Automatic Clustering of Software Systems using a Genetic Algorithm

D. Doval, S. Mancoridis, B. S. Mitchell

Dept. of Mathematics and Computer Science
Drexel University
Philadelphia, PA 19104
e-mail: {uddoval, smancori, bmitchel}@mcs.drexel.edu

Abstract

Large software systems tend to have a rich and complex structure. Designers typically depict the structure of software systems as one or more directed graphs. For example, a directed graph can be used to describe the modules (or classes) of a system and their static inter-relationships using nodes and directed edges, respectively. We call such graphs module dependency graphs (MDGs).

*MDGs can be large and complex graphs. One way of making them more accessible is to partition them, separating their nodes (*i.e., *modules) into clusters (*i.e., *subsystems). In this paper, we describe a technique for finding "good" MDG partitions. Good partitions feature relatively independent subsystems that contain modules which are highly inter-dependent. Our technique treats finding a good partition as an optimization problem, and uses a Genetic Algorithm (GA) to search the extraordinarily large solution space of all possible MDG partitions. The effectiveness of our technique is demonstrated by applying it to a medium-sized software system.*

KEYWORDS
Automatic Clustering, Reverse Engineering, Genetic Algorithms.

1. Introduction

The maintenance and evolution of medium and large software systems can be a daunting task, especially if the system is poorly documented or not documented at all. Continuous maintenance over an extended period of time can have a negative impact on the quality of a system's structure (also referred to as Software Architecture [12, 14]).

Software designers use directed graphs to make the structure of complex software systems more understandable. We call such graphs module dependency graphs (MDGs). In MDGs, the system's modules (*e.g.*, files, classes) are represented as nodes and their relationships (*e.g.*, function calls, inheritance relationships) as directed edges that connect the nodes. However, even the MDGs of small systems can be complex. One way of making MDGs more accessible is to partition them by grouping closely related nodes into clusters, in what is known as the *software clustering problem*. The MDG of a software system can be recovered automatically from its source code using tools such as CIA [6] (for C), Acacia [1] (for C and C++) or Star [7] (for Turing [4]).

Creating a *meaningful* partition of an MDG, however, is difficult, because the number of possible partitions is very large even for a small graph [8]. Also, small differences between two partitions can yield very different results. As an example, let's consider Figure 1, which presents an MDG with a small number of modules and relations. (All diagrams on this paper were produced using *Dot* [6], a tool for drawing and automatically laying out labeled directed graphs.)

The two partitions of the MDG presented in Figures 2 and 3 are very similar, with only two nodes swapped. In spite of this seemingly small difference, the partition defined in Figure 3 better captures the high-level structure of the graph, since it groups nodes that are more interdependent.

In this paper, we describe a technique that automatically finds a good partition of a system's MDG. Our approach treats clustering as an optimization problem, where the goal is to find a good (possibly optimal) partition. To explore the extraordinarily large solution space of all possible partitions for a given MDG, we use a Genetic Algorithm (GA) [2, 9]. In Section 3 we describe how we quantify a partition's

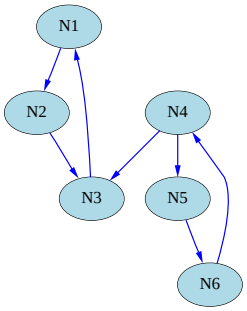

Figure 1. An example of a simple MDG

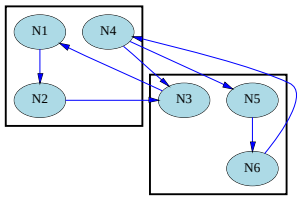

Figure 2. A partition of the MDG

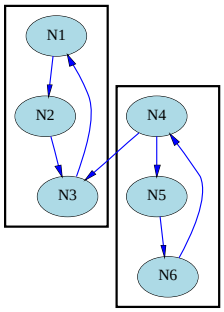

Figure 3. A better partition of the MDG

quality using features of the graph.

We demonstrate the effectiveness of our technique on a medium-sized software system. The tool that implements our technique uses the system's MDG as input, and produces a partition of that MDG as output.

The structure of the remainder of this paper is as follows: in Section 2, we outline the principles of genetic algorithms, in Section 3 we explain our GA that performs automatic clustering. In Section 4, we describe the results of applying our technique to a medium sized software system. In Section 5, we review related work on software clustering. We conclude the paper in Section 6, outlining some of the future directions of our work.

2. Genetic algorithms

Genetic algorithms apply ideas from the theory of natural selection to navigate through large search spaces efficiently. GAs have been found to overcome some of the problems of traditional search methods such as hill-climbing [2, 9]; the most notable problem being "getting stuck" at local optimum points, and therefore missing the global optimum (best solution). GAs perform surprisingly well in highly constrained problems, where the number of "good" solutions is very small relative to the size of the search space.

GAs operate on a set (*population*) of strings (*individuals*), where each string is an encoding of the problem's input data. Each string's *fitness* (quality value) is calculated using an objective function. Probabilistic rules, rather than deterministic ones, are used to direct the search. In GA terminology, each iteration of the search is called a *generation*. In each generation, a new population is created by taking advantage of the fittest individuals of the previous generation.

GAs leverage the idea that sub-string patterns shared by high-fitness individuals can be combined to yield higher-fitness solutions. Further explanations and details on why GAs work can be found elsewhere [2, 9].

Genetic algorithms are characterized by attributes such as objective function, encoding of the input data, genetic operators, and population size. After describing these attributes, we describe the actual algorithm used by GAs in Section 2.5.

2.1. Objective function

The objective function is used to assign a fitness value to each individual in the population. Therefore, it needs to be designed so that an individual with a high fitness represents a better solution to the problem than an individual with a lower fitness.

2.2. Encoding

GAs operate on an *encoding* of the problem's input data. The choice of the encoding is extremely important for the execution performance of the algorithm. A poor encoding can lead to long-running searches that do not produce good results.

An encoding is expressed using a finite *alphabet*. Typical alphabets are binary $(e.g., T, F)$ and numeric $(e.g., 0, 1, 2, \ldots, n)$. The latter is used throughout this discussion.

2.3. Genetic operators

GAs feature the following three basic operators, which are executed sequentially by the GA:

1. Selection and Reproduction
2. Crossover
3. Mutation

Crossover and mutation have a fixed rate of occurrence (*i.e.*, the operators are applied with a certain fixed probability) that varies from problem to problem. A detailed explanation on why these rates vary across problems and how to determine them can be found elsewhere [2].

During **selection and reproduction**, pairs of individuals are chosen from the population according to their fitness.

The reproduction operator can be implemented in a number of ways [2]. For our purposes, we consider *roulette wheel* selection. This type of selection simulates a "spinning wheel" to randomly determine which individuals are selected from the old population to be included in the new population. In roulette wheel selection, each string is assigned a "slot" of size proportional to its fitness in the reproduction "wheel", so individuals with a higher fitness are more likely to be selected.

Selection can be complemented with *elitism*. Elitism guarantees that the fittest individual of the current population is copied to the new population.

Crossover is performed immediately after selection and reproduction. The crossover operator is used to combine the pairs of selected strings (parents) to create new strings that potentially have a higher fitness than either of their parents. During crossover, each pair of strings is split at an integer position k ($1 \leq k \leq l$), using a uniform random selection from position 1 to $l - 1$, where l is the length of the string. Two new strings are then created by swapping characters between positions $k + 1$ and l (inclusively) of the selected individuals. Thus, two strings are used to create two new strings, maintaining the total population of a generation constant.

The procedure of selection and reproduction combined with crossover is simple. However, the emphasis of reproduction on high-fitness strings and the structured (albeit randomized) information exchange of crossover is at the core of the genetic algorithm's power to discover good solutions.

The **mutation** operator is applied to every string resulting from the crossover process. When mutation is applied, each character of the string has a low probability (*e.g.*, typically 4/1000) of being changed to another random value of the same type and range.

Selection, reproduction and crossover can be very effective in finding strings with a high fitness value. The mutation operator is crucial to avoid missing high-fitness strings when the current population has converged to a local optimum. However, mutation is used sparingly to explore new regions of the search space and open up new possibilities without destroying current high-fitness strings.

2.4. Population size

The number of strings in a population is defined by the *population size*. The larger the population size, the better the chance that an optimal solution will be found. Since GAs are very computationally intensive, a trade-off must be made between population size and execution performance.

2.5. The algorithm

GAs use the operators defined above to operate on the population through an iterative process, which is as follows:

1. Generate the initial population, creating random strings of fixed size.
2. Create a new population by applying the selection and reproduction operator to select pairs of strings. The number of pairs will be the population size divided by two, so the population size will remain constant between generations.
3. Apply the crossover operator to the pairs of strings of the new population.
4. Apply the mutation operator to each string in the new population.
5. Replace the old population with the newly created population.
6. If the number of iterations is less than the maximum, go to step 2. Else, stop the process and display the best answer found.

As defined in step 6, the GA iterates a fixed number of times. Since the function's upper bound (the maximum fitness value possible for a string) is often not found, we must

limit the number of generations in order to guarantee the termination of the search process. This may result in a sub-optimal solution.

In the next section, we explain how our Genetic Algorithm solves the *automatic software clustering* problem.

3. Software clustering GA

In the introduction we saw that the structure of a software system can be expressed as a module dependency graph. The goal of our automatic clustering GA is to find a "good" partition of the MDG, so that closely related modules (nodes) are grouped into subsystems (clusters).

3.1. The objective function

We use an objective function to quantify the quality of an MDG partition [8]. Our conjecture is that a high quality partition is one that has both a low number of inter-cluster relationships and a high number of intra-cluster relationships. This conjecture is based on the assumption that well-designed systems are formed by cohesive sets of modules that are loosely related between each other.

Before we can quantify the quality of a partition, we must first quantify intra- and inter-connectivity.

Intra-connectivity is a measure of the density of connections between the nodes of a single cluster. A high intra-connectivity indicates a good clustering arrangement because the nodes grouped within the same cluster are highly dependent on each other. A low intra-connectivity, on the other hand, generally indicates a poor clustering arrangement because the nodes grouped within a cluster are not strongly related.

We define the intra-connectivity A_i of cluster i with N_i components and μ_i intra-edge dependencies (relationships to and from modules within the same cluster) as:

$$A_i = \frac{\mu_i}{N_i^2}$$

A_i is the number of intra-edge dependencies divided by the maximum number of possible dependencies between the components of cluster i, which is N_i^2. A_i is 0 when the modules in a cluster are not connected, and 1 when each module in a cluster is connected to every module in the same cluster (including itself).

Inter-connectivity is a measure of the connectivity between distinct clusters. A high degree of inter-connectivity is undesirable because it indicates that clusters are highly dependent on each other. Conversely, a low degree of inter-connectivity is desirable because it indicates that individual clusters are largely independent of each other.

We define the inter-connectivity $E_{i,j}$ between clusters i and j, each consisting of N_i and N_j components respectively, with ε_{ij} inter-dependencies (relationships between the modules of both clusters) as:

$$E_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \frac{\varepsilon_{ij}}{2N_iN_j} & \text{if } i \neq j \end{cases}$$

Inter-connectivity measures the ratio of inter-dependencies between clusters i and j and the maximum possible number of inter-edge dependencies between those clusters. $E_{i,j}$ is 0 when there are no inter-dependencies between clusters i and j, and 1 when every module in cluster i depends on every module in cluster j and vice versa.

We define the **Modularization Quality (MQ)** of a system as a function that expresses the tradeoff between inter- and intra-connectivity. MQ is the objective function of our GA.

Given an MDG partitioned into k clusters, where A_i is the intra-connectivity of the i^{th} cluster and $E_{i,j}$ is the inter-connectivity between the i^{th} and j^{th} clusters, we define MQ as:

$$MQ = \begin{cases} \frac{\sum_{i=1}^{k} A_i}{k} - \frac{\sum_{i,j=1}^{k} E_{i,j}}{\frac{k(k-1)}{2}} & \forall k > 1 \\ A_i & k = 1 \end{cases}$$

MQ establishes a tradeoff between inter-connectivity and intra-connectivity that rewards the creation of highly cohesive clusters and penalizes the creation of too many inter-cluster dependencies. This trade-off is achieved by subtracting the average inter-connectivity from the average intra-connectivity. MQ is thus bound between -1 (no cohesion within the clusters) and 1 (no coupling between the clusters).

3.2. Encoding

As we explained previously, a good encoding of the input data is critical to the convergence speed of the GA and the quality of the obtained solution. For example, the graph in Figure 4 is encoded as the following string S:

$$S = 2 \quad 2 \quad 4 \quad 4 \quad 1$$

Each node in the graph has a unique numerical identifier assigned to it (*e.g.*, node N1 is assigned the unique identifier 1, node two is assigned the unique identifier 2, and so on). These unique identifiers define which position in the encoded string will be used to define that node's cluster. Therefore, the first character in S, 2, indicates that the first node (N1) is contained in the cluster labeled 2. Likewise, the second node (N2) is also contained in the cluster labeled 2, and so on.

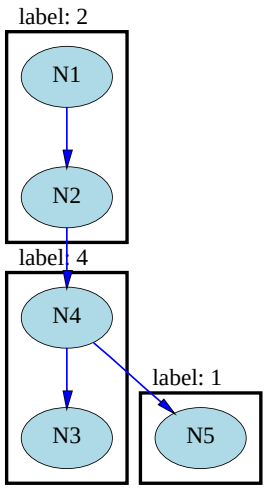

Figure 4. A sample partition

Formally, an encoding on a string S is defined as:

$$S = s_1\ s_2\ s_3\ s_4\ \ldots\ s_N$$

where N is the number of nodes in the graph and s_i, where $(1 \leq i \leq N)$, identifies the cluster that contains the ith node of the graph.

3.3. Genetic operators / population size

As we have mentioned previously, the genetic operators have fixed rates (probability of occurrence) that are problem dependent. The rates we use in our GA are the following (assume N is the number of nodes in the MDG):

- Crossover rate: 80% for populations of 100 individuals or fewer, 100% for populations of a thousand individuals or more, and it varies linearly between those population values.
- Mutation rate: $0.004\ \log_2(N)$. Typical mutation rates for binary encodings are 4/1000. However, because we are using a decimal encoding, we must multiply it by the number of bits that would have been used if the encoding was binary to obtain an equivalent mutation rate.
- Population size: $10N$.
- Number of generations: $200N$.

The formulas for these rates were derived empirically after experimenting with several systems of various sizes.

Having defined the implementation of our GA for automatic clustering, we next demonstrate its effectiveness by applying it to a real software system and evaluating its results.

4. Case study

We used our GA to partition Mini-Tunis [3], an operating system for educational purposes written in the Turing language [4]. We chose Mini-Tunis because it is a nicely designed system with a well-documented structure. Mini-Tunis is also small enough (20 modules) to be explained in a paper, without being a trivial system.

The documentation of Mini-Tunis includes a partitioned MDG. In the Mini-Tunis MDG, ovals represent Turing modules, while edges represent *import* relationships between those modules. As we can see in Figure 5, Mini-Tunis consists of the following major subsystems:

MAIN contains the modules for the front-end of the operating system.

COMPUTER contains the memory management modules.

FILESYSTEM contains two subsystems: FILE, the operating system's file system and INODE, which handles inode references to files.

DEVICE contains modules that handle I/O devices such as disks and terminals.

GLOBALS contains the shared utility modules of the operating system.

We used the Star [7] source code analysis tool to automatically produce the MDG of Mini-Tunis. Subsequently we applied our GA to the MDG.

By looking at Figure 6, we can see that the automatically generated partition is quite similar to the partition found in the Mini-Tunis documentation. The differences indicate that our technique has problems dealing with library and interface modules. For example, the *Inode* module, which is the interface used to access all of the modules in the INODE subsystem, has incorrectly been assigned to the FILE subsystem. This module misplacement is due to a heavy reliance of the FILE subsystem modules on the interface of INODE. Also, *System* and *Panic* are generic library modules. As such, they appear in the cluster where most of the nodes that reference them are located.

Another inconsistency between the documented and the automatically generated partition is that modules from the MAIN and COMPUTER subsystems are in different clusters. We attribute this inconsistency to the possibility that the automatically found solution is a sub-optimal one.

5. Related work

Our work is related to two main bodies of research: software clustering and genetic algorithms. In Section 2 we

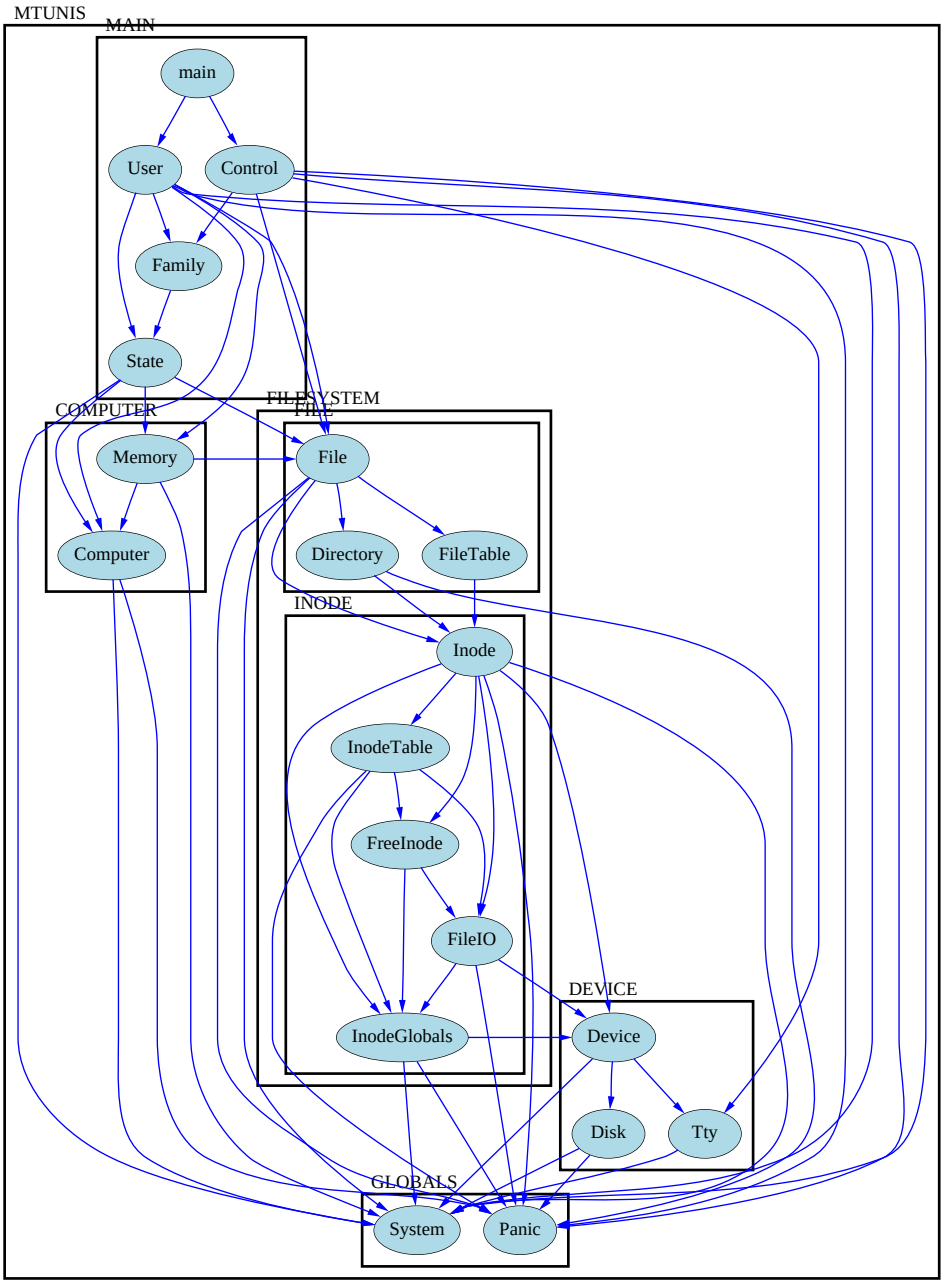

Figure 5. Structure of Mini-Tunis as described in the design documentation

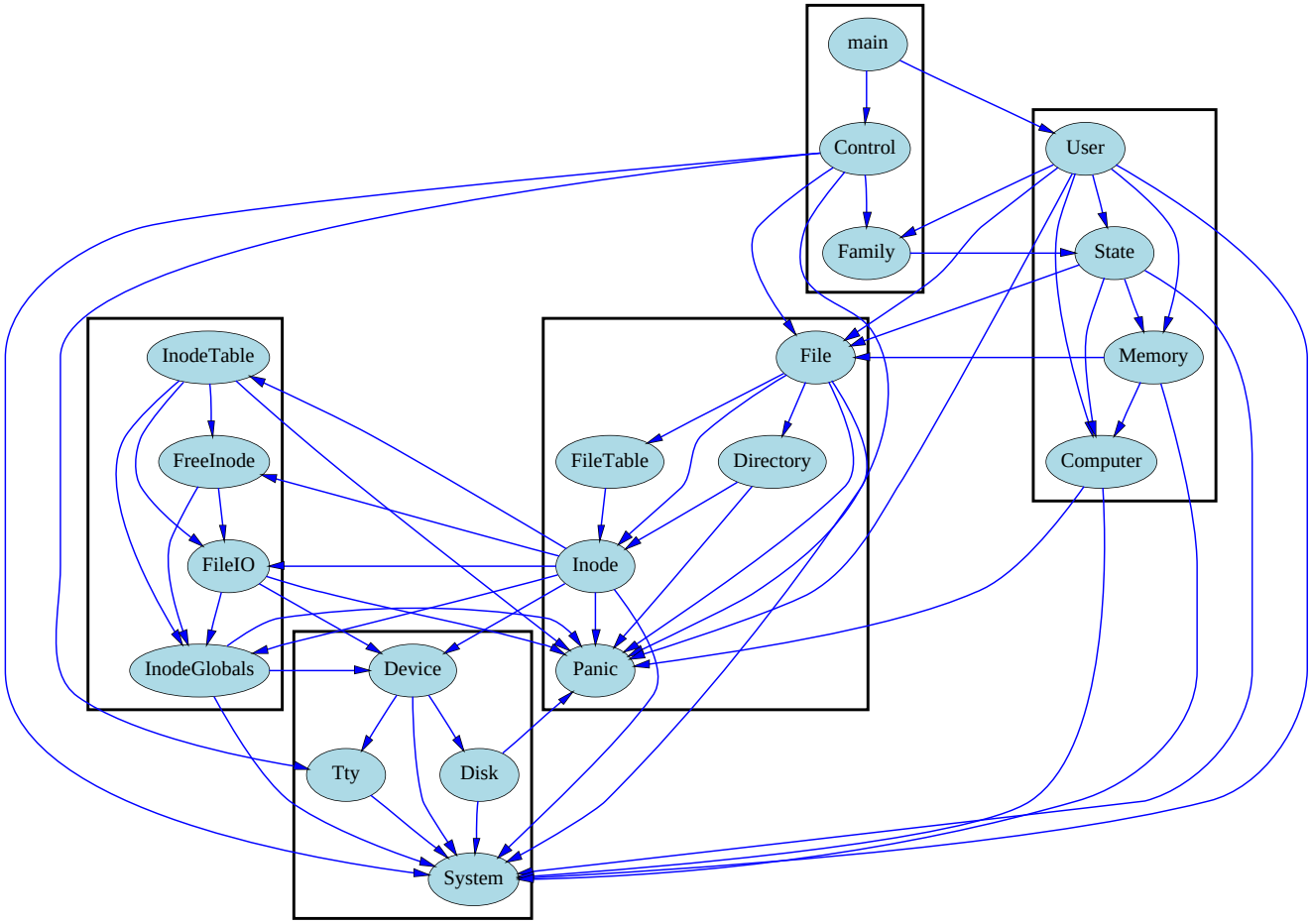

Figure 6. Automatically derived partition of Mini-Tunis

gave an overview of genetic algorithms. In this section, we review some of the previous work on software clustering.

Because source code is usually the only available specification of a system, significant research effort has been devoted to techniques that analyze source code to recover the high-level structure of a system.

A recent paper by Wiggerts [15] provides a good introductory survey of clustering techniques that have been successfully applied to system modularization.

Similar to our approach, many clustering techniques work in a bottom-up fashion by using information extracted from the source code to produce high-level structural views of the system organization. Rigi [10] and Arch [13] are two such tools. These tools, however, require significant user input to direct their respective clustering algorithms.

Hutchens and Basili [5] present a fully automatic clustering technique based on data bindings. They present several techniques for clustering related procedures into modules. Our research addresses a slightly different problem, since it focuses on grouping related modules into subsystems.

Several software modularization techniques adopt a top down approach. For example, in the Software Reflexion Model [11] technique, the designer develops a mapping between a high-level model and the actual source code. Once this mapping is complete, a tool is used to calculate a reflexion model that shows how the designer's model differs from the provided high-level model.

6. Conclusions & future work

Today's software systems are growing in complexity and size. The structure of a large software system is not always apparent from its source code. Even creating a module dependency graph may not be helpful enough to understand the structure of a system. A high-level partition of the graph can make an MDG easier to understand. However, the number of possible partitions of even a small graph can be very large [8]. In this paper we demonstrated how this extremely large space can be explored in a systematic way using a genetic algorithm to find a good partition for a given software system.

We have evaluated our technique on several systems of different size with similar success to the one demonstrated in our case study. In the future, we intend to conduct further validation of our technique using other systems. We also intend to experiment with different genetic operators, parameters, and input encodings, to investigate how they can improve the performance of the GA.

Our technique currently does not provide a mechanism to integrate a designer's knowledge of a system into the automatic clustering process as the Software Reflexion model does. We are currently adding capabilities that enable the user to take advantage of that knowledge.

Also, our objective function treats all module dependencies equally. For example pairs of modules that call each other once are considered equivalent to pairs of modules that call each other twenty times. In an effort to incorporate this concept, we are adding "weights" to the edges of our MDGs and incorporating these weights into our MQ calculation.

7. Access to our software

Interested readers may download a copy of our software from the Drexel University Software Engineering Research Group (SERG) web page at http://www.mcs.drexel.edu/~serg.

8. Acknowledgements

This research is sponsored by a CAREER Award from the National Science Foundation (NSF), under grant CCR-9733569. Additional support was provided by a grant from the research laboratories of American Telephone and Telegraph (AT&T).

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, the U.S. government, or AT&T.

8.1 Note

Diego Doval is now at IBM Thomas J. Watson Research Center, and he can also be reached at ddoval@us.ibm.com

References

[1] Y. Chen, E. R. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Proceedings of the European Conference on Software Engineering/Foundations of Software Engineering*, 1997.

[2] D. E. Goldberg. *Genetic Algorithms in Serach, Optimization and Machine Learning.* Addison Wesley, New York, New York, 1989.

[3] R. C. Holt. *Concurrent Euclid, The UNIX System and Tunis.* Addison Wesley, Reading, Massachusetts, 1983.

[4] R. C. Holt and J. R. Cordy. The Turing Programming Language. *Communications of the ACM*, 31(12):1410–1423, December 1988.

[5] D. Hutchens and R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, pages 749–757, Aug. 1995.

[6] B. Krishnamurthy. *Practical Reusable Unix Software.* John Wiley & Sons, Inc., New York, 1995.

[7] S. Mancoridis, R. C. Holt, and M. W. Godfrey. A Program Understanding Environment Based on the "Star" Approach to Tool Integration. In *Proceedings of the Twenty–Second ACM Computer Science Conference*, pages 60–65, March 1994.

[8] 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.

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

[10] H. Müller, 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, Aug. 1992.

[11] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. ACM SIGSOFT Symp. Foundations of Software Engineering*, 1995.

[12] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, 17(4):40–49, October 1992.

[13] R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. 13th Intl. Conf. Software Engineering*, May 1991.

[14] M. Shaw and D. Garlan. *Software Architecture: Perspectives of an Emerging Discipline.* Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[15] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Working Conference on Reverse Engineering (WCRE97)*, 1997.