A Business Process Driven Approach for Generating Software Modules

Xulin Zhao, Ying Zou
Dept. of Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada

SUMMARY

Business processes describe business operations of an organization and capture business requirements. Business applications provide automated support for an organization to achieve business objectives. Software modular structure represents the structure of a business application and shows the distribution of functionality to software components. However, mainstream design approaches rely on software architects' craftsmanship to derive software modular structures from business requirements. Such a manual approach is inefficient and often leads to inconsistency between business requirements and business applications. To address this problem, we propose an approach to derive software modular structures from business processes. We use clustering algorithms to analyze dependencies among data and tasks captured in business processes and group the strongly dependent tasks and data into a software component. A case study is conducted to generate software modular structures from a collection of business processes from industrial setting and open source development domain. The experiment results illustrate that our proposed approach can generate meaningful software modular structures with high modularity.

KEYWORDS: Business process; Software architecture generation; Clustering algorithms

1 INTRODUCTION

A business process specifies a collection of tasks for an organization to achieve business objectives. Business processes created by business analysts can be optimized to improve business effectiveness and efficiency of an organization [51]. Typically, a business process is composed of a set of interrelated tasks which are joined together by data flows and control flow constructs. Data flows describe inputs into tasks and outputs generated from tasks. Data items are abstract representations of information flowed among tasks. Control flows specify valid execution orders of tasks. Control flow constructs specify the order (e.g., sequential, alternative, or iterative) of the execution of tasks. For example, the business process for purchasing a product on-line consists of a sequence of tasks, such as *Select product*, *Add to the shopping cart*, and *Validate buyer's credit card*. The data item, *product*, can be generated from the task, *Select product*. Business applications automate business processes to assist business users performing tasks. In this ever changing business environment, business processes are continuously customized to meet the requirements of an organization. Business applications are also modified to add new functional features without referencing to the business processes. In today's reality, the key challenge is to maintain the consistency between business requirements and business applications. Industrial reports [1] indicate that over 50% business applications fail to address their business requirements.

Software modular structure is widely used to bridge the gap between business requirements and business applications. Software modular structure refers to the logical view [44] of software architecture and represents the structure of a business application using software components, the interactions among software components (i.e., connectors), and the constraints on the components and connectors. More specifically, a component captures a particular functionality. Connectors define control and data transitions among components. Constraints specify how components and connectors are combined and the properties of components and connectors. Mainstream design approaches [2][3][4] rely on software architects' craftsmanship to create components and connectors using their past experience. However, real-world, large scale business applications need to satisfy hundreds of business requirements that contain numerous intrinsic dependencies [5]. Quite often, a manual design approach is inefficient and leads to inconsistency between business requirements and business applications.

To facilitate the alignment of business requirements with business applications, we propose an approach that automatically generates software modular structures from business processes. Our generation approach consists of two major steps: 1) derive software components from business processes to fulfill functional requirements; and 2) apply software architectural styles and design patterns to address quality requirements. In this paper, we focus on the first step and present our approach for deriving software components from business processes. The derived components are top-level abstraction that describes the initial decomposition of the functionality of a business application. Moreover, modularity is achieved during the construction of software components and has impact on other quality attributes, such as understandability, maintainability, and testability [52][53][55]. We strive to generate software components with high modularity. However, business requirements are embedded in numerous tasks in business processes. The functionality of a task is simply described as task names using short descriptive terms (e.g., *Create a customer order*). In our work, the challenge lies in identifying major functionalities embedded in business processes and distributing them into software components. Instead of manually understanding the meaning of each task name, we identify the data and control dependencies among tasks by analyzing various entities (e.g., tasks and data items) captured in business processes. We apply clustering algorithms [6] to automatically group functionally similar tasks and distribute functionalities to components. We also provide a technique to identify interactions among components by analyzing the transitions among tasks distributed in different components. This paper extends our earlier work published in the 10th International Conference on Quality Software (QSIC 2010) [7]. We enhance our earlier case study in the following two aspects:

1) conducted a stability study to assess the persistence of the generated software modular structures when the business processes are slightly evolved to address the changing requirements;
2) evaluated the effectiveness of our proposed approach using the business processes recovered from another large scale business system (i.e., Opentaps [28]).

The rest of this paper is organized as follows. Section 2 gives an overview of clustering algorithms. Section 3 discusses the overall steps for generating software modular structures. Section 4 presents the techniques for identifying software components and their interactions from business processes. Section 5 evaluates our

proposed approach through a case study. Section 6 reviews the related work. Finally, Section 7 concludes this paper and proposes the future work.

2 OVERVIEW OF CLUSTERING ALGORITHMS

Clustering algorithms group entities with strong dependencies to form clusters. Similarity measurements evaluate the strength of dependencies between entities by assessing the number of common features or connections between entities. For example, the Ellenberg metric [8] evaluates the degree of similarity between components by calculating the percentage of common features shared by components, such as data members, previous components, and subsequent components.

Partitional algorithms [9] and hierarchical algorithms [10][11][12][13][14] are two commonly used clustering algorithms to cluster entities using similarity measurements. More specifically, partitional algorithms define heuristics to optimize a set of initial clusters which can be a set of randomly grouped entities or the result of other clustering algorithms. For example, Mancoridis et al. [15] generate initial clusters by randomly grouping a set of entities, and then apply hill climbing algorithms and genetic algorithms to optimize the initial clusters using the modularization quality (MQ) metric. The MQ metric measures the cohesion and coupling of software components by evaluating intra-connectivity within components and inter-connectivity among components. The definition of MQ is shown in formula (1). In general, the value of the MQ metric is bounded between -1 and 1. -1 means that a software system has no cohesion and 1 indicates that the software system has no coupling. The extreme values are rarely achieved in practice. The exact range of a MQ value is determined by the intrinsic dependencies within a software system. If components have strong dependencies to each other, the MQ value tends to close to -1. If components can be divided into multiple independent groups, the MQ value tends to close to 1. The MQ is used to access the overall effect of the cohesion within a software component and the coupling among the software components.

$$MQ = \begin{cases} \sum_{i=1}^{k} A_i - \frac{\sum_{i,j=1}^{k} E_{i,j}}{k(k-1)/2} & k > 1 \\ A_1 & k = 1 \end{cases} \qquad A_i = \frac{\mu_i}{N_i^2} \qquad E_{i,j} = \begin{cases} 0 & i = j \\ \frac{\varepsilon_{i,j}}{2N_i N_j} & i \neq j \end{cases} \tag{1}$$

k is the number of clusters. A_i assesses to intra-connectivity and $E_{i,j}$ evaluates inter-connectivity. μ_i is the sum of connections between entities within the cluster C_i. $\varepsilon_{i,j}$ is the sum of connections between entities in the cluster C_i and entities in the cluster C_j. N_i and N_j are the number of entities in the cluster C_i and the cluster C_j, respectively.

Agglomerative algorithms and divisive algorithms are hierarchical algorithms which form a hierarchy of clusters. Agglomerative algorithms are bottom-up approaches that generate clusters by grouping entities in the lowest level of the granularity and moving up to coarser grained entities in a stepwise fashion. Divisive algorithms are top-down approaches that produce clusters by gradually dividing the coarsest grained entities into more fine grained entities. Using an agglomerative algorithm, the most similar pair of entities is selected to form a new cluster.

When more than two entities have the same similarity, the algorithm makes arbitrary decisions by randomly merging two entities. However, arbitrary decisions are harmful to clustering quality and should be avoided in the clustering process [16]. The weighted

combined algorithm (WCA) [8] is used to reduce information loss and decrease the chance for entities to have identical similarity values. A study [12] shows that clustering results of WCA are more consistent with expert decompositions than other hierarchical algorithms. Therefore, we use WCA to produce software modular structures in conformance with those designed by software architects.

3 AN APPROACH FOR GENERATING SOFTWARE MODULAR STRUCTURES FROM BUSINESS PROCESSES

Figure 1 gives an overview of our approach that generates software modular structures from business processes. The generation process consists of three major steps: 1) business process analysis which analyzes business processes to extract the artifacts relevant to tasks and their dependencies; 2) system decomposition which breaks the functional requirements specified in business processes into a collection of more specific functionalities implemented by software components; and 3) architecture representation which represents the generated software modular structures in a format used by software architects for further improvement.

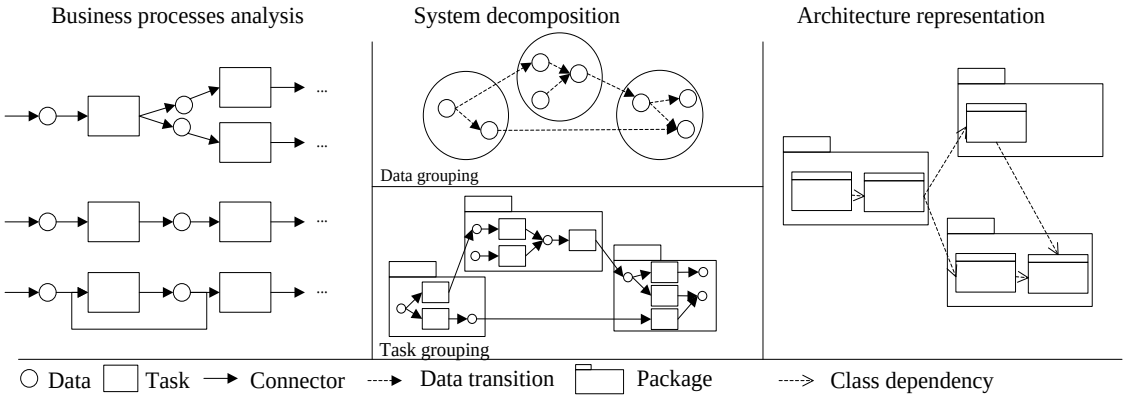

Figure 1: Business process driven software modular structure generation

3.1 Business process analysis

Business analysts specify business processes using graphical notations in a business process modeling (BPM) tool, such as IBM WebSphere Business Modeler (WBM) [17]. A business process can be stored as documents in proprietary formats. BPM languages, such as integrated definition methods (IDEF) [18], business process modeling notation (BPMN) [19], and business process execution language (BPEL) [20], define standard notations to represent entities in business processes. To provide a generic solution to handle business processes described in different languages, we create a meta-model to capture commonality among these BPM languages as shown in Figure 2. Figure 3 illustrates an example business process that is used to describe the typical steps for conducting on-line product purchasing in e-commerce applications. In a business process, a task can be performed by a particular role, such as a customer or a bank teller (shown in Figure 3). A sub-process is a group of tasks that can be reused in different business

processes. A data item contains information required for performing a task or captures the output from a task. For example shown in Figure 3, the data item, *product_info*, represents the structure for describing the product information, and the data item, *product* specifies the structure for representing the searching result. Tasks in a sequence are executed one after another. Loops define a set of tasks to be repeated multiple times. Alternative allows one execution path to be selected among multiple alterative execution paths. Parallels describe multiple execution paths to be executed simultaneously. We develop a parser to extract information from business process specifications in XML format.

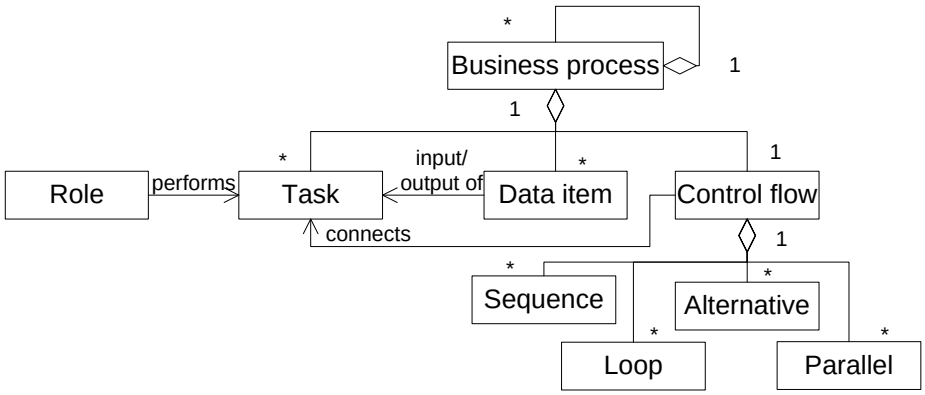

Figure 2: The meta-model for business processes

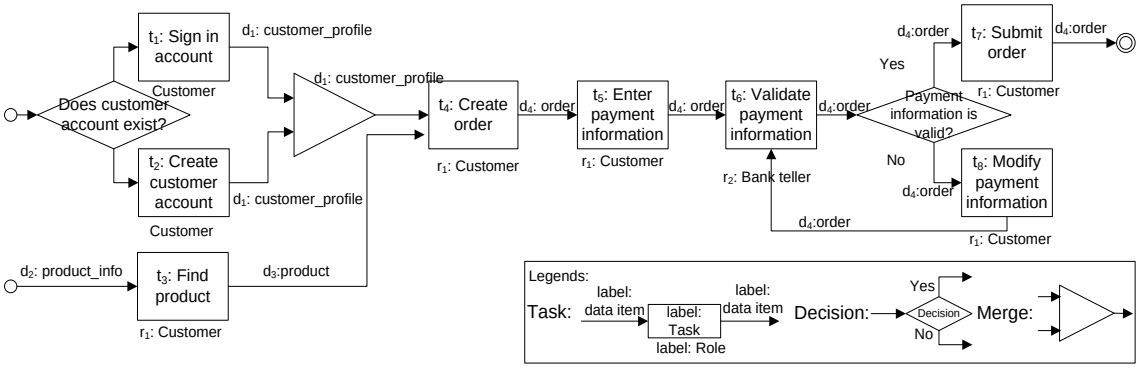

Figure 3: The *p_1:Purchase product* business process

3.2 System decomposition

In general, business requirements can be decomposed into different software components in three ways [21][22]: functional decomposition, data oriented decomposition, and object oriented design. More specifically, functional decomposition recursively refines high-level functional requirements into a collection of more concrete functional requirements. Eventually, each software component implements one concrete functional requirement. Data oriented decomposition identifies a set of essential data structures from the requirements. Each software component is intended to implement one data structure. Object oriented design combines both approaches by forming packages that are

composed of data structures and their associated functions. Each package corresponds to one software component.

In a business process specification, functionalities are reflected in tasks and data structures are well-specified in the data items flowing between tasks. To fully use the information available in the business processes, we use the object oriented design approach to create software components. The data items and tasks specified in business processes represent the lowest level details. One type of data items can be used to form other types of data items. However, it is infeasible to directly map a task or its related data items into a software component. This results in a large amount of fine grained components and makes the software modular structure difficult to understand and use. Therefore, our work focuses on clustering data items to form data structures. Each data structure is composed of multiple closely related data items in business processes. Furthermore, we group tasks to describe operations on the identified data structures which collect the input or output data items of tasks.

3.3 Architecture representation

A major purpose for designing software architecture is to support the communication and collaboration among different stakeholders such as end-users, software developers, and system engineers [4]. To achieve this purpose, software architectures should be represented in a way that is easy for different stakeholders to understand and use. In our work, we represent the generated software architectures using the 4+1 view model supported by Unified Modeling Language (UML) and IEEE 1471 (IEEE Recommended Practice for Architectural Description of Software-Intensive Systems) [23]. More specifically, the 4+1 view model consists of five interrelated software architecture views: logical view, process view, development view, physical view, and scenario.

- A logical view, referred to as the software modular structure, describes the distribution of business requirements among components.
- A process view captures the dynamic behaviors of a business application.
- A development view presents the static structure of a business application.
- A physical view defines mappings between components and hardware.
- Scenarios describe how business requirements are fulfilled using components of a business application.

Each of the views can be described by different UML diagrams. For example, we use UML component diagrams to represent logical views and UML deployment diagrams to depict physical views. In the following section, we show our techniques for generating and representing the logical view from business processes. Other views can be produced from the logical view using model transformations [24][25].

4 SYSTEM DECOMPOSITION

In this section, we discuss our approach that first identifies data groups to form data structures of software components and then assigns tasks, which use the data items in data groups, to components to describe functionalities and enhance the modularity of the generated components.

4.1 Grouping data items

To improve the cohesion within a software component, we strive to identify a group of strongly interrelated data items that specify the data structure of a component. To analyze the dependencies among data items, we create a data dependency graph to analyze data flows within business processes. Essentially, a data dependency graph contains a set of nodes and connectors. A node denotes a data item in a business process. A connector represents a transition from an input data item of a task to an output data item of the task. For example shown in Figure 3, the data item, d_2*:product_info* is the input data item of the task, t_3*: Find product*; and the data item, d_3*:product*, is the output of the task. Therefore, a connector is created between data items, d_2 and d_3. Figure 4 illustrates the data dependency graph generated from the example business process.

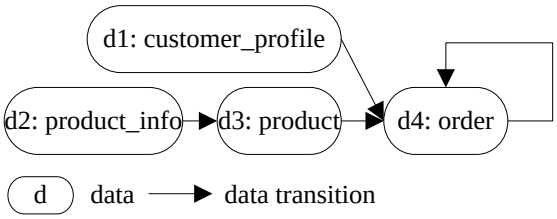

Figure 4: The data dependency graph for p_1*:Purchase product*

DataDependency=<PreviousDataItem,SubsequentDataItem,ContainingBusinessProcess>;
PreviousDataItem =<d_1, d_2, …, d_m>;
SubsequentDataItem =<d_1, d_2, …, d_m>;
ContainingBusinessProcess =<p_1, p_2, …, p_v>;
Subscripts *m* and *v* are the number of data items and business processes respectively.

Figure 5: The format of data dependency vectors

As discussed in Section 2, we apply the WCA algorithm to group data items in a data dependency graph. The WCA algorithm produces a number of data groups at different levels of granularity. To select an optimal grouping result, we use the MQ metric to evaluate the quality of data groups. We aim to achieve high cohesion within a data group and low coupling among data groups. The MQ metric only concerns direct dependencies among data groups. Therefore, we analyze dependencies among data items and their adjacent data items without considering the transitivity of dependencies.

To group data items, we examine three features of data items for describing dependencies of a data item: previous data items, subsequent data items, and containing business processes. These features are organized as a dependency vector (i.e., *DataDepenency* shown in Figure 5), which consists of three data components: *PreviousDataItem*, *SubsequentDataItem,* and *ContainingBusinessProcess*. Each data component in a dependency vector is also defined as a vector. More specifically, *PreviousDataItem* for a current data item is represented as *PreviousDataItem* =<d_1, d_2, …, d_i, …, d_m>, where d_i represents one data item defined in a business process, and *m* is the total number of data items defined in the business processes. d_i is set to 1 when d_i is the incoming data items of the current data item. Otherwise, d_i is set to 0. Similarly, the *SubsequentDataItem* vector marks a data item to 1 if the data item appears as the outgoing data items of the current data item. The *ContainingBusinessProcess* vector, i.e., <p_1, p_2, …, p_i,…, p_v>, represents a collection of business processes that need to be

implemented in a business application. v is the total number of business processes. p_i refers to a business process. It is set to 1 when p_i uses the current data item; otherwise, p_i is set to 0. For example, Table 1 illustrates the values of the vectors for the data dependency graph shown in Figure 4. Each row in the table represents a dependency vector of a data item. For example shown in Figure 4, the data item, *d_1: customer_profile*, has no previous data item, one subsequent item, *d_4: order*, and one containing business process, *p_1: Purchase product*. Therefore, we set d_4 in the *SubsequentDataItem* vector and p_1 in the *ContainingBusinessProcess* vector of d_1 to 1 as illustrated in Table 1.

Table 1: The data dependency table

	PreviousDataItem				**SubsequentDataItem**				**ContainingBusinessProcess**
Data item	**d_1**	**d_2**	**d_3**	**d_4**	**d_1**	**d_2**	**d_3**	**d_4**	**p_1**
d_1	0	0	0	0	0	0	0	1	1
d_2	0	0	0	0	0	0	1	0	1
d_3	0	1	0	0	0	0	0	1	1
d_4	1	0	1	1	0	0	0	1	1

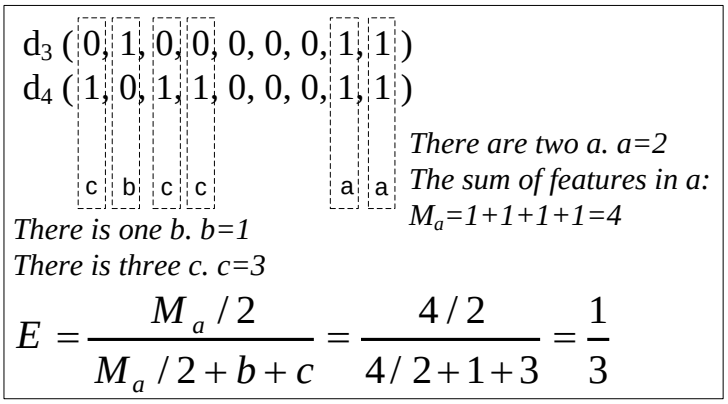

Figure 6: The process for calculating similarity

$$E = \frac{M_a / 2}{M_a / 2 + b + c} \qquad (2)$$

Given two data items D_x and D_y ($x \neq y$), M_a denotes the sum of features that present for both data items. b represents the number of features that presents for D_x and absent for D_y. c represents the number of features that present for D_y and absent for D_x.

Using the data dependency vectors, we evaluate the similarities among any pairs of data items using the Ellenberg metric [8]. The definition of the Ellenberg metric is shown in formula (2). The metric evaluates the similarities of two entities by analyzing their common features. In our work, the entities to be evaluated are data items in business processes. Features of data items are described using data dependency vectors as shown in Figure 5. Both data items can have the common features such as the common previous data items, the same subsequent data items or the same business process that contains them. The more common features the two data items present, the more similar they are. We demonstrate the calculation of the similarity between data items, d_3 and d_4, in Figure

6. Data items, d_3 and d_4, are described by two data dependency vectors whose values are listed in Table 1. The similarities among any pair of data items in the example business process shown in Figure 3 are listed in Table 2.

Table 2: The data similarity table

Data item	**d_1**	**d_2**	**d_3**	**d_4**
d_1	-	1/3	2/3	2/5
d_2	-	-	1/4	1/6
d_3	-	-	-	1/3
d_4	-	-	-	-

Using the similarities of any pairs of data items, we iteratively cluster data items in the five steps listed in Figure 7. The time complexity of the WCA algorithm is $O(n^2)$. The complexity of the algorithm is caused by calculating the similarity values between pairs of data items.

1) Initialize a data item as a data group;
2) Merge the two data groups that have the highest similarity value to form a new data group. For example, we choose d_1 and d_3 to create the data group, *DataGroup <1>*, shown in Figure 9 (a);
3) Calculate the features of the newly formed data group using formula (3) [8]. For example, we calculate features of the merged data group, *DataGroup <1>*, using the data dependency vectors of data items, d_1 and d_3, as illustrated in Figure 8. For a given feature in the merged data group, *DataGroup<1>*, we calculate the sum of the corresponding feature values in the data items, d_1 and d_3. For example, the second features of d_1 and d_3 are 0 and 1, respectively. Therefore, the sum of the second feature is 0+1=1. We also normalize feature values with the total number of data items in the newly formed data group. The merged data group, *DataGroup <1>* contains two data items (i.e., d_1 and d_3). Hence, we divide the second feature value of *DataGroup<1>* by 2 and use ½ as the normalized feature value in the data dependency vector of *DataGroup<1>*;
4) Calculate similarities between the newly formed data group and other data groups using formula (2); and
5) Repeat steps 2) to 4) until only one data group is left.

Figure 7: The data grouping process

$$f_i = \frac{f_{i1} + f_{i2}}{n_1 + n_2} \qquad (3)$$

f_i, f_{i1} and f_{i2} refer to the ith feature of the newly formed data group and its two constituent data groups respectively. n_1 and n_2 are the number of data items in the two constituent data groups.

For example, Figure 9 shows three iterations for grouping data items defined in the example business process (shown in Figure 3). One data group is generated in each iteration. We organize the generated data groups in a hierarchy shown in Figure 10. Leaves of the hierarchy are data items defined in the business process. Internal nodes represent data groups created in each iteration by clustering the data items and the data

groups in the lower levels of the hierarchy. The levels in the hierarchy denote the order of the data items being merged in the clustering process. For example, the data group, *DataGroup<1>* (i.e., *{customer_profile, product}*) in level 1 is generated in the first iteration.

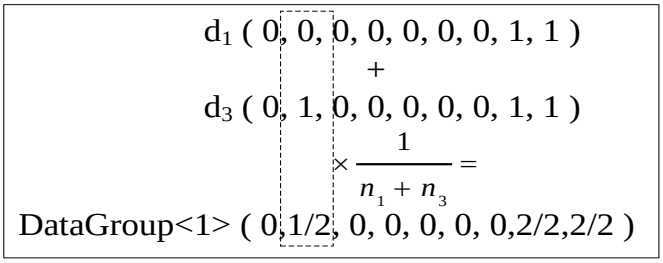

Figure 8: The process for calculating the feature for *DataGroup <1>*

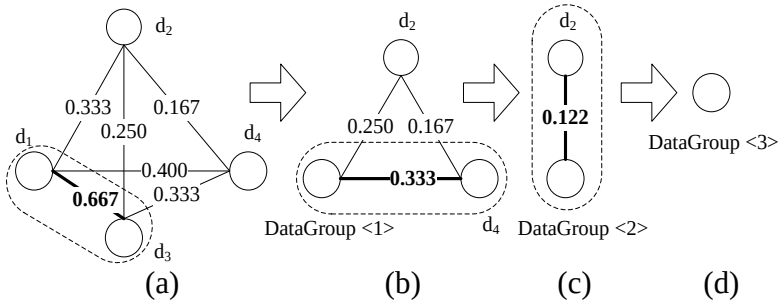

Figure 9: The data grouping process for the example business process

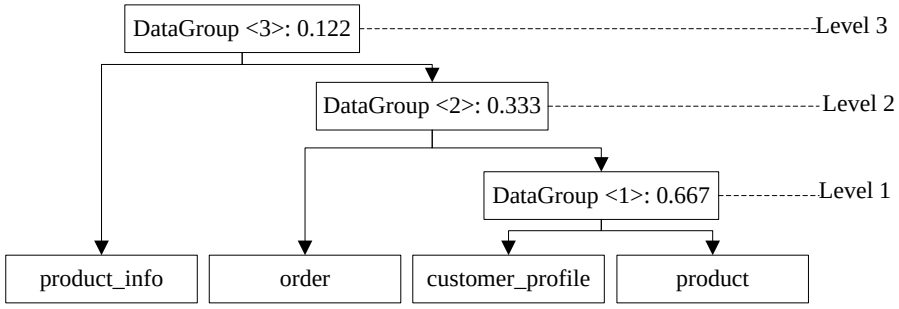

Figure 10: The result hierarchy of data groups

We traverse the hierarchy of data groups at each distinct level to produce a collection of grouping results. For example shown in Figure 10, we can produce three data groups at level 2. The grouping result is composed of three data groups: *{product_info}*, *{order}*, and *DataGroup <1>*. Furthermore, the data group, *DataGroup <1>*, contains two data items, *customer_profile* and *product*. Therefore, the data grouping result can be also represented as *{product_info}*, *{order}*, and *{customer_profile, product}*. We can obtain three different data grouping results by visiting different levels in the hierarchy of data groups. The group results are listed in Table 3.

A higher similarity value indicates a stronger functional dependency [6][54]. In the clustering algorithm, the most similar data items are grouped first. Data groups in lower levels are composed of data items with higher similarity values. Such data groups are

composed of data items (or data groups) with stronger dependencies and have higher cohesion. Data groups in higher levels are composed of coarser grained data groups with lower similarity values. Therefore, such data groups consist of data items (or sub data groups) with looser dependences. The coupling among data items (or sub data groups) in such data groups are lower.

Data groups in the lowest level contain only one data item. A grouping result that consists of data groups in the lowest level has the highest cohesion. The data group in the highest level (i.e. the root data group) consists of sub data groups that present the lowest coupling. To produce Software modular structures with high modularity, we use the MQ metric defined in formula (1) to assess the modularity of a data group by considering cohesion within a data group and coupling among data groups. We calculate MQ values for all grouping results and choose the one with the highest MQ value to generate the software modular structures.

Table 3: MQ values of the three data grouping results

Hierarchy level	**Grouping result**	**MQ**
1	{product_info}, {order}, {customer_profile}, {product}	-0.358
2	{product_info}, {order}, {customer_profile, product }	-0.219
3	{ product_info}, {order, customer_profile, product}	-0.172

However, the MQ metric uses only the interactions between components to calculate their dependencies, as specified in formula (1). Such a metric is not sufficient to assess the modularity of a component with multiple quality attributes. We extend the definition of MQ metric to evaluate the modularity using data dependency vectors as defined in Figure 5. The intra-dependency within a data group, μ_i, is measured by the sum of similarities between all data items inside the data group. The inter-dependency between two data groups, $\varepsilon_{i,j}$, is measured by the sum of similarities of all data items in the two data groups. The value range of the extended MQ is between -1 and 1. For example, we apply the extended MQ metric to evaluate the modularity of the three data grouping results listed in Table 3. Result MQ values are shown in Table 3. As a result, the data grouping result generated from level 3, *{product_info}, {order, customer_profile, product}*, has the highest MQ value which indicates the best modularity. Hence, we separate data items into two groups as illustrated in Figure 11.

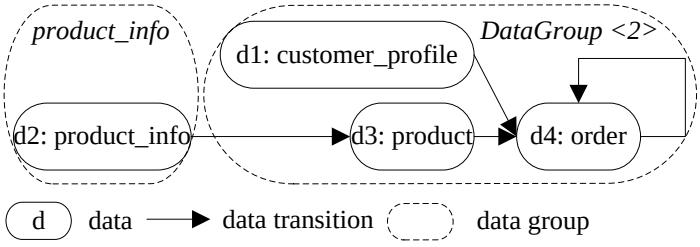

Figure 11: The selected data grouping result

4.2 Associating tasks with data groups

To identify functionalities for components, we assign tasks to data groups using the dependencies between data groups and tasks. We classify tasks into two categories: inner tasks and outer tasks.

Inner tasks depend on data items within a data group. More specifically, the data group contains both input data items and output data items of the task. All information required to perform this type of tasks is located within one data group. Therefore, we group inner tasks with their dependent data groups into one component. For example shown in Figure 11, we can create two components from the data grouping result. One component contains the data item, d_2*: product_info*. The other component contains three data items: d_1*:customer_profile*, d_3*: product*, and d_4*: order*. The task, t_4*: Create order,* takes data items, d_1*: customer_profile* and d_3*: product*, as input and generates data item, d_4*: order*, as output. All these data items belong to the data group, *DataGroup <2>*. Therefore, we merge the task, t_4*: Create order*, to the latter component.

Outer tasks depend on data items distributed in multiple data groups. To assign an outer task to an appropriate data group, we evaluate the dependency strength between a task and the related data groups using the Ellenberg metric defined in formula (2). Similar to measuring the similarity among data items, we create data vectors to describe the features of tasks and data groups. A data vector describes the dependency of a task or a data group on all data items defined in the business processes. Therefore, a data vector is represented as $DataVector = <d_1, d_2, \ldots, d_m>$, where m is the total number of data items in business processes and $d_i\ (1 \le i \le m)$ denotes one data item. We set d_i to 1 if it is depended by a task or included in a data group. Otherwise, we set d_i to 0. For the example shown in Figure 3, the task, t_3*:Find product*, takes the data item, d_2*: product_info*, as its input and generates the data item, d_3*: product*, as its output. Therefore, we set d_2 and d_3 to 1 in the data vector of the task, t_3*: Find product*, illustrated in Figure 12 (a). Similarly, the data group, *product_info*, contains one data item, d_2*: product_info*. Therefore, we set d_2 to 1 in the data vector, *product_info* as illustrated in Figure 12 (a). The calculated Ellenberg values in Figure 12 (b) indicate that the task, t_3*: Find product*, depends more strongly on the data group, *product_info*, than on the data group, *DataGroup<2>*. To achieve high cohesion, we assign the task, t_3*: Find product*, to the component corresponding to the data group, *product_info*.

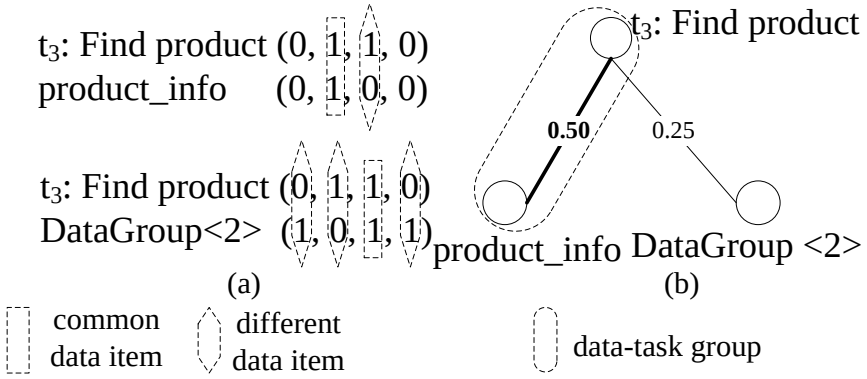

Figure 12: The data vectors and the component selection method

To produce the software modular structure, we further identify connectors between components by analyzing task transitions among components. For example, Figure 13

shows the two components generated from the example business process shown in Figure 2. The first component contains data items in the data group, *DataGroup<2>*, and the associated tasks. The second component is composed of the data item in the data group, *product_info*, and the associated task. The connector between these two components are identified from the transition between the task, *t_3:Find product*, and the task, *t_4:Create order*.

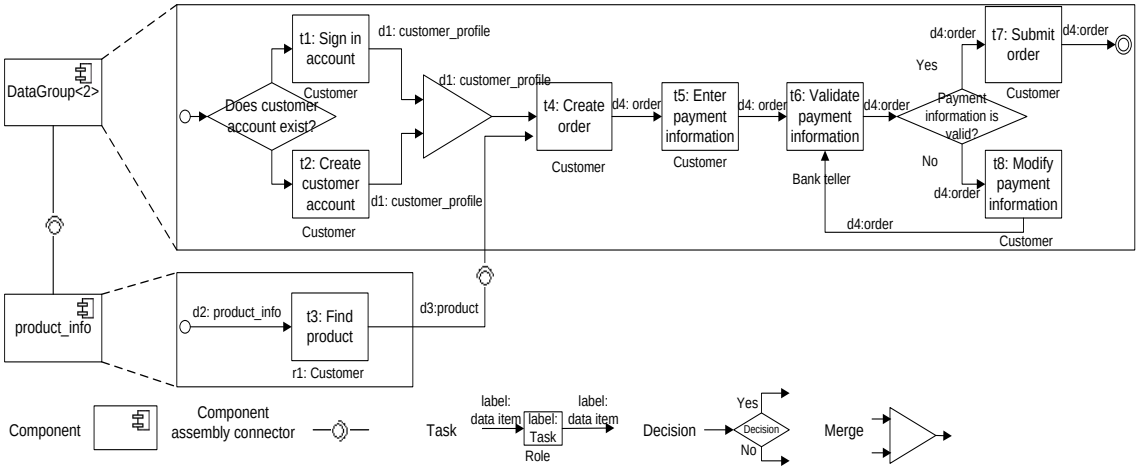

Figure 13: The generated software modular structure

5 CASE STUDY

We conduct a case study to evaluate the effectiveness of our proposed approach in generating software modular structures from business processes.

5.1 Objectives

Our proposed approach uses clustering techniques to generate software modular structures from business processes. A promising software clustering process needs to produce meaningful and consistent software modular structures. The objectives of the case study evaluate the following aspects:

- *Authoritativeness* of the generated software modular structures. It regards the structural similarity between a generated software modular structure and an authoritative software modular structure. One source of authoritative software modular structures is those designed by expert architects. However, descriptions of such software modular structures are often difficult to find. A common source of authoritative software modular structures is to recover as-implemented software modular structures from documentations or existing source code. In our work, we use the as-implemented software modular structures as authoritative designs and compare the generated software modular structures with as-implemented software modular structures to evaluate the authoritativeness of the generated software modular structures.
- *Stability* of our proposed approach. It concerns the persistence of the generated software modular structures when the business processes are evolved to reflect the changing requirements. We want to assess if our proposed approach can produce

persistent software modular structures when the business processes are slightly evolved.

- *Modularity* of the generated software modular structures. Modularity is one of the internal quality attributes which directly or indirectly affect the external quality attributes, such as maintainability and reusability [26]. We aim to produce software modular structures with high modularity which can lead to satisfactory external quality attributes.

To achieve the aforementioned objectives, we conduct a case study in five steps:

1) Generate software modular structures from business processes of subject business systems;
2) recover the as-implemented software modular structures from various sources (e.g., documentations or source code) of subject business systems;
3) compare the generated software modular structures with the as-implemented software modular structures to assess the authoritativeness of our generated software modular structures;
4) analyze the extent to which the generated software modular structures is affected by changes in business processes in order to examine the stability of our proposed approach;
5) evaluate the modularity of the generated software modular structures.

5.2 Subject business systems

We evaluate our proposed approach on two large scale business systems: IBM WebSphere Commerce (WSC) server [27] and Opentaps [28]. The subject business systems are selected as representatives from different development domains: commercial systems and open source systems.

5.2.1 IBM WSC server

IBM WSC server is a commercial platform for building e-commerce web sites and applications. This system implements business processes to support B2B (business-to-business) and B2C (business-to-consumer) transactions. The business processes for the system are available online [27]. These business processes are organized in 5 categories. Table 4 shows descriptions of the functionalities encapsulated in business processes of the 5 categories. Table 5 lists the number of tasks, data items, roles, and business processes in each category of business processes specified in IBM WSC. The business processes are modeled using WebSphere Business Modeler [17] and stored as XML documents. We developed a business process parser to parse the XML documents and extract entities as specified in our meta-model (shown in Figure 2).

Table 4: Functionality of IBM WebSphere Commerce

Category	**Description of functionality**
Marketing	Facilitate marketing campaigns and activities
Merchandise management	Create, load, and manage products in online stores
Order management	Manage state of orders and components of orders
Customer management	Create and manage customer profiles and member resources
Customer service	Assist customer service representatives to provide services to customers

Table 5: Numbers of entities in IBM WebSphere Commerce

Category	# Task	# Data item	# Role	# Business process
Marketing	69	14	3	8
Merchandise management	66	24	6	24
Order management	204	48	14	25
Customer management	17	8	6	7
Customer service	22	13	5	10

Table 6: Functionality of Opentaps

Category	Description of functionality
Accounting	Manage agreements, invoices, fixed assets, and general ledger
Content	Administrate product contents, websites, blogging, and forums
CRM	Offer customer services
Ecommerce	Manage shopping cart
Financial	Handle account receivables and account payables
Manufacturing	Manage bill of materials, production tasks, and work orders
Marketing	Process customer segments and marketing campaigns
Order	Deal with orders, quotes, and customer requests
Party	Provide general party management
Product	Support product and catalog management
Purchasing	Manage supply chain
Warehouse	Track inventory
Work effort	Handle time sheets, events, and tasks

Table 7: Numbers of entities in Opentaps

Category	# Task	# Data item	# Role	# Business process
Accounting	143	47	5	69
Content	126	34	4	53
CRM	88	41	5	33
Ecommerce	10	10	2	8
Financial	52	24	2	18
Manufacturing	34	20	2	21
Marketing	31	9	3	19
Order	78	41	5	22
Party	85	24	5	23
Product	119	65	4	60
Purchasing	19	4	2	14
Warehouse	47	14	3	24
Work effort	49	7	2	21

5.2.2 Opentaps

Opentaps is an open source system for enterprise resource planning (ERP) and customer relationship management (CRM). The system consists of 13 business applications for supporting various organizational business activities, such as customer relationship management, warehouse and inventory management, supply chain management, and financial management. The business processes of Opentaps are not available. In our case study, we automatically recover business processes from the source code of Opentaps using the Business Process Explorer (BPE) developed by Zou et al. [29]. The recovered business processes are specified using the format supported by WebSphere Business Modeler. The recovered business processes maintain the mappings between tasks in the business processes and the corresponding source code. Table 6 summarizes the functionalities of the 13 business applications. Table 7 lists the numbers of entities in each business application.

5.3 Experiment design

In this section, we describe the experiment design for evaluating authoritativeness, stability, and modularity of the generated software modular structures.

5.3.1 Evaluation of authoritativeness of the generated software modular structures

To assess the authoritativeness of the generated software modular structures, we compare our generated software modular structures with the as-implemented software modular structures. We manually recover architectural components to construct the as-implemented software modular structures. As studied by Tzerpos and Holt [30], the directory structure of a software system designates its functional decomposition. We analyze the directory structure of the subject business systems and map each top level directory to an architectural component. Due to the unavailability of the source code of IBM WSC server, we derive the directory structure of IBM WSC server from the documentation [27]. In IBM WSC, the directory structure is reflected in class identifiers which consist of various segments to identify the names of the systems, the packages, and the class names. For example, *com.ibm.commerce.order.commands.OrderCreateCmdImpl* designates a class used in IBM WSC server. *com.ibm.commerce* identifies the system itself. *order* denotes a top level package that captures a major functionality of the system. Hence, we identify *order* as an architectural component. *commands* designates a sub-package of the *order* package and can be mapped to a sub-component of the *order* component. *OrderCreateCmdImpl* is the name of a class within the *order* component. The directory structure of Opentaps is directly obtained from the source code [28].

In our generated software modular structures, functionalities of components are described by tasks in business processes. To compare our generated software modular structures with the as-implemented software modular structures, we need to describe functionalities of as-implemented components in the same way. Therefore, we further analyze relationships among directories, classes, and tasks to identify tasks captured in as-implemented software components. As aforementioned, IBM WSC server is a commercial platform without available source code. By consulting the architects from IBM WSC server, each task in a business process is implemented by a command class in the code. We use the naming convention of classes to recover the corresponding tasks in the business processes. For the example of a class, *OrderCreateCmdImpl*, *CmdImpl* indicates

that the class is the implementation of a task command. *Create Order* corresponds to the name of the task, *Create Order*. Opentaps is an open source software system with available source code. We use the BPE tool [29] to automatically recover tasks from source code of Opentaps.

$$MoJoFM\ (A,B) = (1 - \frac{mno(A,B)}{\max(mno(\forall A,B))}) \times 100\ \% \quad (4)$$

A and B are two software architecture designs. mno(A,B) represents the number of *move* and *join* operations required for changing A to B. mno($\forall$ A, B) denotes the maximum distance to the partition B.

To evaluate the authoritativeness of the generated software modular structures, we use the MoJoFM metric [31] to compare the generated software modular structures and the as-implemented software modular structures. The definition of the MoJoFM metric is shown in formula (4). The metric counts the number of operations required to make one software modular structure the same as another. Large numbers of operations indicate high refactoring effort and reveal a low structural similarity between the two versions of software modular structures. Two types of operations are used to change software modular structures: move and join. A move operation moves one task from a component to another. A join operation merges two components to form a new component.

During the calculation of the MoJoFM metric, we consider a component as a set of tasks and evaluate the similarity of two components by comparing tasks within them. For example, suppose we have three components, C_1={t_a, t_b, t_c}, C_2={t_d, t_e}, and C_1'={t_a, t_b, t_d}, we can know that C_1 and C_1' are similar since they have two tasks, t_a and t_b, in common. C_1 and C_2 are not similar since tasks in the two components are totally different. Suppose we have two software modular structures that capture the same set of tasks {t_a, t_b, t_c, t_d, t_e}. One software modular structure, SA_1, consists of two components and is represented as {{t_a, t_b, t_c}, {t_d, t_e}}. The other software modular structure, SA_2, consists of three components and is represented as {{t_a, t_b, t_e}, {t_c}, {t_d}}. To evaluate the similarity between SA_1 and SA_2, we use move operations and join operations to make SA_2 identical to SA_1 and count the number of operations used. More specifically, we make SA_2 identical to SA_1 in the following three steps:

1) *move* the task t_e in the component {t_a, t_b, t_e} to the component {t_c}. As result, SA_2 becomes {{t_a, t_b}, {t_c, t_e}, {t_d}};
2) *move* the task t_c in the component {t_c, t_e} in the result of step 1 to the component {t_a, t_b}. Consequently, SA_2 becomes {{t_a, t_b, t_c}, {t_e}, {t_d}}
3) *join* the two component, {t_e} and {t_d}, in step 2. Correspondingly, SA_2 becomes {{t_a, t_b, t_c}, {t_d, t_e}} which is exactly the same as SA_1.

We record the numbers of operations used (i.e., 2 move operations and 1 join operations) and use the numbers to calculate the MoJoFM value using formula (4). From the calculation process, we can know that the calculation of the MoJoFM metric is independent from names of components. The value of MoJoFM is bounded between 0 and 100%. 0 means that two software modular structures are completely different and 100% indicates that two software modular structures are identical.

5.3.2 Evaluation of stability of the generated software modular structures

In the business domain, business processes are subject to changes due to new business requirements or business process optimization activities (e.g., removing bottlenecks and inconsistency) in the business process design. The minor changes, such as task addition,

removal or merging two separated tasks into one, do not have significant impact on the software modular structures. In other words, the software modular structures need to be persistent after the minor changes applied to business processes. Therefore, we examine the stability of the software modular structures by comparing the software modular structures generated from the initial business processes with the ones generated from the changed business processes.

To conduct the stability study, we introduce changes to business processes. It is challenging to enumerate all possible changes in business processes. Instead, we use a random process to introduce disturbances to business processes. To design such a random process, we need to know the types of potential changes and the appropriate amount of the changes for each type. As studied by Tzerpos and Holt [32], a clustering result (i.e., the generated software modular structure) can be affected by four types of changes: task addition, task deletion, connector (i.e., the control and data transition between tasks) addition, and connector deletion. Other changes, such as modification of task names, and modification of data items have no impact on the generated software modular structure, since our clustering process is independent of task names and the internal structures of data items. Moreover, the four types of changes happen with different frequencies. To ensure that the random process correctly reflects practical changes, we assign different weights (shown in Table 8) to the four types of changes as suggested by Tzerpos and Holt [32].

$$Difference\ (A,B) = 1 - MoJoFM\ (A,B) = \frac{mno\ (A,B)}{\max(\ mno\ (\forall A,B))} \times 100\ \% \qquad (5)$$

A and B are two software architecture designs. mno(A,B) represents the number of *move* and *join* operations required for changing A to B. mno($\forall$ A, B) denotes the maximum distance to the partition B.

Table 8: A Summary of changes in the stability study

Change type	**Weight**	**# of Changes of IBM WSC (Total # of tasks=378)**	**# of Changes of Opentaps (Total # of tasks=881)**
Task addition	50%	$378 \times 1\% \times 50\% \approx 2$	$881 \times 1\% \times 50\% \approx 5$
Task deletion	25%	$378 \times 1\% \times 25\% \approx 1$	$881 \times 1\% \times 25\% \approx 3$
Connector addition	15%	$378 \times 1\% \times 15\% \approx 1$	$881 \times 1\% \times 15\% \approx 1$
Connector deletion	10%	$378 \times 1\% \times 10\% \approx 1$	$881 \times 1\% \times 10\% \approx 1$
Total	100%	5	10

In the stability study, we consider only slight changes in business processes. Similar to the work by Tzerpos and Holt [32], we use 1% of changes in the functionally as a magnitude of slight changes. The tasks in business processes capture the functionality. Therefore, the total functionality is evaluated using the total number of tasks in business processes. If our approach for generating software modular structures is stable, then 1% of functionality changes in the business processes result in no more than 1% differences in the generated software modular structure [32]. To compare software modular structures generated before and after the changes, we use the MoJoFM metric to compare the structural differences between both software modular structures (as illustrated in formula (5)). Table 8 summarizes the weights for each type of changes and the number of changes made to the business processes. Changes in different parts of business processes can affect

the result software modular structure differently. Hence, we repeated 1000 times of the study. Each time, we randomly introduce different changes.

Dramatic changes, such as removal of entire business processes, are not considered in our case study. When dramatic changes are made, a new software modular structure would be needed to accommodate the changes. In this case, we concern the authoritativeness of the generated software modular structure, rather than the persistency of the generated software modular structure.

5.3.3 Evaluation of modularity, cohesion, and coupling of the generated software modular structures

We use the MQ metric as defined in formula (1) to assess the modularity of software modular structures. More specifically, we use the intra-connectivity of tasks within a component to evaluate the cohesion of components. The inter-connectivity of tasks among different components is used to assess the coupling of components.

5.4 Comparison of as-implemented and generated software modular structures

5.4.1 Software modular structures of IBM WSC

To identify the as-implemented software modular structure, we study the documentation for the IBM WSC and the APIs provided by IBM WSC server. The documentation describes the functional decomposition of the entire system. Figure 14 illustrates the subsystems in IBM WSC and their relations. We identify components for each subsystem by studying the documentation for the packages and the classes within a package. Figure 15 shows a more detailed software modular structure by grouping functionally similar packages into a component within a subsystem. As shown in Figure 15, each component captures a primary functionality of IBM WSC server. The name of each component is summarized by studying the functionality of software packages. For example, the *Tools* subsystem provides a set of utility operations to find data items from the databases or create input data items. We have developed a prototype tool to generate software modular structure using business processes of IBM WSC. The generated software modular structure is shown in Figure 16.

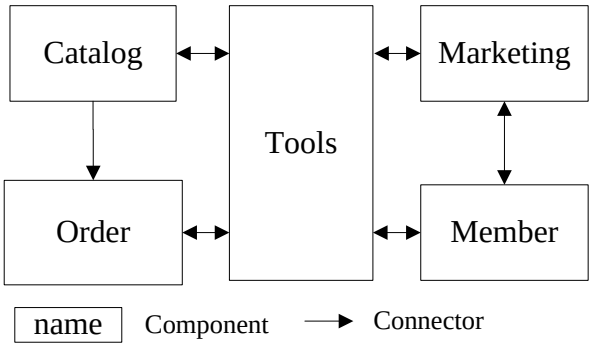

Figure 14: The as-implemented software modular structure of IBM WSC

The name of a generated component is identified by studying the description of tasks specified in business processes. An underlined label in Figure 15 and Figure 16 illustrates the corresponding subsystem which contains the functionally similar components. We assign the same name to the similar components in both software modular structures. For example, both Figure 15 and Figure 16 have a component named, *Order*. This indicates

that the two components capture the same primary functionality. However, this does not indicate that tasks contained in both components are identical. We attach a prime sign on the name of each generated component to differentiate generated components from as-implemented components. For example, order processing is closely related to payment handling in IBM WSC, therefore, we group payment handling tasks and order processing tasks into one component, *Order'* in the generated software modular structure to improve cohesion and reduce coupling in the generated software modular structure. Moreover, the differences in the two software modular structures can be observed by comparing connectors in Figure 15 and Figure 16. The generated software modular structure shown in Figure 16 contains fewer connectors than the as-implemented Software modular structure shown in Figure 15. Hence, we envision that our generated software modular structure present lower coupling than the as-implemented software modular structure.

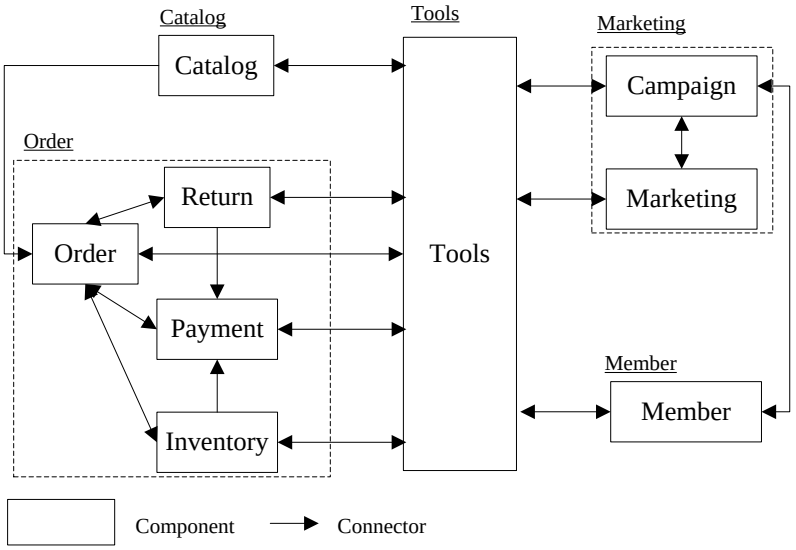

Figure 15: An in-depth view of the as-implemented software modular structure of IBM WSC

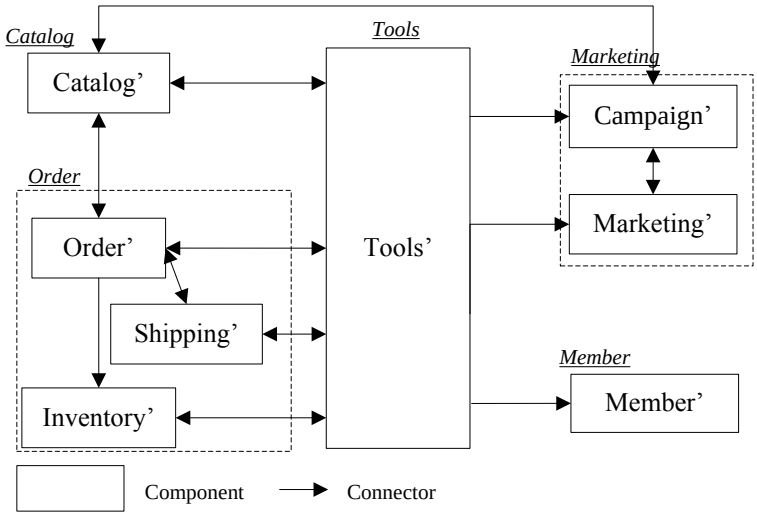

Figure 16: The generated software modular structure of IBM WSC

5.4.2 Software modular structures of Opentaps

We use our prototype tool to generate a software modular structure design from the recovered business processes of Opentaps. Figure 17 shows the as-implemented software modular structure of Opentaps. Connectors between components are identified from transitions among tasks. Figure 18 illustrates the generated software modular structure. Names of components are determined by studying the source code of packages contained in the architectural components. Comparing Figure 17 and Figure 18, we find that the generated software modular structure contains fewer components than the as-implemented software modular structure. Two pairs of components, *(Party, CRM)* and *(Ecommerce, Order)*, in the as-implemented software modular structure are merged in the generated software modular structure. A new component, *Invoice*, is created from tasks related to invoice processing within the as-implemented component, *Accounting*. Tasks related to shipment are grouped to form a new component, *Shipment*. Tasks in the component, *Purchasing*, in the as-implemented software modular structure are distributed to the *Catalog* component and the *Shipment* component in the generated software modular structure. The differences in the distribution of the components in both software modular structures are resulted from the modularity evaluation. In the generated software modular structure, we aim to group the highly cohesive functionality within the same component.

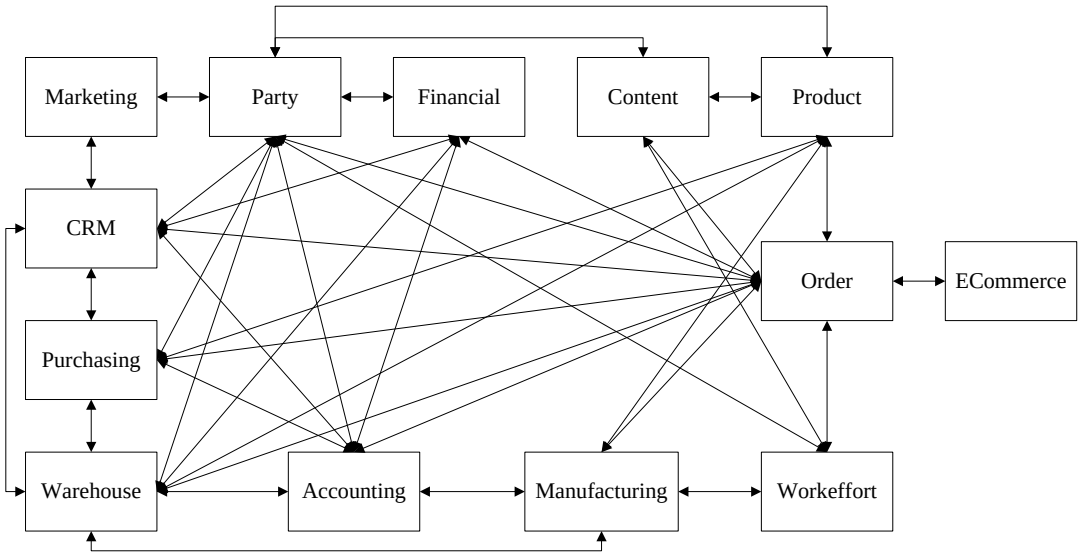

Figure 17: The as-implemented software modular structure of Opentaps

5.5 Analysis of experiment results

5.5.1 Result of authoritativeness evaluation of the generated software modular structure

We calculate MoJoFM values to evaluate the structural similarity between as-implemented software modular structures and generated software modular structures. Table 9 lists the results of structural similarity of both software modular structures for each subject system. As shown in Table 9, the MoJoFM value is 72% for IBM WSC. It indicates that 28% of tasks in the as-implemented software modular structure are moved to form the generated software modular structure. The MoJoFM value is 76% for Opentaps. This value shows that 24% of tasks in the as-implemented software modular

structure are moved to produce the generated software modular structure. As discussed by Wen and Tzerpos [31], such values are desirable and show that the components generated by our proposed approach are consistent with the as-implemented software modular structures.

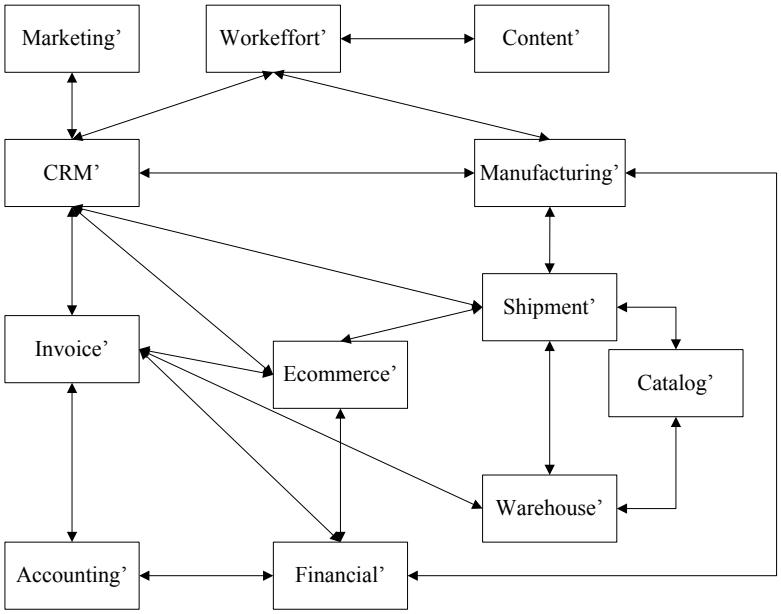

Figure 18: The generated software modular structure of Opentaps

Table 9: Results of authoritativeness and stability study

Application	Authoritativeness	Stability
IBM WSC	72%	96%
Opentaps	76%	93%

Table 10: Results of modularity study

Quality Attribute	IBM WSC		Opentaps	
	As-implemented	Generated	As-implemented	Generated
MQ	0.025	0.042	0.081	0.111
Cohesion	0.034	0.048	0.084	0.113
Coupling	0.009	0.006	0.003	0.002

5.5.2 Result of stability evaluation of the generated software modular structures

The stability of the generated software modular structures is evaluated and the results are listed in Table 9. As reported in [32], the clustering process can generate persistent software modular structure when at least 80% software modular structures are structurally similar. In our study, the stability value is 96% and 93% among 1000 times of random changes for IBM WSC and Opentaps, respectively. The results indicate that our approach can produce persistent software modular structures given 1% of functionality changes in business processes.

5.5.3 Result of modularity evaluation of the generated software modular structures

Table 10 lists the results of modularity evaluation for both subject systems. It is desirable to achieve high MQ values, high cohesion and low coupling values in software modular structures. As illustrated in Table 10, the generated software modular structures for both systems have more desirable values in three metrics. Differences between the two software modular structures of IBM WSC are introduced from the decomposition of components. As discussed in Section 5.4.1, tasks are re-distributed to the generated software components to improve cohesion and reduce coupling. This results in the improvement of the modularity of the generated software modular structure. In Opentaps, the improvement of the modularity results from merging the closely related tasks and components (as discussed in Section 5.4.2). More specifically, two pairs of closely related components are merged. Tasks in one component are re-distributed based on the strength of dependencies among them. Such changes increases cohesion of components and reduces coupling among components. With high cohesion and low coupling, components can be easily understood, implemented, and changed with little effect on other components. Therefore, increasing cohesion and decreasing coupling can reduce the development cost for applications and improve external quality attributes, such as maintainability, reusability, and flexibility [33][34].

5.6 Threats to validity

We discuss the influences that might threat the validity of our case study. We assess three types of threats to validity: construct validity, internal validity, and external validity.

Construct validity concerns whether the selected metrics are appropriate for the purpose of our case study [35]. A common technique to evaluating a clustering algorithm is to compare its output with an authoritative clustering result [31]. MoJoFM is a metric specifically designed to compare the structural similarity of two clustering results. Cohesion and coupling are the major concerns in the modularization stage of a software system. MQ is used to evaluate the modularity of software modular structures. However, both metrics are relative measures and their effectiveness depends heavily on the quality of the benchmark software modular structure. Although the success of IBM WSC and Opentaps ensures that their software modular structures are well-designed, we cannot assure that the software modular structures are optimal. In the future, we plan to further evaluate the effectiveness of our proposed approach by comparing software modular structures generated by our approach with those created by other approaches [48][49]. Moreover, we assume that the possible changes in business processes are less than 1%. The 1% magnitude is suggested by Tzerpos and Holt [32] based on their experience. We plan to study more changes and figure out the threshold of the functionality changes that would affect the stability.

Threats to **internal validity** are factors that can affect the accuracy of our observations [35]. We predict interactions among components from transitions among tasks in our case study. However, the mapping between task transitions and component interactions may not necessarily one to one mappings. In the implementation stage, one task transition might be mapped to multiple interactions among components (e.g., a handshake process in network protocols). Moreover, one component interaction can be implemented as multiple interactions among tasks. In the future, we aim to further assess the effectiveness of our approach by clarifying the coherence between task transitions and component interactions.

Threats to **external validity** are factors that can hinder the generalization of our conclusion [35]. We conduct our experiment on IBM WSC and Opentaps. The success of

both systems can ensure that they can be considered as representatives from the closed source or open source domains. We envision that our approach can be customized to generate software modular structure for other business systems. We plan to further assess the generality of our approach by investigating the feasibility of generating software modular structures from the business processes specified for more domains.

6 RELATED WORK

6.1 Bridge requirements and software architecture

A variety of approaches have been presented to bridge the gap between requirements and software architecture. The research efforts focus on introducing synchronization steps into the software architecture design process. For example, Nuseibeh [36] adapts the spiral life cycle model to incrementally develop software architecture and synchronize software architecture with requirements. Grunbacher et al. [37] enhance Nuseibeh's approach by providing an intermediate model to assist the synchronization of software architecture and requirements. However, little guidance is available to assist software architects to analyze requirements in the aforementioned approaches. Our approach can provide an initial architecture for software architects to refine it.

Other approaches are proposed to guide software architects to derive software architecture from requirements. Jackson [38] uses problem frames to record typical system partitions in different domains. The created problem frames are then used to guide software architects to construct software architecture from requirements. Zhang et al. [39] use feature graphs to represent dependencies among requirements and group requirements with common features to create architectural components. The major problem of these approaches is that the creation of problem frames and feature graphs is time-consuming and labor-intensive. In our work, we analyze business processes created by business analysts and automatically group functionally similar tasks to produce architectural components. Our proposed approach reduces the cost of building software architecture and improves the consistency between requirements and software architecture.

6.2 Software architecture styles and frameworks

Software architecture styles provide reusable solutions to build software architecture. A software architecture style defines the vocabulary of components and connectors for describing software architecture of a family of software systems. Garlan and Shaw [40] and Buschmann [41] summarized common paradigms in software architecture design and presented a number of software architecture styles. Klein et al. [42] correlate software architectural styles to quality models to assist software architects to compare and choose software architectural styles. Such a design paradigm provides significant support for software architects to address quality requirements. However, the architectural styles provide little support to help software architects decompose functionalities in requirements. Our proposed approach automatically generates components and connectors from business processes. The generated architecture can be further refactored to conform to software architectural styles.

Software architecture frameworks provide structured and systematic approaches for designing software architecture. Software architecture frameworks use views to represent

different perspectives of a software architecture design. Moreover, software architecture frameworks provide comprehensive guidance to assist software architects to construct, analyze, and verify software architecture designs. Typical software architecture frameworks include Zachman Framework for Enterprise Architecture (ZF) [43], 4+1 View Model of Architecture [44], and The Open Group Architecture Framework (TOGAF) [45]. These frameworks provide solutions to a wide spectrum of software architecture design problems and can be tailored to specific needs in different domains. However, software architecture design process is still time-consuming and inaccurate since software architects need to manually derive architectural components from requirements. Our proposed approach reduces the cost of software architecture design process by automatically generating architectural components and their connections from business processes.

6.3 Software clustering

Clustering techniques have been widely used to decompose a software system to meaningful subsystems. Wiggerts presented a review of clustering algorithms for software clustering [6]. Mancoridis et al. devised a clustering system, Bunch [9], for identifying software architecture from the module dependency graph (MDG) of a software system. Tzerpos and Holt [30] presented a collection of subsystem patterns and proposed an ACDC (Algorithm for Comprehension-Driven Clustering) algorithm to produces decompositions of a software system by applying these subsystem patterns. Different from our proposed approach, these approaches take existing code or documentations as input to recover software architecture from legacy systems. Our proposed approach generates software architecture designs for designing and developing new business applications from requirements encapsulated in business processes. Lung et al. [46] also used clustering techniques to decompose requirements to subsystems. The authors manually identify dependencies among requirements and apply the hierarchical agglomerative clustering method to produce a hierarchy of clusters. However, the approach does not consider quality requirements. In our approach, we reuse business processes as a starting point and automatically generate software architectures with desired modularity.

6.4 Business driven development

Mitra [47] and Koehler et al. [48] presented business driven development approaches that use business processes to guide the development of business applications. A set of heuristics are provided to guide software architects to manually create models and components. Arsanjani [49] and Zimmermann et al. [50] use decision models to assist the identification of components from business requirements. The proposed approaches aim to reuse legacy components and create new components in order to reduce the development cost of business applications. In our proposed approach, we combine clustering analysis and quality metrics to automatically generate software architecture with desired modularity. Such generated architecture can be used as an initial guide for the subsequent architecture design.

7 CONCLUSION

In this paper, we present an approach to automatically generate software modular structures from business processes. Our proposed approach helps produce software modular structure by automating the creation of components and their interactions. Furthermore, our approach supports to maintian the consistency between business processes and software modular structures. Results of our case study illustrate that our proposed approach can generate meaningful software modular structures with desired modularity.

In the future, we plan to conduct more case studies to investigate the applicability of our approach in other domains. Moreover, we use metrics such as MQ and MoJoFM to compare our generated software modular structures with as-implemented software modular structures. We want to further assess the authoritativeness of our generated software modular structures by comparing them with those created by other approaches or experts. In addition, we will test the stability of our proposed approach using the empirical data suggested by Tzerpos and Holt. We are interested in studying the influence of different amount of changes on the stability evaluation results. Furthermore, our approach focuses on fulfilling functional requirements encapsulated in business processes. We will enhance our approach to optimize the generated software modular structures to achieve more quality requirements, such as reusability, portability and security. We are interested in developing techniques to refactor the generated software modular structures by applying software architectural styles.

REFERENCES

[1]. McDavid D. The Business-IT Gap: A Key Challenge, IBM Research Memo, http://www.almaden.ibm.com/coevolution/pdf/mcdavid.pdf [2010]

[2]. Hofmeister C, Kruchten P, Nord RL, Obbink H, Ran A, and America P, Generalizing a Model of Software Architecture Design from Five Industrial Approaches. *In the Proceeding of the 5th Working IEEE/IFIP Conference on Software Architecture*; pp. 77-88, 2005

[3]. Tang A, Han J, and Chen P, A Comparative Analysis of Architecture Frameworks, *In the Proceeding of the 11th Asia-Pacific Software Engineering Conference*; pp.: 640–647, 2004

[4]. Bass L, Clements P, and Kazman R. *Software Architecture in Practice, 2nd edition*, Addison-Wesley, 2003

[5]. Briggs R and Gruenbacher P. EasyWinWin: Managing Complexity in Requirements Negotiation with GSS. *In the Proceeding of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, vol. 1, 2002

[6]. Wiggerts TA. Using clustering algorithms in legacy systems re-modularization. *In the 4th Working Conference on Reverse Engineering (WCRE'97)*; pp. 33–43, 1997

[7]. Zhao X and Zou Y. A Business Process Driven Approach for Software Architecture Generation. *Proc. the 10th International Conference on Quality Software*, 2010.

[8]. Maqbool O and Babri HA. The weighted combined algorithm: a linkage algorithm for software clustering. *In Conference on Software Maintenance and Re-engineering (CSMR'04)*; pp. 15–24, 2004.

[9]. Mancoridis S, Mitchell BS, Chen Y, and Gansner ER. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. *In the Proceeding of the IEEE International Conference on Software Maintenance (ICSM'99)*; pp. 50, 1999

[10]. Hutchens DH and Basili VR. System Structure Analysis Clustering with Data Bindings. *IEEE Transactions on Software Engineering;* vol. 11(8), pp. 749-, 1985.

[11]. Anquetil N and Lethbridge TC. Experiments with clustering as a software re-modularization method. *In the 6th Working Conference on Reverse Engineering (WCRE'99)*; pp. 235–255, 1999.

[12]. Maqbool O and Babri HA. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering;* vol. 33(11), pp. 759-780, 2007.

[13]. Lung CH. Software Architecture Recovery and Restructuring through Clustering Techniques. *In the Proceeding of the third international workshop on Software architecture*; pp. 101-104, 1998.

[14]. Lung CH, Zaman M, and Nandi A. Applications of clustering techniques to software partitioning, recovery and restructuring. *Journal of Systems and Software*; vol. 73(2), pp. 227-244, 2004.

[15]. Mancoridis S, Mitchell BS, Rorres C, and Chen Y. Using automatic clustering to produce high level system organizations of source code. *In the Proceeding of the 6th International Workshop on Program Comprehension*; pp. 45, 1998.

[16]. Van Deursen A, and Kuipers T. Identifying objects using cluster and concept analysis. *In the Proceeding of the 21st international conference on Software engineering*; pp. 246–255, 1999

[17]. IBM WBI Modeler, http://www-01.ibm.com/software/integration/wbimodeler/ [2010]

[18]. Integrated definition methods (IDEF), http://www.idef.com/ [2010]

[19]. Business process modeling notation (BPMN), http://www.bpmn.org/ [2010]

[20]. Business process execution language (BPEL), http://www.ibm.com/developerworks/library/specification/ws-bpel/ [2010]

[21]. Wasserman AI. Toward a Discipline of Software Engineering. *IEEE Software*; pp. 23-31, 1996

[22]. Northrop LM and Richardson WE. Design Evolution: Implications for Academia and Industry. *In Proceeding of the SEI Conference on Software Engineering Education*; pp.205-217, 1991

[23]. IEEE. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. Institute of Electrical and Electronics Engineers, 2000

[24]. Merilinna J. *A Tool for Quality Driven Architecture Model Transformation*. PhD thesis, VTT Electronics, VTT Technical Research Centre of Finland, 2005

[25]. Ambriola V and Kmiecik A. Transformations for Architectural Restructuring. *Journal of Informatica*; vol. 28(2), pp. 117-128, 2004

[26]. Zou Y and Kontogiannis K. Migration to Object Oriented Platforms: A State Transformation *Approach. In the Proceeding of IEEE International Conference on Software Maintenance (ICSM)*, pp.530-539, 2002.

[27]. IBM WebSphere Commerce infocenter, http://publib.boulder.ibm.com/infocenter/wchelp/v6r0m0/index.jsp [2010]

[28]. Opentaps, http://www.opentaps.org/ [2010]

[29]. Zou Y, Guo J, Foo KC, Hung M. Recovering Business Processes from Business Applications. *Journal of Software Maintenance and Evolution: Research and Practice*; Vol. 21 (5), pp. 315-348, 2009

[30]. Tzerpos V and Holt RC. ACDC : An Algorithm for Comprehension-Driven Clustering. *In the Proceeding of the Seventh Working Conference on Reverse Engineering (WCRE'00)*; p. 258, 2000

[31]. Wen Z, and Tzerpos V. An Effective measure for software clustering algorithms. *In the Proceeding of the IEEE International Workshop on Program Comprehension*; pp. 194–203, 2004

[32]. Tzerpos V and Holt RC. On the Stability of Software Clustering Algorithms. *In the Proceedings of the 8th International Workshop on Program Comprehension*. pp. 211, 2000

[33]. Mitchell BS. *A Heuristic Search Approach to Solving the Software Clustering Problem*, PhD Thesis, Drexel University, Philadelphia, PA, 2002

[34]. Dhama H. Quantitative models of cohesion and coupling in software. *Journal of Systems and Software*; vol. 29 (1), pp. 65–74, 1995

[35]. Perry DE, Porter AA, and Votta LG. Empirical Studies of Software Engineering: A Roadmap. *In the Proceeding of the Conference on The Future of Software Engineering*; pp: 345-355, 2000

[36]. Nuseibeh B. Weaving Together Requirements and Architectures, *Computer*, vol. 34(3), pp.115-117, 2001

[37]. Grunbacher P, Egyed A, Medvidovic N. Reconciling software requirements and architectures with intermediate models. *Software and System Modeling*; vol. 3(3), pp. 235--253, 2004

[38]. Jackson MA. *Software Requirements and Specifications: a lexicon of practice, principles and prejudices*, ACM Press. Addison-Wesley Publishing, 1995

[39]. Zhang W, Mei H, and Zhao H. Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering*; vol. 11(3) , pp. 205-220, 2006

[40]. Garlan D and Shaw M. An Introduction to Software Architecture. *In Advances in Software Engineering and Knowledge Engineering*, vol. 1, 1993

[41]. Buschmann F, Meunier R, Rohnert H, Sommerlad P, and Stal M. *Pattern-Oriented Software Architecture*, Wiley, 1996.

[42]. Klein MH, Kazman R, Bass L, Carriere J, Barbacci M, and Lipson HF. Attribute-Based Architecture Styles, *In the Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*; pp.225-244, 1999

[43]. Zachman JA. A framework for information systems architecture. *IBM Systems Journal*; vol.26 (3), pp. 276-292, 1987

[44]. Kruchten P, Architectural Blueprints—The 4+1 View Model of Software Architecture. *IEEE Software*; vol.12 (6), pp.42-50, 1995

[45]. The Open Group Architecture Framework (TOGAF), www.opengroup.org/togaf/ [2010]

[46]. Lung CH, Xu X, and Zaman M. Software Architecture Decomposition Using Attributes. *International Journal of Software Engineering and Knowledge Engineering, Special Issue on Selected Papers from ICSEKE 2005*; 2005

[47]. Mitra T. Business-driven development, *IBM developerWorks article, IBM*, (2005)

[48]. Koehler J, Hauser R, Küster J, Ryndina K, Vanhatalo J, and Wahler M. The Role of Visual Modeling and Model Transformations in Business driven Development. *In Fifth International Workshop on Graph Transformation and Visual Modeling Techniques*; pp. 1-10, 2006

[49]. Arsanjani A. Service-oriented modeling and architecture (SOMA). *IBM developerWorks article*, IBM 2004

[50]. Zimmermann O, Koehler J, and Leymann F. Architectural Decision Models as Micro-Methodology for Service-Oriented Analysis and Design. *In Workshop on Software Engineering Methods for Service Oriented Architecture*, 2007

[51]. Ko RKL. A Computer Scientist's Introductory Guide to Business Process Management (BPM), *Crossroads*, vol. 15(4), 2009

[52]. Schwanke RW. An Intelligent Tool For Re-engineering Software Modularity, *In Proceedings of the 13th International Conference on Software Engineering*, pp. 83-92, 1991.

[53]. Sullivan KJ, Griswold WG, Cai Y, Hallen B. The Structure and Value of Modularity in Software Design, *In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 99-108, 2001.

[54]. Al-Otaiby TN, AlSherif M, Bond WP. Toward software requirements modularization using hierarchical clustering techniques, *In Proceedings of the 43rd annual Southeast regional conference*, vol. 2, pp. 223-228, 2005.

[55]. Card DN, Glass RL. *Measuring Software Design Quality*. Prentice Hall,Englewood Cliffs, N.J. 1990.