for Program Understanding”, In Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE98), volume ACM SIGPLAN Notices33, pages 1-10, June, 1998.

[38] A. van Deursen and T. Kuipers, “Identifying objects using cluster and concept analysis”, In International Conference on Software Engineering, ICSM'99, pages 246-255. IEEE Computer Society, May, 1999.

[39] L. Kaufman and P.J. Rousseeuw, “Finding Groups in Data: An Introduction to Cluster Analysis”, John Wiley & Sons, 1990.

[40] C. Montes de Oca and D. Carver, ”A visual representation model for software subsystem decomposition”, In Proc. Working Conf. on Reverse Engineering, October, 1998.

[41] K. Sartipi, K. Kontogiannis, and F. Mavaddat, “Architectural Design Recovery using Data Mining Techniques”, In Proceedings of the IEEE European Conference on Software Maintenance and Reengineering (CSMR 2000), pages 129-139, March, 2000.

[42] K. Sartipi and K. Kontogiannis, “Component Clustering Based on Maximal Association”, In Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE 2001), pages 103-114, October 2001.

[43] G. Murphy, D. Notkin, and K. Sullivan, “Software Reexion Models: Bridging the Gap between Design and Implementation”, IEEE Transactions on Software Engineering, pages 364-380, April, 2001.

[44] T. Eisenbarth, R. Koschke, and D. Simon, “Aiding Program Comprehension by Static and Dynamic Feature Analysis”, In Proceedings of the IEEE International Conference of Software Maintenance (ICSM 2001), November, 2001.

[45] N. Anquetil, C. Fourrier, and T. Lethbridge, "Experiments with hierarchical clustering algorithms as software modularization methods", In Proceedings of the Working Conference on Reverse Engineering, 1999.

[46] Vassilios Tzerpos and R. C. Holt, "ACDC: An Algorithm for Comprehension-Driven Clustering", In Proceedings of WCRE 2000, Brisbane, Australia, November 2000.

[47] R. Koschke, “Evaluation of Automatic Re-Modularization Techniques and their Integration in a Semi-Automatic Method”, PhD thesis, University of Stuttgart, Stuttgart, Germany, 2000.

[48] Vassilios Tzerpos and R.C. Holt, "On the Stability of Software Clustering Algorithms", Proceedings of the 8th International Workshop on Program Comprehension, Limerick, Ireland, June 2000.

[49] Hope A. Olson and Dietmar Wolfram, "Indexing Consistency and its Implications for Information Architecture: A pilot Study", IA Summit 2006.

[50] Ramón Asensio Monge, Francisco Sanchis Marco, Fernando Torre Cervigón, "An Assessment of the Consistency for Software Measurement Methods", ArXiv Computer Science e-prints, cs/0204014, April 2002.

[51] T.A.Wiggerts, “Using Clustering Algorithms in Legacy Systems Remodularization”, In Proc. Working Conference on Reverse Engineering, October, 1997.

[52] D. G. Corneil and M. E. Woodward, “A Comparison and Evaluation of Graph Theoretical Clustering Techniques”, INFOR, 16, 1978.

[53] P. K. T. Vaswani, “A Technique for Cluster Emphasis and Its Applications to Automatic Indexing”, Information Processing, 68(2):1300-1303, 1968.

[54] Gregor von Laszewski, “A Collection of Graph Partitioning Algorithms”, Technical Report SCCS 477, Northeast Parallel Architecture Center at Syracuse University, May, 1993.

[55] Rodrigo A. Botafogo and Ben Schneiderman, “Identifying Aggregates in Hypertext Structures”, In Proceedings of Hypertext 91, page 63-74, 1991.

[56] B.W. Kernighan and S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs”, Bell Systems Technical Journal, 49:291-307, 1970.

[57] V. Tzerpos and R. C. Holt, “MoJo: A Distance Metric for Software Clustering”, In Proceedings of Working Conference on Reverse Engineering, October 1999.

[58] Abbasi, A. Q, "Application of Appropriate Machine Learning Techniques for Automatic Modularization of Software Systems", M-Phil Thesis, Quaid-i-Azam University, Islamabad, 2008.

[59] S. North and E. Koutsofios, “Applications of Graph Visualization”, In Proc. Graphics Interface, 1994.

[60] Schwefel H-P, “Collective Phenomena in Evolutionary Systems”, Int’l Society for General System Research, Vol. 2. Budapest, pp. 1025–1033, 1987.

[61] Herdy M., “Reproductive Isolation as Strategy Parameter in Hierarchically Organized Evolution Strategies”, In: Männer R and Manderick B (eds) Parallel Problem Solving from Nature, 2, pp. 207–217. Elsevier, Amsterdam, 1992.

[62] Hans-George Beyer, “Some Aspects of the Evolution Strategy’ for Solving Tsp-like Optimization Problems”, In: Männer R and Manderick B (eds) Parallel Problem Solving from Nature, 2, pp. 361–370. Elsevier, Amsterdam, 1992.

[63] Hans-George Beyer, "The Theory of Evolution Strategies", Springer, April 27, 2001.

[64] Rudolph G., “An Evolutionary Algorithm for Integer Programming”, In: Davidor Y, Männer R and Schwefel H-P (eds) Parallel Problem Solving from Nature, 3, pp. 139–148, Springer-Verlag, Heidelberg, 1994.

[65] Altenberg L., “The Evolution of Eolvability in Genetic Programming”, In: Kinnear K (ed) Advances in Genetic Programming, pp. 47–74. MIT Press, Cambridge, MA, 1994.

[66] Rechenberg I., “Evolutionsstrategie ’94”, Frommann-Holzboog Verlag, Stuttgart.

[67] Schwefel H-P, “Evolutionsstrategie und numerische Optimierung”, Dissertation, TU Berlin, Germany, 1975.

[68] Rechenberg I., “Evolutionsstrategien”, In: Schneider B and Ranft U (eds) Simulationsmethoden in der Medizin und Biologie, pp. 83–114. Springer-Verlag, Berlin, 1978.

[69] Schwefel H-P, “Adaptive Mechanismen in der biologischen Evolution und ihr Einfluß aufdie Evolutionsgeschwindigkeit” Technical report, Technical University of Berlin, 1974.

[70] Schwefel H-P, “Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie”, Interdisciplinary systems research; 26. Birkh¨auser, Basel, 1977.

[71] Silja Meyer-Nieberg, "Self-Adaptation in Evolution Strategies", PhD Thesis, University of Dortmund, Dortmund, 2007.

[72] Y. Chen, E. Gansner, and E. Koutsofios, “A C++ Data Model Supporting Reachability Analysis and Dead Code Detection”, In Proc. 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September, 1997.

[73] B. CHAKRABORTY and P. CHAUDHURI, "On the Use of Genetic Algorithm with Elitism in Robust and Nonparametric Multivariate Analysis", Austrian Journal of Statistics, Vol 32, No 1 and 2, 2003.

[74] B. Mitchell, S. Mancoridis, "Comparing the decompositions produced by software clustering algorithms using similarity measurements", In Proceedings of the 17th International Conference on Software Maintenance, pages 744-753, Florence, Italy, November, 2001.

[75] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt, “Comparison of Clustering Algorithms in the Context of Software Evolution”, In Proceedings of ICSM 2005: International Conference on Software Maintenance, Budapest, Hungary, Sept 25-30, 2005.

Appendix A: Expert Decompositions for the Test Systems Used in Our Study

An effective way of evaluating the software clustering results is to compare the decomposition generated by the clustering technique against expert/reference/benchmark decomposition. In order to obtain the expert decompositions for the systems used in our study, we approached the original designers of the test systems used in our study. Based on their knowledge of the system, source code, class listings and partial documentation of their corresponding systems, the designers provided us with the expert decompositions. As discussed in Chapter 7, the decompositions are compared using some *similarity measure*. In our study, we used the *precision and recall* similarity measure to compare the clustering results against the expert decompositions.

For the interested audience, we are presenting the expert decompositions of the test systems used in our study, in this Appendix.

A.1 Expert Decomposition for Test System 1

No. of Clusters: 6 **No. of Modules:** 36

Cluster /Subsystem ID	Contained Modules	No. Of Modules Present
0	8, 20	2
1	4, 26, 28, 34	4
2	15, 22, 23, 27	4
3	3, 6, 7, 12, 24, 29, 33, 35	8
4	0, 1, 2, 5, 16	5
5	9, 10, 11, 13, 14, 17, 18, 19, 21, 25, 30, 31, 32	13

A.2 Expert Decomposition for Test System 2

No. of Clusters: 4 **No. of Modules:** 41

Cluster /Subsystem ID	Contained Modules	No. Of Modules Present
0	0, 1, 7, 8, 9, 10, 11, 12, 13, 14, 15, 19, 40	13
1	2, 3, 4, 5, 24, 31, 32, 33, 34, 35, 38, 23	12
2	16, 21, 22, 25, 26, 27, 28, 29, 37	9
3	6, 17, 18, 20, 30, 36, 39	7

A.3 Expert Decomposition for Test System 3

No. of Clusters: 3 **No. of Modules:** 69

Cluster /Subsystem ID	Contained Modules	No. Of Modules Present
0	0, 4, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 29, 33, 34, 35, 36, 37, 45, 46, 47, 48, 49, 50, 52, 53, 55, 56, 57	31
1	1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 30, 31, 32, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68	24
2	12, 13, 14, 15, 28, 38, 39, 40, 41, 42, 43, 44, 51, 54	14

A.4 Expert Decomposition for Test System 4

No. of Clusters: 8 **No. of Modules:** 80

Cluster /Subsystem ID	Contained Modules	No. Of Modules Present
0	0, 1, 3, 12, 15, 25, 27, 47, 57 , 58, 60, 63, 64, 65, 67, 68, 70, 71	18
1	5, 10, 11, 16	4
2	2, 18, 20, 29, 30, 33, 48, 49, 55, 61	10
3	7, 8, 9, 17, 19, 21, 23, 28, 35	9
5	4, 6, 13, 14, 22, 24, 26, 34, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46	19
5	31, 32, 62,66	3
6	50, 51, 52, 53, 54, 56, 59, 69, 72	9
7	73, 74, 75, 76, 77, 78, 79	7