A Novel Approach to Modularization Programming of Legacy Systems

Chun-Che Huang
Department of Information Management, National Chi Nan University

Hsi-Chuan Ho
Department of Management Information System, National Chengchi University

Jiann-Min Yang
Department of Management Information System, National Chengchi University

Abstract

The modularization of large legacy software systems has attracted a great deal of attention in recent years. Such modularization improves the maintenance and reuse of smaller, more manageable pieces of program source codes, and also provides insight into the overall structure of the software system. This paper develops a methodology to determine the modularization programming of a legacy software system. This methodology breaks programs into *modules*, or groups of related functions based on the observation that a module can be defined as a group of functions having *high cohesion* and *low coupling*. Furthermore, the determination of good alternatives to perform the desired attributes, with some fuzzy constraints, is crucial in building software systems. With known modules, a rule-based fuzzy representation of the module development problem is presented, while the tradeoff of the attributes of performance among modules is analyzed, using a fuzzy neural network approach. The approach taken to reach this solution is illustrated with simulation software.

Keywords: Program modularization, legacy system, fuzzy associative memory.

大型既有軟體系統程式模組化方法之研究

黃俊哲
暨南國際大學資訊管理研究所

何習銓
政治大學資訊管理研究所

楊建民
政治大學資訊管理研究所

摘要

最近幾年，對於大型既有軟體系統的模組化已經引起相當大的關注。這些模組化，改善了原始碼的維護性及其再使用性，也對於軟體系統的整體架構提供更清晰的觀察。本篇論文發展了一個決定既有系統的模組化程式的方法論。此方法論將其相關功能分群，切割程式成數個模組，因為一個模組可以被定義為一群具有高凝聚力或低耦合度的功能性單位。此外，如何應用模糊限制來決定較好的解決方案來呈現所需的屬性，也是建構軟體系統的關鍵。對於已知的模組，使用規則導向的模糊表達方式，來表達模組建構的問題，同時應用模糊類神經的方法，分析模組之間效能相關屬性的取捨。此方法已成功解決模擬軟體所需模組化的問題並將解決的過程詳加描述。

關鍵字：模組化程式, 既有系統, 模糊關聯記憶。

1. INTRODUCTION

The software industry continues to grow at a fast rate. New software technologies and new system features quickly outdate current systems. As these legacy systems change, personnel skills migrate to newer technologies, leaving fewer people capable of maintaining the older systems. More important is the compatibility of these older systems with the newer ones. Re-engineering is one of the candidate technologies available to solve this problem. Jacobson's decision matrix decides the situation in which it is reasonable to re-engineer software (Fig. 1) [17]. If a legacy system is constantly changing, it is reasonable to just maintain it. If a system is difficult to change, however, and more radical changes are needed, re-engineering is worthwhile if it has a high business value. A system with a high changeability, and a low business value, is not worth re-engineering.

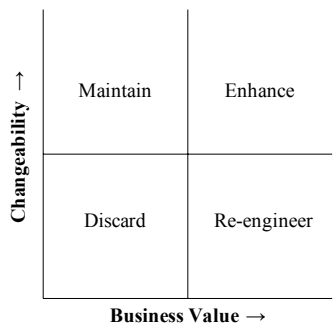


Fig. 1. Decision matrix for re-engineering/maintenance[17].

This paper considers the re-engineering of non-object-oriented legacy software systems [25] since these systems have become obsolescent in terms of their architecture, the platforms on which they run and their suitability and stability in adapting to environmental changes. The non-object-oriented legacy software systems are coded in procedural languages, such as COBOL, FORTRAN, and RPG [6]. The essence of software re-engineering is to improve or transform existing software so that it can be understood, controlled, and used more effectively. Software re-engineering is important in order to recover and re-use existing software assets, bring high software maintenance costs under control and establishing a base for continuous evolution [22].

Re-engineering is the examination, analysis and alteration of an existing software system so that it can be reconstituted into a new form [22]. The goal is to understand the existing software (specification, design, and implementation) and then by re-structuring it, to improve the system's functionality and performance. The objective is to maintain existing functionality and lay the groundwork for additional functionality at a later date.

This process typically encompasses a combination of procedures as shown in Fig. 2 [26], which gives an example of a re-engineering process. The input to the process is a legacy program, while the output is a structured, modularized version of the same program. At the same time the program re-engineering takes place, the data for the system may also be re-engineered. The activities involved in this re-engineering process are:

- Source code translation: The program is converted from an old programming language to a more modern version of the same language or to a different language.
- Reverse engineering: The program is analyzed and information extracted, which helps document its organization and functionality.
- Program structure improvement: The control structure of the program is analyzed and modified to make it easier to read and understand.
- Program modularization: Related parts of the program are grouped together and, where appropriate, redundancy is removed.
- Data re-engineering: The data processed by the program is modified to reflect program changes.

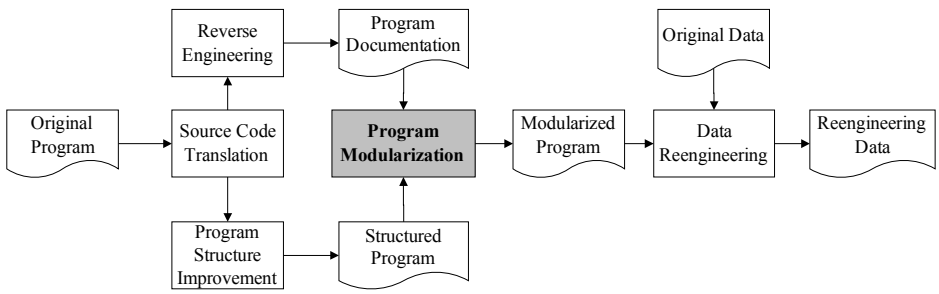


Fig. 2. The re-engineering process [26].

Many studies have contributed to the aforementioned processes and have made us aware of the problems [2,10,11, 27]. When facing the complex program codes of a legacy system, engineers may be unable to carry out re-engineering if they have only the current inadequate system documentation to refer to. If a modularization methodology is available, dividing programs into functional blocks, with new documents being generated based on these functional blocks, the system will be much easier to understand. Consequently, this paper focuses on program modularization, which can enhance subsequent processes.

Program modularization is the re-organisation of programs, which gathers together related functions into a single *module*. Once this has been done, it becomes easier to remove redundancies within these related functions, to optimise their interactions and to simplify their interfaces with the remaining programs. Program modularization is usually a manual process, carried out by program inspection and re-organisation and requiring identification of the relationships between functions and their roles. Although browsing

and visualising tools help in this process, it is impossible to automate it completely. In this paper, "function" is defined as an executable "statement" in programming languages and is a portion of codes within a larger program, which performs a specific task. A subroutine is often involved such that the subroutine can be executed ("called") numerous times and/or from numerous places during a single execution of the program, possibly even by itself. A function consists of numerous statements, *statement1*, *statement2*, etc…and each statement uses parameters to determine what jobs are required to work. For example, the following statement "block" in Pascal language is viewed as a function.

BEGIN

…

...

END

A module is a basic unit of software development, maintenance, and management. One of the basic activities of the software re-engineering process is the partitioning of the software specifications into a number of program modules, which together satisfy the original program statement [7]. From a re-engineering standpoint, the partitioning of system programs, into a number of program modules, improves the roles of both maintenance and management. Anquetil and Lethbridge [1], as well as Wiggerts [28] applied clustering algorithms to legacy system modularization or re-modularization. Jermaine [18] proposed the k-cut method to carry out modularization. While all these approaches are developed for modularization programming of legacy systems, however, two issues have been concerned. (i) They do not explicate modularity properties, e.g., function-swapping and function-sharing among programs. The aforementioned approaches cluster programs based on the frequency of interaction and with the consideration of cohesion and coupling only. From the extended application standpoint, modularity properties are key factors in optimizing modularization programming. With the consideration of the modularity properties, a legacy system can be maintained much more effectively based on the properties than without this concern . For example, if an error/bug occurs, an engineer is able to detect and handle this error quickly based on the property, which recognizes which function is swapped or shared. It reduces the impact of code modification and debug in the maintain stage. From the standpoint of function expansion in the legacy system, additional functions can also be embedded to a module effectively based on the properties, which results in a different type of subsystems and different types of functionality. The modularity property, re-usability of this module, therefore, can help engineers to efficiently develop new subsystems. (ii) The previous approaches determined clusters (modules) in a crisp way and include all nodes which have high frequency interaction definitely into modules without the consideration of function independency. In addition, a program function may exist on the boundary of clusters (referred to as

boundary functions); it is difficult to identify which cluster a boundary function should belong to without incorporation with the developers' programming experience, design rules and literature results. In the tradition cluster methods, for example, the researchers in [1, 18, 28] determine the boundary function randomly, which definitely increase the risk of misplacement.

This paper develops a methodology to determine programming modularization of non-object-oriented legacy software systems. This methodology breaks down the programs into *modules*, or groups of related functions, based on the observation that a module can be defined as a group of functions having *high cohesion* and *low coupling*. That is, interaction among functions *within* a module is frequent, but interaction *among* functions in different modules is rare. In addition, the types of modularity and independent functions are identified to aid the re-use of further syetm development. To include the boundary functions, a fuzzy neural network approach is used, with the goal of discovering "how to cluster the boundary functions into modules so that the desired attributes can be performed at a reasonable cost." The rules derived from the programming experience, design rules and literature results are involved to aid the decion-making of module development and provide attributes are performed at a reasonable cost. The proposed rule-aided solution approach provides a systematical process to reduce the misplacement of boundary functions, which is not considered in previous works. The complete process is illustrated in Fig. 3.

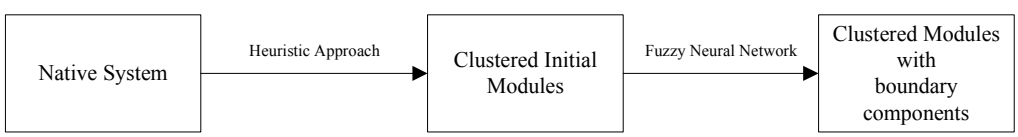


Fig. 3. The Process of Modularization.

This paper is organized as follows. Section 2 illustrates an interaction graph used to represent software modularity, while the functions in a module are determined by a heuristic approach in Section 3. With most functions determined, Section 4 presents a fuzzy neural network approach to analyze the tradeoff between the attributes of performance among modules and to include a boundary function in the desired module. The proposed approach is applied to a simulation software. The conclusion is presented in Section 5.

2. SOFTWARE MODULARITY

2.1. Concept of modularity

In software architecture, a possible approximation of the concept of interaction between functions is by locating function invocations. Thus, when one function "calls" another (as determined through static program analysis and a mapping table from code to function), an interaction is said to exist among the functions. The network of function invocations making up a large program can be represented by a graph, where the nodes and vertices in the graph represent functions, and the edges represent invocations [18]. A desired module is one having a low frequency of function invocations (low degree of function interaction) within a module and a high frequency of function invocations (high degree of function interaction) between modules (Fig. 4).

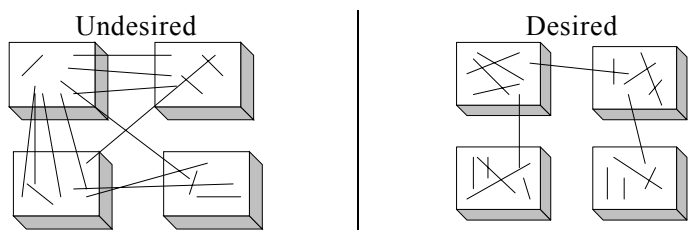


Fig. 4. Coupling and cohesion in two software systems.

Furthermore, the classification of interactions among modules is described as coupling and cohesion [5,21,26,29]. Coupling is an indication of how tightly the modules are interconnected. High coupling, that is, a large number of connections, is generally bad (Fig. 4). Cohesion, on the other hand, is a measure of how well a particular module's functions - its code and local data structures - bind together. High cohesion, that is, a strong measure of locality, is good. The essence of the approach proposed, therefore, is to divide the software into modules, which will result in there being a minimum interaction between modules (low coupling) and a high degree of interaction within each module (high cohesion). A well-defined module can be designed, coded, tested and amended, without the undue complication of having to deal with other modules. An important benefit is that when an error occurs in a module, the spread of damage to other modules can be minimal [4]. Concepts such as coupling and cohesion are very often interpreted differently and inconsistent definitions can be found in the literature [5].

Myers defined seven levels of cohesion [21]. In descending order, these are functional, informational, communicational, procedural, classical, logical and coincidental. A high (functional)-strength module performs a single well-defined function. In the light of

modern theoretical computer science, Myers' first two levels are considered to be equally good (Table 1). A number of coupling levels can also be distinguished: in descending order, they are Data, Stamp, Control, Common and Content [21] (Table 1).

Table 1. Cohesion/Coupling types [21]

	Level	Type	Weight score		Level	Type	Weight score
Cohesion (W_{ch})	Good	Functional cohesion	7	Coupling (W_{cp})	Good	Data coupling	5
		Informational cohesion	6			Stamp coupling	4
		Communicational cohesion	5			Control coupling	3
		Procedural cohesion	4			Common coupling	2
		Temporal cohesion	3		Bad	Content coupling	1
		Logical cohesion	2				
	Bad	Coincidental cohesion	1				

2.2. Module representation

The characteristics of interaction mentioned above imply that two types of relationships are involved in modularity:

1. a high degree of functional interaction within a module (cohesion), and
2. a low degree of functional interaction between modules (coupling).

Based on these two relationships, an interaction graph is used to represent functions within the program and a clustering technique is used for grouping. In abstract terms, modularization represents the grouping of large amounts of program source codes in groups (modules) in such a way that these program source codes in one group are much more closely related to each other than they are to program source codes belonging to different groups. In cluster analysis, such groups are called clusters. In this paper, the terms "module" and "cluster" refer to the same notion, but "module" is used in the viewpoint of describing software systems and "cluster" in algorithmic viewpoint. Everitt [12] defined clusters in a similar way, as "continuous regions of space containing a relatively high density of points, separated from other such regions, by regions containing a relatively low density of points". This is a very general definition, which appeals to our intuition.

The interaction graph (A, E, W), is a graph with weights on the edges, where A is the node set representing functions, E is the edge set representing interactions and W is the weight set of frequencies of function invocations, in this set of programs.

An example of an interaction graph is shown in Fig. 5, where A ={*l*, *m*, *n*, ...}and E =

{(*l*, *m*), (*m*, *n*), …}. The weight w(*l*, *m*) = 8 indicates that the interaction (function invocation or call) from function *l* to function *m* occurs eight times.

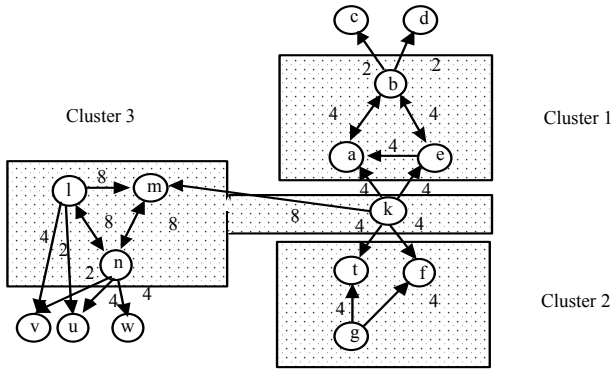


Fig. 5. An example of an interaction graph.

The *weight density* of a subgraph is defined as the ratio of the total weight of the cohesion edges included in the subgraph, over the number of cohesion edges included. The weight density determines the quality of clusters.

In this paper, a cluster in the interaction graph is represented as a set of connected nodes, N that satisfies:

$$wdN - wij > \theta \quad \text{for all } i \in N \text{ and } j \notin N,\ N \subseteq A \tag{1}$$

where wdN: weight density of the set N of nodes (functions)

wij: weight on the intra-edge from node $i \in N$ to $j \notin N$

θ: threshold index used to include a node in (or exclude from) a cluster.

For example, consider cluster (module) 3 = {*k, l, m, n*} in Fig. 5. The weight density (wd3) of cluster 3 is 8. If θ is set to 5, no additional nodes can be included in cluster 3. If θ is set to 4, then nodes *a, e, t, f, w, u* and *v* would be included in cluster 3. The larger the value of θ, the smaller the size of a cluster. The value of the module threshold index θ is determined by human experience.

The functions (nodes) that do not belong to any module (cluster) are referred to as independent functions (nodes). The independent functions (nodes) may refer to the functions (nodes) either absolute independent (none/few interaction with other functions) or *basic functions* (nodes), which can be incorporatd with other module(s). In case that the basic functions (nodes) are incorporated with other module(s), different types of modular subsystems are generated, e.g., the basic functions *c* and *d* in Fig. 5.

The interaction graph allows the representation of different types of modularity. From the viewpoint that the initial modules are clustered, how can these modules be maintained, and new function(s) added, so that the efficiency is enhanced by re-engineering? To answer this question, different types of modularity are explored and discussed:

Property 1: Function-swapping modularity

In the interaction graph representing a module (cluster) and a basic function set (basic node set) that form function-swapping modularity:

1. The nodes in the basic node set are the leaves of the cluster, and
2. The weight density of the cluster equals the total weight of all coupling edges between the basic node set and the cluster.

Example 1: The function-swapping modularity is illustrated in Fig. 6(a), where three software functions are within module 1 = {*a*, *b*, *e*}; module 1, when incorporates with function *c* or function *d*, makes up two different subsystems. From the viewpoint of maintenance, this module plays a core role. Since similar functions interact with this module, engineers can quickly understand the system. For example, if an error or bug occurrs, engineers will quickly be able to conclude which module is abnormal, based on the type of error or bug. From the viewpoint of function embedment, a new function can be added to this module, which results in a different type of subsystem. Re-use of this module, therefore, can help engineers to efficiently develop new subsystems.

For example, in a point of sale (POS) system, by using different entity objects, e.g., charts and template objects with the same object generation process, different types of reports can be generated. In database management system (DBMS) software, by using various types of query parameters/formats and different output formats with the same database table, different output forms can be presented to the decision makers.

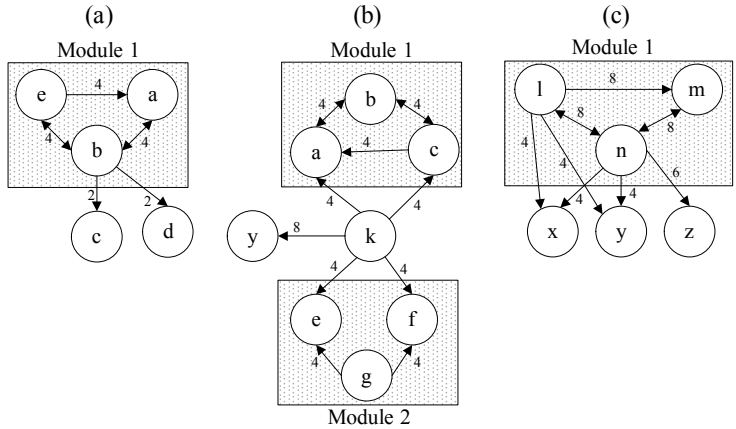


Fig. 6. Modularity represented by graphs: (a) function-swapping modularity, (b) function-sharing modularity, and (c) bus modularity.

Property 2: Function-sharing modularity

A *cut node* is defined as a basic node providing a unique connection between clusters. Removing the cut node results in disjointed clusters.

The interaction graph represents a modular (cluster) set and a basic function (node) that form function-sharing modularity.

1. The node is a cut node connecting all clusters in the cluster set, and
2. The total of the weight density of each cluster in the cluster set equals the weight of the coupling edge incident to the cut node

Example 2: Function-sharing modularity is illustrated in Fig. 6(b), where the two modules use node *y* and basic node *k*. The module set comprises module 1 and module 2, the cut node is *k*, and the coupling edge is (*y, k*). The cut node plays a core role. This function directly interacts with many other modules. Therefore, the engineers must put more effort into keeping it at a normal status. If this core role is modified or replaced, it could possibly cause a ripple effect or an unsupported accident.

For example, function-sharing modularity can lead to the use of the same statistical testing functions, e.g., the $\chi 2$ testing function in different Quality Control (QC) software packages. The use of the reporting function allows the general ledger module and the accounts payable module to generate various types of report in an accounting system.

Property 3: Bus modularity

The interaction graph represents a module (cluster) and a basic function (basic node) set that forms bus modularity.

1. All nodes in the node set are leaves of the cluster
2. The density weight of the cluster is greater than any weight on the coupling edge between the basic node in the node set and the cluster, and
3. The nodes in the node set are independent; i.e., they do not interact with each other.

Example 3: The bus modularity is illustrated in Fig. 6(c) where the 3 different functions within module 3 are {*l, m, n*}. This module simultaneously interacts with several other external components and has a naturally high coupling rate among other functions. It relies more on these other functions to process its tasks. If the module is modified or replaced with a new one, the ripple effects will be considerable. Therefore, when maintaining or adding new function(s), engineers must pay close attention to the interactions among them.

Examples of bus modularity are: the query systems of a distributed database, in which clients use various input/output functions, each with a distinct type of I/O format, to query different types of data across various platforms of servers through the Internet.

It should be noted that, for the above three properties, when a basic function is replaced with a module interacting with other modules, it can result in different types of function-swapping, function-sharing or global bus modularity.

3. SELECTION OF FUNCTIONS INTO A MODULE

The module in the interaction graph is represented as a set of connected nodes that satisfies inequality (1) in Section 2. We then continue with the task of selecting functions.

3.1. Problem Formulation

The task of selecting functions is formulated as follows:

Group the nodes of the interaction graph into mutually separable clusters (modules) with:

1. the minimum total weight density of intra-clusters, and
2. the maximum total weight density of inter-clusters,

This is subject to the following constraints:

Constraint C1: Empty clusters of functions are not allowed.

Constraint C2: The number of functions in a cluster cannot exceed the upper bound Nu.

Constraint C3: The weight density on the inter-edges minus the weight on the new intra-edge is greater than the threshold index θ.

Constraint C4: The inter-edges (cohesion) in a cluster require a high level of cohesion, e.g., functional or informational cohesion.

Constraint C5: The intra-edges (coupling) across a cluster require a high level of coupling, e.g., data or stamp coupling.

Clustering allows exploration of potential modules among software functions and to analysis of various types of modularity. The challenge is to group functions into a module, each with acceptable size and characteristics.

Next, a heuristic algorithm is applied to determine clusters (modules) in an interaction graph. In this approach, the nodes incident to a cluster with the maximum weight and a high level of cohesion are included in the cluster. The only case in which the algorithm excludes a high weight edge is when this edge implies inclusion of an interaction function that will make the module infeasible, i.e., inconsistent with the functions included.

3.2. Heuristic Algorithm

The non-object-oritend lagecy software systems written in procedural languages which are typically composed of a set of programs related through external/internal calls and can be represented by a graph whose nodes represent the functions and whose edges

depict the call relation. To date, there are several tools available to generate such graph [9,14,24]. The proposed algorithm for identifying the cluster of statements contributing to modularization programming of legacy system is based on the analysis of weight density summarized in a graph. The algorithm takes an interaction graph as an input and returns a clustered graph as follows:.

Input: Interaction Graph

Output: Clustered Graph

Step 0 (Initialization)

Specify the upper bound NU on the number of functions in the module and the threshold index θ. Unlabel all edges.

Step 1 (Labeling)

Select unlabeled edges with the same largest weight and label them; i.e., remove them from the candidates for the selection set.

Step 2 (Clustering)

Identify the connected subgraphs with all new edges labeled. Combine the nodes in a subgraph into a cluster as long as the constraints C1, C2, C3, C4, and C5 are satisfied. Repeat Step 1 until no more nodes can be combined.

Step 3 (Identification)

Identify modules corresponding to the clusters in the graph.

Step 4 (Classification)

Classify the modules generated, based on the three properties in Section 2.

Step 5 (Termination)

Stop and produce the output of results.

Example 4: Selection of simulator modules

The following example illustrates a simulated system. The system is composed of 6,100 lines of C codes and 80 functions. The main features of the system include: entities, departure, arrival, process, queues/stacks, statistical accumulator, animator and simulation clock. All features are carried out by the functions composed of numerous lines of programming. The codes of the nodes (functions) are illustrated in Table 2. Each function is represented with a three-field code. The set of partial functions, with the interaction graph, is presented in Fig. 7. The initial interaction graph is generated by Graphviz –Codeviz package [14].

Table 2. Codes of node in the interaction graph.

Field	Attribute
First field of the code	
F	Fundamental function
C	Conditional function
Pc	Control function
Pa	Admin function
R	Random generator
Em	Empty node that indicates the ending process
The remaining fields	Function number

The heuristic algorithm is applied to the functions of the simulation system to determine the modules. The resulting interaction graph of the functions generated by the heuristic algorithm is illustrated in Fig. 8.

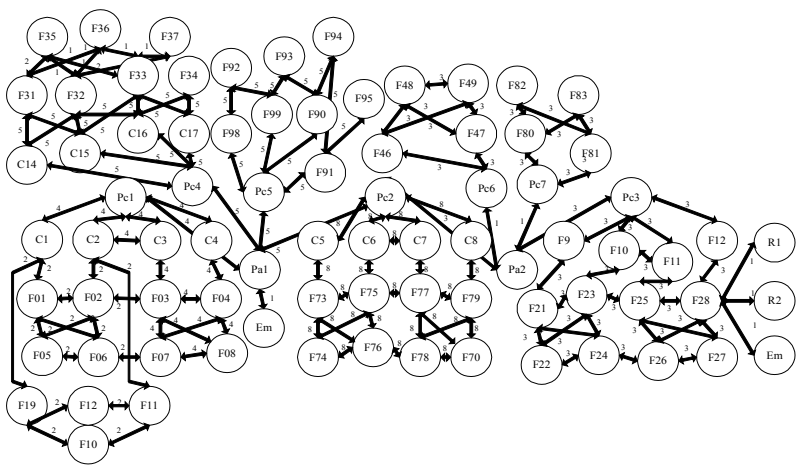


Fig.7. The partial interaction graph of a simulation system.

Results

1. Nine modules are formed:

M1 = {Pc2, C5, C6, C7, C8, F73, F74, F75, F76, F77, F78, F79, F70}, namely DEPARTURE module.

M2 = {Pc4, C14, C15, C16, C17, F31, F32, F33, F34}, namely ENTITY module.

M3 = {Pc5, C90, C91, C92, C93, C94, C96, C98, C99}, namely BASIC PROCESS module.

M4 = {Pc1, C1, C2, C3, C4, F3, F4, F7, F8}, namely ADVANCED SERVER module.

M5 = {Pc3, C9, C10, C11, C12, F21, F22, F23, F24, F25, F26, F27, F28}, namely ARRIVE module.

M6 = {F46, F47, F48, F49}, namely ANIMATE module.
M7 = {F80, F81, F82, F83}, namely ADVANCED PROCESS module.
M8 = {F01, F02, F05, F06}, namely QUEUE module.
M9 = {F10, F11, F12, F19}, namely STACK module.
Based on the three properties of Section 2, the following types of modularity are determined:

2. Module M5, with the number generators R1, R2, and an empty node, form the function-swapping modularity, where the ARRIVE (M5) may cooperate with the exponential number generator (R1), the triangular number generator (R2) and other generators to create various types of arrival entities.
3. Module M3, with module M4 and an empty node, form the module-swapping modularity, where the BASIC PROCESS (M3) may corporate with the ADVANCED SERVER (M4) to enhance functionality or retain basic functionality (with an empty node).
4. Module M4, with modules M8 and M9, form the module -swapping modularity, where the same ADVANCE SERVER (M4) may cooperate with the QUEUE (M8) or STACK (M9) alternatively, to form different types of waiting systems.
5. Module M1, with modules M3 and M7, form the module sharing modularity, where the BASIC PROCESS and ADVANCED SERVER (M3 and M7) share the same DEPARTURE (M1).
6. Module M2, with functions F35, F36 and F37, form bus modularity, where the ENTITY (M2) may cooperate with the transportation tools, forklift unit 1 (F35), forklift unit 2 (F36) and forklift unit 3 (F37) to deliver the entities from module 2 to other modules.

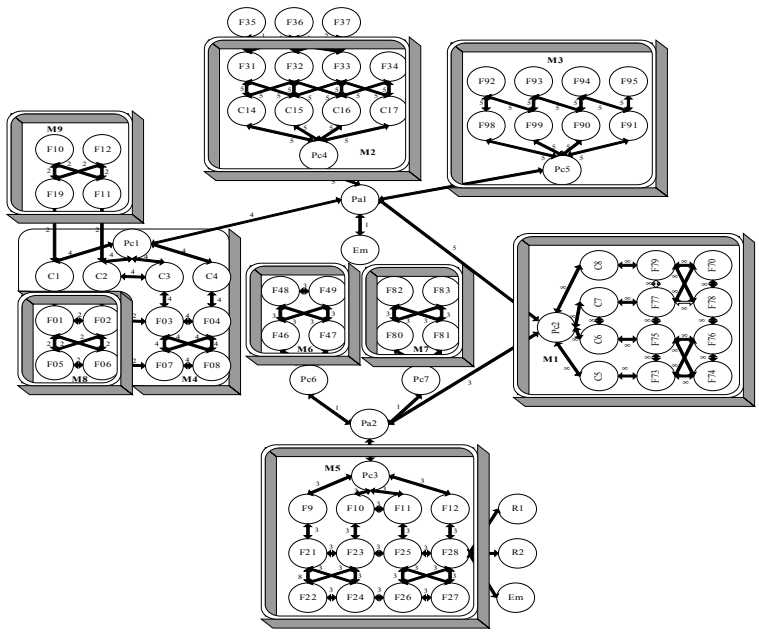


Fig. 8. The resulting modules of simulation system.

4. DEVELOPMENT OF A MODULE

In Section 3, the functions are selected via clustering, using the heuristic algorithm. However, numerous functions on the boundary of clusters (called *boundary functions*), e.g., Pa1, Pa2, Pc6 and Pc7 nodes of Fig. 8, are difficult to either include or exclude from a module by using the heuristic algorithm. In this section, with most functions known, the module development problem is centered on "how to cluster the boundary functions into modules so that the desired attributes can be performed at a reasonable cost." A fuzzy neural network approach, therefore, is used to develop a module.

Generic performance attributes of modules are as follows [3, 13, 16]:

- Simplicity. The module's function must be simple and easy to understand.
- Well-defined invocation. Each module should have a well-defined set of input and output data elements. The invocation (execution) of any procedure in a module should be obvious.
- Locality. The module should contain all code and data necessary to perform its function, except for data passed into the module as part of its invocation.
- Free of side effects. The module's invocation should not influence another module's operation, except by the direct return of data.

A module is usually generated as a result of the attribute trade-off between two

modules, such as small size, well-defined invocation, locality and few side effects. The analysis of attribute trade-offs is a complex task; specifically, performance attributes are only described in fuzzy terms early in the program modularization stage, e.g., small size, well-defined invocation, or few side effects. From the perspectives of performance and cost, a large number of available program modularization options preclude an exhaustive analysis of viable alternatives. Hence, the determination of good alternatives, which will satisfy the performance requirements at a reasonable cost, with some fuzzy constraints is crucial in building the module.

Neural networks model complex interconnections between two spaces; however, they are not well suited for interpreting human knowledge [15]. On the other hand, rule-based systems are effective in knowledge representation. The fuzzy approach can provide decision-support and arm rule-based systems with powerful reasoning capabilities [8]. In order to take advantage of these approaches, a fuzzy associative memory (FAM) approach [19] is used to model the predicament of module development. The FAM approach is illustrated with the simulation software.

4.1. Fuzzy Rules and Knowledge Representation

The program modularization rules for the module, and the selection of performance attributes at an early program modularization stage, may be essentially generated using fuzzy rules, described as follows:

Corresponding to a set of boundary functions {CBC1, …, CBCi, …, CBCl}, cluster the boundary functions to the modules in the set of {M1, …, Mj, …, Mm}.

Premise: P1 is A1, P2 is A2, P3 is A3, P4 is A4, P5 is A5, P6 is A6, P7 is A7

Conclusion: O1 is B1, …, or Ok is Bk, ..., or On is Bn, where Ok = (M1, …, Mj, …, Mm, COST)k

Notations

i: index of boundary functions
j: index of possible modules
k: index of alternatives
l: the number of boundary functions
m: the number of set modules
n: the number of alternatives

P1: size of the module
P2: simplicity
P3: well-defined invocation
P4: locality
P5: side effects
P6: type of cohesion
P7: type of coupling

A1, ..., A7: fuzzy members represented by linguistic values∈ {S, SM, M, ML, L}, where S: small, SM: small-medium, M: medium, ML: medium-large, L: large.

O = {O1, ..., Ok, ..., On} output alternatives, where Ok = (M1, …, Mj, …, Mm)k, k = 1, ..., n.

M1, …, Mm: m modules, some of which may include the boundary functions in the

set of CBC.

Bk = (b1, …, bij, …bm)k, k = 1, ... n, fuzzy members where bij∈ {S, SM, M, ML, L} indicates the possibility level that boundary function CBCi i∈ module Mj.

COST: cost function = $\sum_{i}$ wiAi, i = 1, …, 7, where w1, …, w7, the weights of A1, …, A7, are determined by the software characteristics.

The fuzzy rules describing the relations between the performance attribute space and the output alternative space are represented as follows:

Rk≡(A1∩A2∩A3∩A4∩A5∩A6∩A7)→Bk for the alternative k, k= 1, …, n.

Note that the rules may use only some of the seven performance attributes, A1, …, A7, depending on the type of software functions. For example, size is often specified as a performance goal in many of the software functions. The side-effect of mathematical functions may impact the correctness of computing. Increasing locality may increase cohesion but can also increase the size of a module.

The relation R is encoded with an encoding scheme called the correlation-minimum encoding, often used in fuzzy associative memory technology [23].

Let uRk(u1, ..., u7, vk) = uA1(u1)∩...∩uA5(u5)∩uA6(u6)∩uA7(u7)∩ uBk(vk)

where: uAt(ut) is the membership function of At, t = 1, …, 7

uBk(vk) is the membership function of Ok

∩ is the pairwise minimum operator.

An example of a fuzzy rule for a simulator is shown below:

IF the boundary function performs multiple, completely unrelated actions, or its action cannot be defined, that is, the module must be described in terms of its logic, rather than its action,

Then it is strongly recommended that it be excluded from the module.

4.2. Network Construction and the Learning Procedure

The network for the generic module development problem is a two-layer feed forward heteroassociative fuzzy network that stores an arbitrary fuzzy spatial pattern pair (Pt, Ok), where t = 1 to 7 and k = 1 to n. Fuzzy Hebbian learning is applied [20]. The network and learning approach used allow integration of numerical methods into a rule-based system without compromising the merits of inexact reasoning and fast decision making [16]. In this approach, the sth pattern pair is represented by the fuzzy set {Pt, Ok} = {(P_1^s, ..., P_7^s), (O_1^s, ..., O_n^s)} and the weight wtk = min{uP(Pt^s), uO(Ok^s)} indicating connection strength from neuron t to k. The architecture in Fig. 9 is built with h subnetworks, one for each fuzzy rule. The storage separation of fuzzy rules consumes more space but provides a good explanation facility for the inference procedure of the fuzzy associative memory (FAM). The user can directly determine which rules contribute to the output, as well as the extent

of their membership activation. This feature overcomes the shortcomings of the poor explanation properties in conventional connectionist models.

Input to the network: (P1, ..., P7)

Output from the network: (O1, ..., On), where O*k* = (M1, …, M*j*, …, Mm, COST), *k* = 1, ..., n

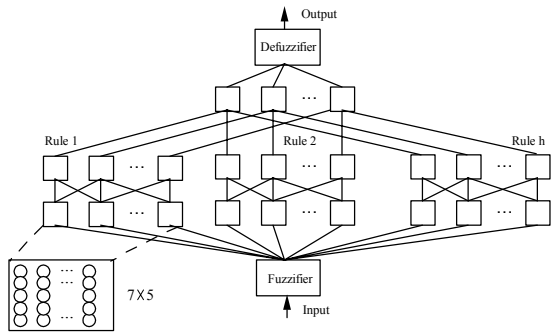


Fig. 9. Network architecture for module development.

The training process of a fuzzy neural network (learning from rules) is different from building conventional neural networks that learn directly from training data sets. In a fuzzy neural network, the premise of a training data set; e.g., performance attributes, are fuzzified through fuzzy membership functions, which can map any numeric measurement of these items into finite fuzzy variables. Fuzzification is actually the process of compressing widely distributed numeric information into a syntactic representation, so that the training of a fuzzy neural network is less computationally intensive, than that of a conventional neural network. The fuzzifiers (membership functions) can be generated from experience and statistics, or can be constructed by the neural network approach. The example of membership functions for linguistic variables (S, SM, M, ML, L) corresponding to performance attributes, is shown in Fig. 10.

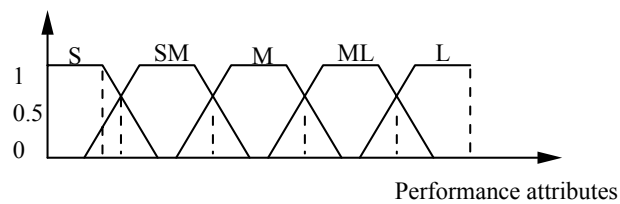


Fig. 10. Membership function of a fuzzy set {S, SM, M, ML, L}.

4.3. Illustrative Example: Simulation Software

Considering the aforementioned example of a simulated system, the boundary functions are included in the desired modules. First, the rules are trained, and then a test case is given as an example.

4.3.1. Training Rules

Examples of training rules, based on the works of Schach [23] and Bellt et al. [4], are presented next. Interpretation of these rules can be found in the references above. The rules perform effectively only on the simulation programs coded with C language and do not guarantee the effectiveness in other package. Different types of computer programs and applications of domain areas involve in different design rules.

- IF the BF (boundary function) performs multiple, completely unrelated actions, or its action cannot be defined, that is, the module must be described in terms of its logic, rather than its action (u-p6-1 =1), THEN it is strongly recommended that it be excluded (u-b*i*.-5 = 1).
- IF the BF has four parameters, but three of them are not needed in a particular situation, and this degrades readability, with the usual implications for maintenance, both corrective and enhancement (u-p6-2 =1), THEN it should be excluded (u-bi.-4 = 0.2 and u-bi.-5 = 0.8)

  For example: module new_operation

  function_code = 7;

  new_operation (function_code, dummy_1, dummy_2, dummy_3);

  //dummy_1, dummy_2, and dummy_3 are dummy variables,not used if function_code is //equal to 7.
- IF the BF performs editing of insertions and deletions and modifications of master file records (u-p6-2 =1), THEN it should be excluded (u-b*i*.-4 = 0.2 and u-b*i*.-5 = 0.8).
- IF the BF performs all input and output (u-p6-2 =1), THEN it should be excluded (u-b*i*.-4 = 0.2 and u-b*i*.-5 = 0.8).
- IF the BF performs a number of actions, each with its own point, with independent code for each action, all perform on the same data structure (u-p6-6 =1), THEN it should be included (u-b*i*.-1 = 0.2 and u-b*i*.-2 = 0.8). An example is given in Fig.11.

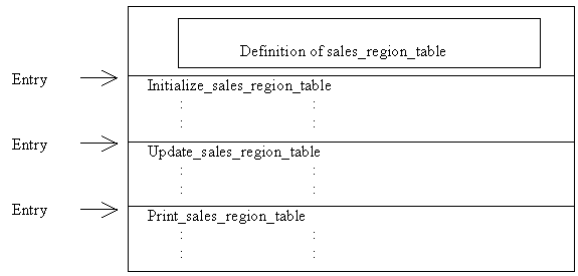


Fig. 11. Module with informational cohesion.

- IF the BF modifies a statement of a module (u-p7-1 =1), THEN it is strongly recommended that it should be excluded (u-b*i*.-1 = 1)
- IF the BF refers to the local data of module q in terms of some numerical displacement within q (u-p7-1 =1), THEN it is strongly recommended that it be excluded (u-b*i*.-5 = 1)
- IF the BF branches to a local label of module q (u-p7-1 =1), THEN it is strongly recommended that it should be excluded (u-b*i*.-5 = 1).
- IF the BF has accessed to the same global data as the other module (u-p7-2 =1), THEN it should be excluded (u-b*i*.-3 = 0.2, u-b*i*.-4 = 0.6, u-b*i*.-5 = 0.2)
- IF the BF passes an element control to another module, that is, one module explicitly controls the logic of the other (u-p7-3 =1), THEN it may or may not be excluded (u-b*i*.-2 = 0.2, u-b*i*.-3 = 0.6, u-b*i*.-4 = 0.2)
- IF a data structure in the BF is passed as a parameter, but the called module operates on some but not all of the individual functions of that data structure (u-p7-4 =1), THEN it should be included (u-b*i*.-1 = 0.2, u-b*i*.-2 = 0.6, u-b*i*.-3 = 0.2)
- IF all parameters are homogeneous data items, i.e., every parameter is either a simple parameter or a data structure, all of whose elements are used by the called module (u-p7-5 =1), THEN it is strongly suggested that it be included (u-b*i*.-1 = 1)

4.3.2. Test Case

After the network in Fig. 10 has been trained, a test case is considered.

Test case:

The set of boundary functions: {CB1, CB2, CB3, CB4} = {Pa1, Pa2, Pc6, Pc7} of Example 4

The module set: {M1, M2, M3, M4, M5, M6, M7, M8, M9}

The performance requirements: medium sized module, medium simplicity, well-defined invocation, low locality, low side-effects, high cohesion and low coupling

The input of P1 is shown in Fig. 12.

COST = 0.4×A1+0.1×A2+0.1×A3+0.1×A4+0.1×A5+0.1×A6+0.1×A7, where the size

of a module (A1) is emphasized.

Input of the test case: {u-p1-3 = 1, u-p2-3 = 1, u-p3-1 = 1, up4-5 = 1, up5-5 = 1, up6-1 = 1, up7-5 = 1}

Output of the test case (two alternatives):

O1 = {u-b1 3-1 = 1, u-b2 7-1 = 1, u-b3 6-1 = 1, u-b4 7-1 = 1, u-COST3 = 0.9}

O2 = {u-b1 3-1 = 1, u-b2 5-1 = 0.9, u-b3 6-1 = 1, u-b4 7-1 = 1, u-COST4 = 0.8}

Alternative O1 suggests that including:

- Boundary function Pa1 in module M3
- Boundary function Pa2 in module M7
- Boundary function Pc6 in module M6
- Boundary function Pc7 in module M7.

The cost of alternative O1 is medium (u-cost index, u-cost = 3, medium) with a membership value (probability) of 0.9.

The O1 output result of basic function Pa1 is shown in Fig. 13.

Alternative O2, suggests including basic function Pa2 in module M5. The cost of alternative O2 is medium-high (u-cost index, u-cost = 4, medium-high) with a membership value (probability) of 0.8. The final solution for the simulation is the O1 alternative, because of its lower cost ($3 < 4$) and higher probability ($0.9 > 0.8$). Note that if the basic function Pa2 is included in module M5, the size of module M5 is clearly larger (thirteen functions). The output results of COSTs for alternatives O1 and O2 are illustrated in Fig. 14. The final result of example 4 is presented in Fig. 15.

In this case, after the software engineer evaluates the results in Fig. 15, specifically the boundary functions, it shows that the proposed solution approach provides a great promise in the modularization programming of legacy systems. For example, in Fig. 15, Pc6 refers to the control unit of image processing and the M6 module to the functions of image presentation. Pc6 is a boundary function since it is a starter; that is, only few interactions with the M6 module and M7 module (printing module) at the beginning of image presentation. The interaction frequency of Pc6 with M6 and M7 is low. Based on previous clustering approaches, Pc6 was not included to M6 incorrectly, which results in difficulty in maintenance since the functionality requirement is not satisfied. In addition, it is not easy to update and replace this image presentation module with new version since the current boundary between M6 and other module is not clarified and Pc6 can not be tracked effectively as an entry point. The risk of calling incorrect functions and passing incorrect parameters between functions are increased in the future maintenance and system update. The similar situation occurs in Pa7, a printing control unit and M7, a printing module.

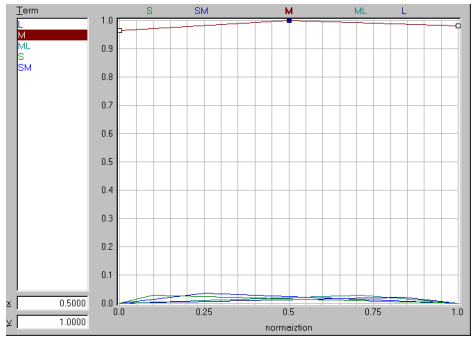


Fig. 12. The input of P1 with requirements (in fuzzy terms) of Medium.

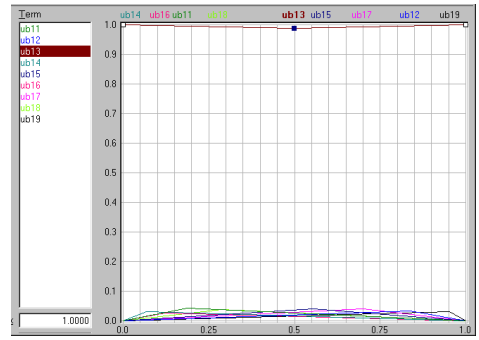
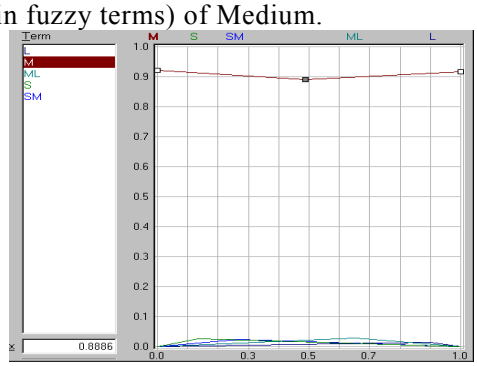


Fig. 13. The O1 output result of basic function Pa1.



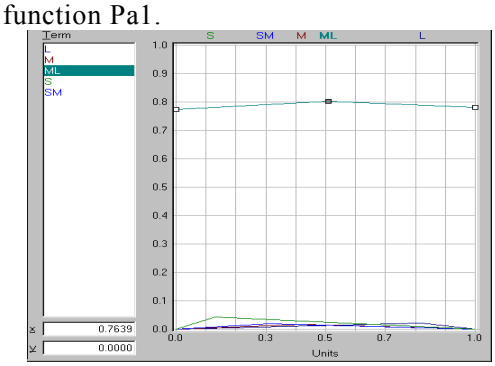


Fig. 14. The COSTs of alternatives O1 and O2.

5. CONCLUSIONS

In this paper, a methodology was developed to identify the programming required to modularize a legacy software system, giving consideration to performance and cost. Program functions were represented with interaction graphs and the modules were determined using a heuristic approach. For the known functions, a fuzzy neural network approach was applied to analyze the tradeoff of the attributes of performance among the modules and clusters of the boundary functions.

The main contribution and goals of this paper was the determination of modules, the identification of modularity type and independent functions, and the construction of program modularization. The maintainability and scalability was practiced and enhanced, since the modules were determined based on interactions among functions, using a heuristic algorithm. The proposed solution approach was practically applied to a simulator package and our goals were successfully achieved.

The following three issues require further study:

1) The generic fuzzy rules should be defined and interpreted by experts. Also, the

input performance attributes and the cost function should be selected based on the characteristics of different types of products. The determination of the threshold index θ to limit the size of Modules and the weights *w*1, …, *w*7 of A1, …, A7 is arbitrary; therefore, further studies are required.

2) Development of an agent-based dynamic analysis system would assist in dynamically analyzing the interactions among systems and probing the activities in detail.

3) An efficient mechanism to determine the weight of the interaction graph is desirable. In the current system, a table that maps source codes into functions is used. To obtain the weight, all codes are scanned and mapped into functions. The frequencies of function invocations (interactions) between any two functions are counted. Consequently, the total frequencies obtained are equal to the set of weights of the interaction graph. Those source codes or functions not included in the mapping table, should have the frequencies of their invocations (interactions) determined manually.

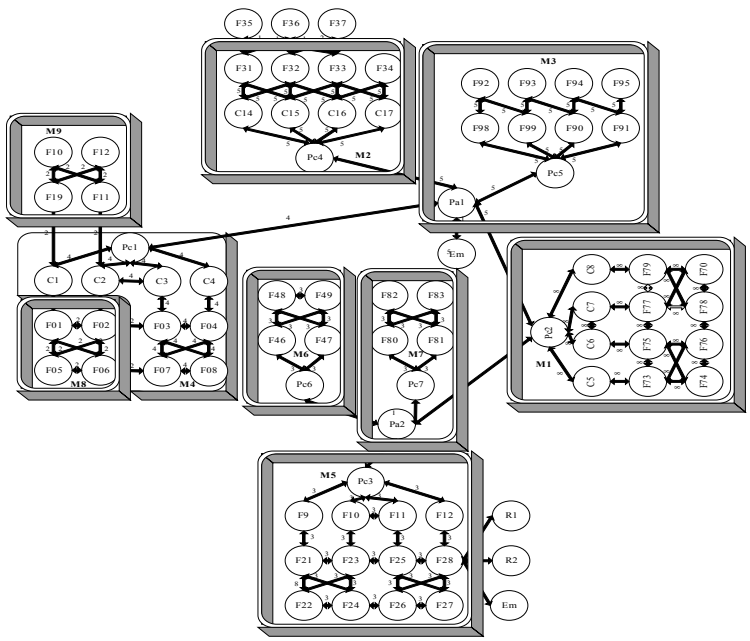


Fig. 15. The resulting modules for nine types of simulator packages.

REFERENCES

1. Anquetil, N. and Lethbridge, T. C. "Experiments with clustering as a software remodularization method," Proceedings. Sixth Working Conference on Reverse Engineering, Atlanta, GA, USA 1999, pp: 235 -255.
2. Baburin, D.E. "Using graph based representations in reengineering," Sixth European Conference on Software Maintenance and Reengineering, Budapest, Hungary 2002, pp: 203 – 206.
3. Baetjer, H. Software as Capital: An Economic Perspective on Software Engineering, IEEE Computer Society Press, New York, 1998.
4. Bell, D., Morrey, I., and Pugh, J. Software Engineering, Prentice Hall , New Jersey, 1986.
5. Briand, L. C., Morasca, S., and Baasili, V. R. "Property-based software engineering measurement," IEEE Transactions on Software Engineering (22:1) 1997, pp: 68-86.
6. Canfora, G., Cimitile, A., Lucia,A.D., and Lucca, G. A.D., "Decomposing legacy programs: a first step towards migrating to client–server platforms," Journal of Systems and Software (54:2), 2000, pp: 99-110
7. Card, D. N., Page, G. T., and McGarry, F. E. "Criteria for software modularization," IEEE Proceedings of the 8th International Conference on Software Engineering, London, England 1985, pp: 372 – 377.
8. Cox, E. "Fuzzy fundamentals," IEEE Spectrum (28:10) 1992, pp: 58-64.
9. DMS Software Reengineering Toolkit, http://www.semdesigns.com/
10. Ebner, G. and Kaindl, H. Tracing all around in reengineering, IEEE Software (19:3) 2002, pp: 70 –77.
11. Eick, S.G., Graves, T. L., Karr, A. F., Mockus, and Schuster, A., P. "Visualizing software changes," IEEE Transactions on Software Engineering (28:4) 2002, pp: 396 –412.
12. Everitt, B. Cluster Analysis, Heineman Educational Books, London, 1974.
13. Fisher, A. S. CASE: Using Software Development Tool 2nd Edition, Wiley Publishers, New York, 1991.
14. Graphviz - Graph Visualization Software, http://www.graphviz.org/About.php
15. Gu, Z., Zhang, Y. F., and Nee, A. Y. C. "Generic from feature recognition and operation selection using connectionist modeling," Journal of Intelligent Manufacturing (6:4) 1995, pp: 266-274.
16. Humphrey, W. S. A Discipline foe Software Engineering, Addison-Wesley, New York, 1995.

17. Jacobson, I. and LindstrÄom, F. "Re-engineering of old system to an object- oriented architecture," Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOP- SLA'91), Phoenix, Arizona, USA October 1991, pp: 340-350
18. Jermaine, C. "Computing program modularizations using the k-cut method," Proceedings. Sixth Working Conference on Reverse Engineering, Atlanta, GA, USA 1999, pp: 224 -234.
19. Kosko, B. Neural networks and Fuzzy Systems, Prentice-Hall, New Jersey, 1992.
20. Kosko, B. Neural networks and Fuzzy Systems, Prentice-Hall, New Jersey, 1986.
21. Myers, G. J. Composite/Structured Design, Van Nostrand Reinhold, New York, 1978.
22. Rosenberg, L. H. and Hyatt, L. E. "Hybrid Re-Engineering," Third IEEE International Symposium on Requirements Engineering, January 1997.
23. Schach, S. R. Software Engineering, Irwin, Illinois, 1993.
24. Scientific Toolworks, Understand for C++ Toolkit, http://www.scitools.com/
25. Serrano, M. A., Carver, D., Montes de Oca, C., "Reengineering legacy systems for distributed environments, " Journal of Systems and Software Volume(64:1), 2002, pp:37-55
26. Sommerville, I. Software Engineering 6th edition, Addison-Wesley, New York, 2001.
27. Tonella, P. "Concept analysis for module restructuring," IEEE Transactions on Software Engineering (27:4) 2001, pp: 351 – 363.
28. Wiggerts, T. A. "Using clustering algorithms in legacy systems remodularization," Proceedings of the Fourth Working Conference on Reverse 29. Engineering, Amsterdam, Netherlands 1997, pp. 33 – 43.
29. Yourdon, E. and Constantine, L. L. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, New Jersey, 1979.