Improving the Modifiability of the Architecture of Business Applications

Xulin Zhao, Foutse Khomh, and Ying Zou
Dept. of Elec. and Comp. Engineering
Queen's University
Kingston, Ontario, Canada
{xulin.zhao, foutse.khomh, ying.zou}@queensu.ca

Abstract—**In the current rapidly changing business environment, organizations must keep on changing their business applications to maintain their competitive edges. Therefore, the modifiability of a business application is critical to the success of organizations. Software architecture plays an important role in ensuring a desired modifiability of business applications. However, few approaches exist to automatically assess and improve the modifiability of software architectures. Generally speaking, existing approaches rely on software architects to design software architecture based on their experience and knowledge. In this paper, we build on our prior work on automatic generation of software architectures from business processes and propose a collection of model transformation rules to automatically improve the modifiability of software architectures. We extend a set of existing product metrics to assess the modifiability impact of the proposed model transformation rules and guide the quality improvement process. Eventually, we can generate software architecture with desired modifiability from business processes. We conduct a case study to illustrate the effectiveness of our transformation rules.**

***Keywords*: Modifiability, Software architecture, Quality evaluation, Software metric.**

I. INTRODUCTION

Nowadays business environments undergo frequent changes due to fast market growth and technological innovations. Business processes describe the business operations of an organization and captures the business requirements of business applications. A business process is composed of a set of interrelated tasks that are joined together by control flow constructs and data flows. The control flow constructs specify the valid execution order (e.g., sequential, parallel, or alternative) of tasks. Data items represent the information flowed among tasks. For example, a business process for purchasing a product on-line consists of the following sequence of tasks such as *Select product*, *Add to the shopping cart*, and *Validate buyer's credit card*.

Business processes are specified by business analysts and implemented by developers in business applications. As the business processes change, organizations must modify their business applications to continue supporting the processes. In particular, the modifiability of a business application determines the easiness of the application to be modified in response to changes caused by the environment, requirements or functional specification [2]. Studies indicate that 50-70% of the cost in the lifecycle of a software system is devoted to modifications after the initial development [10]. Therefore, improving the modifiability is critical to reduce development costs and ensure the success of a business application.

Software architecture plays an important role in preparing applications for likely changes in order to reduce the cost associated with their modification. Software architecture describes the overall structure of an application using software components, the interactions among software components (i.e., connectors), and the constraints on the components and connectors. Specifically, a component captures a particular functionality. Connectors define control and data transitions among components. Constraints specify the properties of components and connectors.

To bridge the gap between business requirements and business applications, our earlier work [21] automatically generates architectural components from business processes. The generated architecture captures the functionality specified by business processes. More specifically, we use clustering techniques to group functionally similar tasks in business processes to produce software components. The control flow and data flow between tasks that are captured in different software components are converted into the connectors among the components.

In the existing approaches [4][11][23][24], software architectures are high-level abstractions of business applications in terms of components and connectors. There is no information available to describe the content of a component and data structures in the initial development stage. Scenario based approaches such as the architecture-level modifiability analysis (ALMA) approach by Bengtsson et al. [2] are used to manually analyze the quality of software architecture in the early stage without low level details of software systems. ALMA is a unified architecture-level analysis approach that focuses on modifiability and consists in five main steps: goal selection, software architecture description, change scenario elicitation, change scenario evaluation and interpretation. Such approaches require software architects to create scenarios based on their knowledge and experience. Therefore, such an approach is labor intensive and error prone. In addition, the subjective judgement involved in such approaches raises questions of the credibility of the evaluation results.

Numerous software product metrics are used to automatically evaluate the quality of software design and implementations. However, these product metrics require detailed design or source code information not available at architectural level and therefore cannot be used to evaluate software architectures. Software product metrics take the data flows and control flows among the components of various granularities (e.g., classes, methods) as inputs to produce numerical evaluations of the quality of a software product; without such detailed information, it is difficult to automatically evaluate quality attributes of a software architecture [4][5].

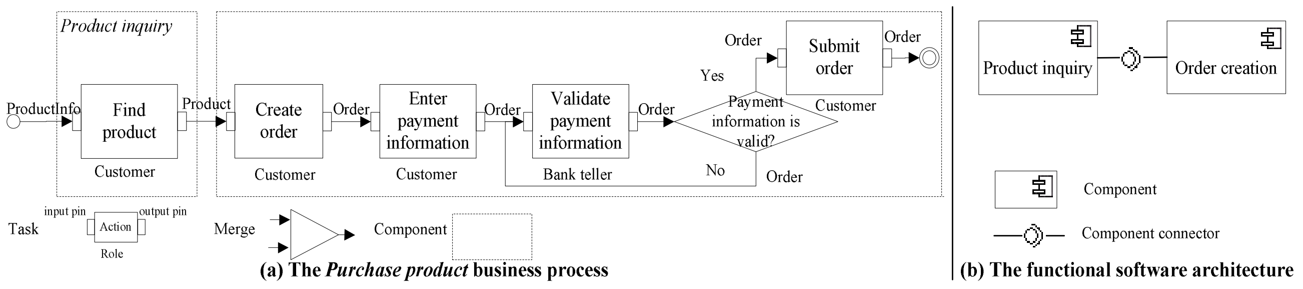

Figure 1: The Purchase product business process and the generated functional software architecture

In our prior work [21] on the automatic generation of software architectures, the generated software architectures are enriched with information on tasks and data items specified in business processes. This information provides detailed description of the functionalities and data structures of software components. It becomes feasible to evaluate the quality of the generated architecture using the existing product metrics by leveraging the control flow and data flow information embedded in business processes. As an extension of our prior work, this paper presents an approach that uses the business process entities associated with the generated architecture to evaluate the quality of the generated architecture. Due to their distributed nature, business applications are commonly developed by applying the three-tier software architectural style [28]. The three-tier software architectural style helps improve modifiability and reusability by separating user interfaces and persistent data from business logics to standardize interfaces between tiers. The standardization of interfaces between tiers prevents the propagation of modifications. In our work, we aim to restructure the generated architecture to improve the modifiability. We propose a collection of transformation rules to convert the generated architecture to a three-tier software architecture. We extend a collection of existing product metrics to evaluate the quality impact of our transformation rules and guide the generation of software architecture to achieve high modifiability.

The rest of this paper is organized as follows: Section II presents an overview of business processes. Section III describes overall steps of our software architecture generation approach. Section IV discusses our approach for evaluating the modifiability of software architecture. Section V presents the transformation rules for restructuring software architecture. Section VI illustrates the feasibility and effectiveness of our approach through a case study. Section VII discusses existing work related to our approach. Finally, Section VIII concludes the paper and explores the future work.

II. OVERVIEW OF BUSINESS PROCESSES

In this section, we give an overview of business processes. Figure 1 (a) illustrates a business process for purchasing products online. The business process consists of five tasks for helping customers find products, create orders, and make payments. A task represents a primitive activity that accomplishes a business objective. Each task is associated with a role which specifies a subject to perform the task. A role can be a software component, a device, or an abstract representation of a category of users that have common responsibilities. For example shown in Figure 1 (a), the role, *Customer*, performs all tasks in the example business process except the task, *Validate payment information*.

The specification of a task consists of an action and a number of pins that represent data transitions among tasks. The action of a task is often designated by the name of the task using a descriptive text, such as *Find product* and *Create order*. We classify tasks into three categories according to their functionalities:

- *Human tasks* are interacted with users to take a user's inputs and display the results of a user's requests. The roles of human tasks are often human, such as *Customer*;
- *Automatic tasks* carry out the computation. The role of automatic tasks is often software components or devices.
- *Data-access tasks* are special case of automatic tasks and encapsulate the operations for database manipulation such as *Insert*, *Update*, *Query*, and *Delete*.

Input pins hold input data used by a task. Output pins contain output data generated by a task. For instance, the task, *Find product*, receives the data item, *ProductInfo*, from its input pin and returns the data item, *Product*, through its output pin. As specified in a business process specification, a data item contains a set of attributes of various types, such as primitive types (e.g., String, Integer, or Float) or user defined types. Figure 2 shows the definition of the data item, *Order* (shown in Figure 1 (a)), and the attributes of the data item.

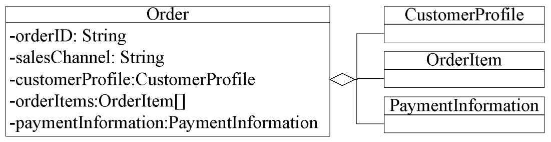

Figure 2: The definition of the data item, *Order*

A data item can be transferred between tasks or between a task and an external entity (e.g., user or device) in an operational environment. A variety of data items can be exchanged between tasks and the operational environment. For instance, a task might accept requests from users, send a surface mail to users, retrieve data from an external database, or download data from an external server. We divide the data exchange into two types: 1) an *inter-task data interaction* happens when the output data item of a task is served as the input data item of another task; 2) an *environment-task data interaction* occurs when a data item is passed between a task and the operational environment of the task.

Control flows specify the execution orders of tasks. The control flow of a business process contains four control flow constructs: *Sequence*, *Loop*, *Alternative*, and *Parallel*. Tasks in a sequence are accomplished one after another. A loop is composed of a set of tasks executed iteratively. Alternative allows one execution path to be chosen from multiple possible execution paths with respect to the output of the preceding task, the value of a data item, or the result of a user decision. A parallel defines multiple execution paths that can be executed simultaneously.

III. Overview of Our Approach

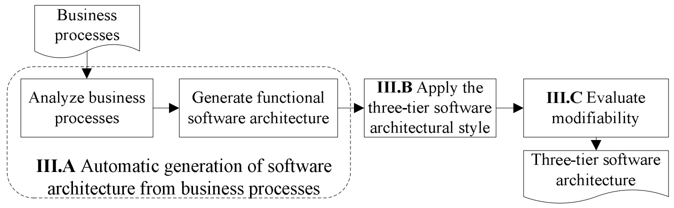
Business processes
Analyze business processes
Generate functional software architecture
III.B Apply the three-tier software architectural style
III.C Evaluate modifiability
Three-tier software architecture
III.A Automatic generation of software architecture from business processes

Figure 3: Overview of our approach

Figure 3 gives an overview of our approach which is broken down into three major steps: (1) automatic generation of software architectures from business processes; (2) application of the three-tier software architectural style to improve the modifiability of the generated software architecture; (3) evaluation of the modifiability of the software architecture.

A. *Automatic Generation of Software Architecture from Business Processes*

As discussed in our prior work [21], we identify components and connectors by analyzing their tasks, data items, and control dependencies specified in business processes. In a business process specification, tasks specify functionalities of the underlying business application and data items designate data structures of the business application. The architecture is generated in three steps:

1) Grouping dependent data items to define data structures for components: two data items are dependent if one of the data items is an input of a task and produces the other data item as an output of the task. For example shown in Figure 1 (a), *ProductInfo* and *Product* are dependent data items. Moreover, all the data items used in the same business processes are dependent. We apply the WCA clustering algorithm [29] to group the data items with strong inter-dependencies into a data group. For example, Figure 4 illustrates the clustering result of the data items specified in business processes (shown in Figure 1 (a)).

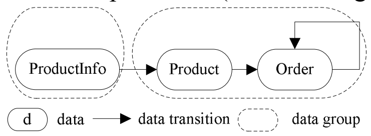

Figure 4: Example of result data groups

2) Associating tasks to data groups for describing functionalities of components. A task is dependent on a data item if the task either takes a data item as an input or generates a data item as an output. To generate software components, we analyze the dependency strength between tasks and data groups. Similar to forming data groups, we apply the unbiased Ellenberg metric [29] to calculate dependency strength between tasks and data groups. For example shown in Figure 1 (a), the task, *Find product*, is closely related to the data group, *{ProductInfo}*. Therefore, we combine the task, *Find product*, and the data group, *{ProductInfo}*, to form a software component, *Product inquiry*. Similarly, we compose the remaining tasks and the data group, *{Product, Order}*, to create another component, *Order creation*.
3) Deriving connectors from data and control flows among tasks. For our running example business processes shown in Figure 1 (a), we create a connector between the two components, *Product inquiry* and *Order creation*, from the task transition between the task, *Find product*, and the task, *Create order*. Figure 1 (b) illustrates the resulting functional software architecture. We use UML (Unified Modeling Language) notations to describe the generated software architecture designs since UML has become the de facto software modeling language.

B. *Application of the Three-tier Software Architectural Style*

We apply the three-tier software architectural style to improve the modifiability of the architectures generated from the business processes. The three-tier software architectural style divides a business application into the following tiers:

- The presentation tier which receives inputs from users and displays outputs to users.
- The business logic tier which consists of components that complete requests from users.
- The data tier which provides methods for storing and retrieving data in repositories.

We use model transformation techniques to automate this restructuring of the architecture. A model transformation technique takes software architecture as input and produces a modified version of this architecture as output. A model transformation is often described as a collection of transformation rules. A transformation rule identifies information from the source software architecture and uses the identified information to create or update constructs in the target software architecture. Each transformation rule can impact quality attributes of the target software architecture positively or negatively.

C. *Evaluation of the Modifiability of the Software Architecture*

To generate software architecture with desired quality attributes, we perform quality evaluations to assess the impact of our transformation rules and guide the generation of target software architecture. We perform a modifiability evaluation of architectures during the application of the three-tier software architectural style. Modifiability being a high-level quality attribute, a direct measurement is difficult. Therefore, we treat an abstract modifiability requirement as a high level goal and iteratively refine this high level goal into a set of low level goals that can be directly measured using metrics. As pointed

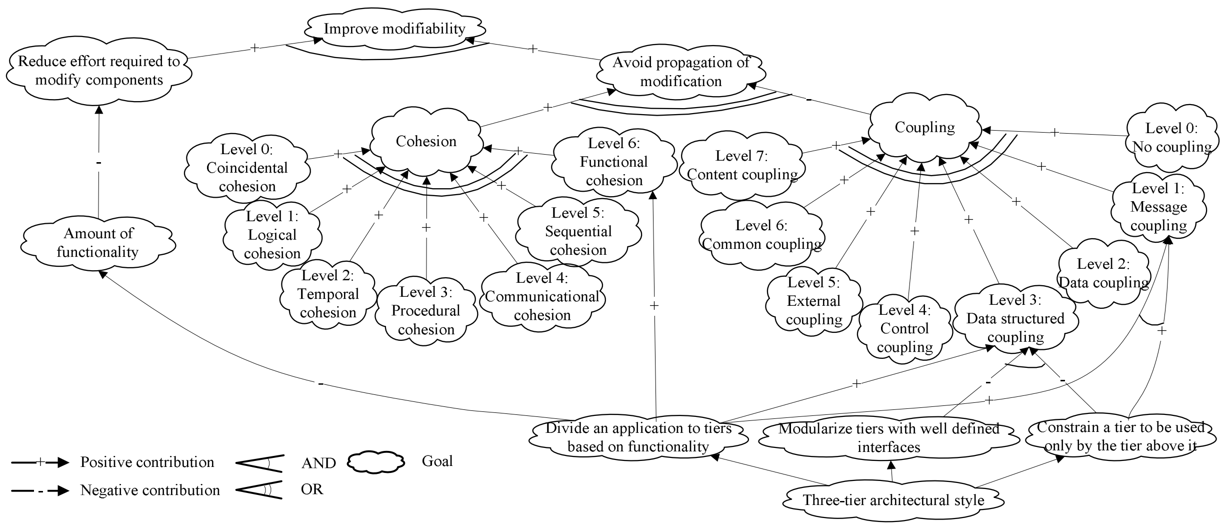
Improve modifiability
Reduce effort required to modify components
Avoid propagation of modification
Cohesion
Coupling
Level 0: Coincidental cohesion
Level 1: Logical cohesion
Level 2: Temporal cohesion
Level 3: Procedural cohesion
Level 4: Communicational cohesion
Level 5: Sequential cohesion
Level 6: Functional cohesion
Level 7: Content coupling
Level 6: Common coupling
Level 5: External coupling
Level 4: Control coupling
Level 3: Data structured coupling
Level 2: Data coupling
Level 1: Message coupling
Level 0: No coupling
Amount of functionality
Divide an application to tiers based on functionality
Modularize tiers with well defined interfaces
Constrain a tier to be used only by the tier above it
Three-tier architectural style
Positive contribution
Negative contribution
AND
OR
Goal

Figure 5: The modifiability goal model

out by Bachmann et al. [3], modifiability can be improved by reducing effort required to modify components and by avoiding propagation of modifications. We break down the goal, *Improve modifiability*, into two sub-goals: *Reduce effort required to modify components* and *Avoid propagation of modification*. These sub-goals are further refined to more concrete goals as illustrated in Figure 5. The three-tier software architectural style improves cohesion and reduces coupling among components by distributing the functionality of a business application into three tiers and standardizing interfaces between tiers [16]. We use the soft goal graph [20] to represent the elaboration of modifiability goal. In the soft goal graph, goal relationships are represented by AND and OR links. An AND relation represents that all sub-goals need to be satisfied in order to achieve the parent goal. For example, sub-goals of the goal, *Improve modifiability* (shown in Figure 5), contribute to the goal with an AND relation. An OR relation denotes that the satisfaction of one of the sub-goals is sufficient to achieve their parent goal. The lowest level goals can be directly measured using metrics. For the purpose of demonstrating our approach, the goal graph shown in Figure 5 provides only a primitive list of metrics.

Changes of metric results reflect the quality impact of transformation rules. The changes are propagated to upper goals through links. Positive changes in the low level goals indicate the improvement of the upper changes. We label such links with "+" sign. Similarly, negative changes of the low level goals show the negative impact on the upper level goals. We mark such links with "-" sign.

IV. METRICS FOR MODIFIABILITY EVALUATION

As shown in Figure 5, the modifiability can be evaluated using metrics for evaluating function points, cohesion and coupling. The traditional metrics take software entities, such as variables, statements, and methods, as input for computation. However, the generated software components are described using business process entities, such as tasks and data items. We extend traditional metrics to take business process entities as inputs for measuring the amount of functionality, cohesion and coupling of the generated architectural components.

A. Measuring Amount of Functionality

If a component contains more functionality, it becomes more complex to modify [3]. Function point analysis (FPA) is used to evaluate the amount of functionalities in components and software systems. We calculate the number of function points of a component using the approach defined in the ISO functional size measurement (FSM) standard [6] which evaluates functionalities using files, interfaces and interactions among applications. In our generated software architectures, detailed designs of components are described using data items and tasks derived from business processes. We analyze dependencies among data items and tasks to compute function points for each component. Specifically, following FSM, we re-mapped functionalities into the following five functional factors:

- *Internal logic files*, which hold data items used within a component. For example, the component, *Order creation*, creates and edits the data item, *Order*. Therefore, the internal logic file of the component is *Order*.
- *External interface files*, which contain external data received from the operational environment. For example, the component, *Product inquiry*, receives the data item, *ProductInfo*, as input. The data item, *ProductInfo*, specifies the data received from the operation environment and is identified as the external interface file of the component.
- *External inputs*, which refer to input pins of tasks that hold external data. For example, the input pin of the task, *Find product*, holds the input data item, *ProductInfo*, for the component, *Product inquiry*. Hence, we identify the input pin of the task, *Find product*, as the external input of the component.
- *External outputs*, which correspond to output pins of tasks that return data to the operational environment. For example, the output pin of the task, *Find product*, holds the return data of the component, *Product inquiry*.

Therefore, we identify the output of the task, *Find product*, as the external output of the component.

- *External inquiries*, which are tasks that capture data access actions. For example, the task, *Find product*, captures the data access action in the component, *Product inquiry*. Hence, we identify the task, *Find product*, as the external inquiry of the component.

The function points of a component are the weighted sum of different types of functional factors within this component. For each functional factor, the FSM approach defines three levels of complexity and specifies a weight for the functional factor at each complexity level. Table 1 shows the complexity levels and corresponding weights for the five functional factors. In the *Product inquiry* component shown in Figure 1 (b), inputs from customers are captured in the data item, *ProductInfo*, which contains 2 primitive data attributes as shown in Figure 6. As specified in the FSM standard, the complexity of a data item with no more than 50 attributes is *Low*. Hence, the *External interface file* of the component, *Product inquiry*, has a *Low* complexity. Similarly, the complexity levels of the other three functional factors are *Low*. As shown in Table 1, weights for the four factors are 5, 3, 4, and 3, respectively. Therefore, the number of function points of the component, *Product inquiry*, is 15 as illustrated in Table 2.

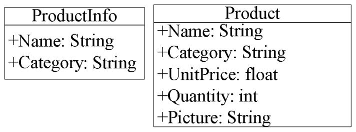

Figure 6: The definition of *ProductInfo* and *Product*

Table 1: Weigths of functional factors in FSM

Functional factor	**Low**	**Average**	**High**
Internal logic files	7	10	15
External interface files	5	7	10
External inputs	3	4	6
External outputs	4	5	7
External inquiries	3	4	6

Table 2: Function point count for *Product inquiry*

Functional factor	**Number**	**Weight**	**Function point**
Internal logic files	0	7	0×7
External interface files	1	5	1×5
External inputs	1	3	1×3
External outputs	1	4	1×4
External inquiries	1	3	1×3
Total			15

Two alternative architecture designs tend to have the same number of function points when they are generated from the same set of business processes. To evaluate the functionality distribution among components, we use the median value of function points from the components in an architecture design to quantify the overall cost of modifying components in a software architecture design. The median value can avoid the impact of the extreme values of functional points of components.

B. *Measuring Cohesion and Coupling*

Cohesion measures the strength of intra-dependencies within a component while coupling assesses the strength of inter-dependencies among components. Cohesion is divided into 6 distinct levels [12] and coupling is broken down to 7 distinct levels [11][15]. Figure 5 illustrates levels of cohesion and coupling in the order from the worst (low cohesion, high coupling) to the best (high cohesion, low coupling). Table 3 and Table 4 show the characteristics of data items and tasks in components for determining the cohesion levels and coupling levels of components. Components with low cohesion and high coupling have much stronger ripple effects of changes than others [11].

We determine the cohesion level of a component by analyzing tasks and data items within the component. If they present characteristics corresponding to a cohesion level *h*, the cohesion level of the component is *h*. The level of coupling between two components is determined by analyzing dependencies between the two components. When the dependencies present characteristics of a coupling level *u* the coupling level of the two components is *u*. Once the levels of cohesion and coupling are determined, we calculate the dependency strength at each cohesion (or coupling) level. A number of metrics [9][18] can be used to calculate dependency strength among components. We use the connectivity metrics defined in [9] to calculate dependencies among components. We chose these metrics because they are designed to evaluate cohesion and coupling based on dependency graphs that have similar structure to business processes; and they have been applied and validated on many systems [9].

The cohesion strength of a component is evaluated by analyzing the number of dependencies within this component as shown in Equation (1). This cohesion strength increases as the number of dependencies within the component grows. The cohesion strength of a software architecture is the average cohesion strength of all components within the software architecture. For our example from Figure 1 (b), we analyze the

Table 3: Cohesion levels and their characteristics

Category	**Cohesion level**	**Characteristics**
High level	Functional	A component performs a single task.
Moderate levels	Sequential	A data item is sequentially transferred across tasks within a component.
	Communicational	All tasks within a component share the same input or output data items.
	Procedural	Tasks within a component are connected by control connectors.
Low levels	Temporal	Tasks within a component are correlated by temporal relations.
	Logical	Tasks that are logically grouped to perform the same type of functionalities.

Table 4: Coupling levels and their characteristics

Category	**Coupling level**	**Characteristics**
High levels	Content	A component uses data or control maintained by another component.
	Common	Components share global data items.
	External	Components are tied to external entities such as devices or external data.
Moderate levels	Control	Controls flow across components.
Low levels	Data structured	Structured data are transferred among components.
	Data	Primitive data or arrays of primitive data are passed among components.
	Message	Components communicate through standardized interfaces.

$$S_{Cohesion} = \frac{\sum_{i=1}^{m} A_i}{m}, A_i = \frac{\mu_i}{n_i^2} \quad (1)$$

A_i is the strength of dependency within the ith component. μ_i refers to the number of dependencies within the ith component. n_i is the number of tasks within the component, and m is the number of components within the generated software architecture.

dependencies introduced by task transitions to evaluate the strength of the procedural cohesion of the architecture. The component, *Product inquiry*, contains no task transition. Therefore, the strength of procedural cohesion of the component, *Product inquiry*, is 0. The component, *Order creation*, includes 4 task transitions and 4 tasks. Hence, the strength of the procedural cohesion of the component, *Order creation*, and the overall cohesion strength of the functional software architecture are calculated as follows:

$$A_{OrderCreation} = \frac{4}{4^2} = \frac{1}{4} \text{ ; } S_{Cohesion} = \frac{0 + \frac{4}{4^2}}{2} = \frac{1}{8}.$$

$$S_{Coupling} = \frac{\sum_{i,j=1}^{m} E_{i,j}}{m(m-1)/2}, E_{i,j} = \begin{cases} 0 & i = j \\ \frac{\varepsilon_{i,j}}{2n_i n_j} & i \neq j \end{cases} \quad (2)$$

$E_{i,j}$ is the inter-dependency between the ith component and the jth component. $\varepsilon_{i,j}$ is the total number of dependencies between the two components. n_i and n_j are the number of tasks in the two components and m is the total number of components in the generated software architecture.

The coupling between two components is assessed by examining the number of dependencies between the two components using Equation (2). The coupling strength between two components increases as the number of dependencies between the two components grows. The coupling strength of a software architecture is the average coupling strength among components within the software architecture. For our running example shown in Figure 1 (b), a control connector is used to connect the two components: *Product inquiry* and *Order creation*. The component, *Product inquiry*, contains 1 task; and the component, *Order creation*, has 4 tasks. Hence, the control coupling of the software architecture is: $S_{Coupling} = \frac{\frac{1}{2\times1\times4}}{(2\times1)/2} = \frac{1}{8}$.

We denote the cohesion and coupling of a software architecture design using the pair value (*h*, *s*) and *(u, s)*. *h* and *u* designates the levels of cohesion and coupling, respectively; *s* refers to the strength of this cohesion or coupling at these levels. To evaluate the overall cohesion of an architecture, we produce a paired value for each level of cohesion or coupling since components may have different levels of cohesion or coupling in a software architecture.

V. Model Transformation Rules for Architecture Improvement

In our prior work [21], we generate software architecture from business processes. Such software architectures, referred to as functional software architecture, describe the initial distribution of functionality without the conformance to the three-tier architectural style. The structure of these software architectures is illustrated in the meta-model shown in Figure 7. We define a set of model transformation rules that automatically restructures software architectures into three-tier software architecture. Our transformation rules distribute these functionalities into components in different tiers. We use object constraint language (OCL) [13] to specify each transformation rule. In the following subsections, we first describe the transformation rules that produce three-tier architecture. Then we describe the generation of connectors among the components in different tiers. Finally, we discuss our technique for evaluating the quality impact of each transformation rule.

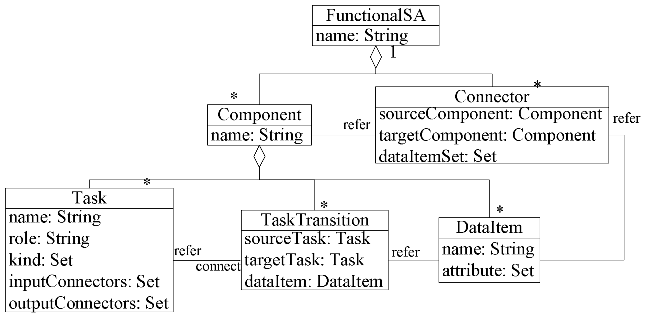

Figure 7: The meta-model of functional software architecture

A. *Generation of Three Tiers*

Functionalities of software architectures generated from business processes are described by the tasks specified in business processes. These tasks capture different functionalities as discussed in Section II. In our work, we take components in the functional software architecture as source components and define transformation rules to automatically generate target components in the three-tier software architectural style. One source component can be broken down into fine grained components which contain unique functionality, and are distributed into different tiers in the target components. In the rest of this subsection we discuss the transformation rules by analyzing tasks and their transitions in source components.

Rule 1: Presentation Tier Generation Rule

$T_p = \{t | t.kind \rightarrow includes(\text{'Human task'})\}$

$C_p = \{c \mid c.sourceTask.kind \rightarrow includes(\text{'Human task'}) \text{ and }$
$c.t\arg etTask.kind \rightarrow includes(\text{'Human task'})\}$

Where t is an instance of Task and c is an instance of TaskTransition as defined in Figure 7.

This rule identifies the presentation tier by grouping human tasks in a source component into a UI component and moving the UI component into the presentation tier. If the role of a task refers to a group of users, we classify such a task as a human task. The expression, *T_p*, selects tasks classified as human tasks from a source component to form one UI component. Moreover, we generate connectors among UI components by analyzing transitions among human tasks that are distributed in different UI components. The expression C_p, identifies task transitions whose source tasks and target tasks are both human tasks. Eventually, components in the presentation tier capture the functionality to handle communications between users and business logics.

Rule 2: Business Logic Tier Generation Rule

$T_b = \{t \mid t.kind \rightarrow includes('\text{Automatic task}')\}$

$C_b = \{c \mid c.sourceTask.kind \rightarrow includes('\text{Automatic task}')$ and $c.targetTask.kind \rightarrow includes('\text{Automatic task}')\}$

Where t is an instance of a Task and c is an instance of TaskTransition as defined in Figure 7.

To provide support for users, we generate components in the business logic tier to handle requests from users and produce the corresponding responses. As specified in the expression, T_b, we select automatic tasks and their transitions from a source component to generate components and connectors within the business logic tier. The expression, extracts automatic tasks from a source component to form a component in the business logic tier. As described in the expression, C_b, we generate connectors among components in the business logic tier by analyzing transitions among automatic tasks. More specifically, the expression, C_b, checks the types of source and target tasks of task transitions and returns those whose source and target tasks are both automatic tasks.

Rule 3: Data Tier Generation Rule

$T_d = \{t \mid t.kind \rightarrow includes('\text{Data-access task}')\}$

Where t is an instance of Task as defined in Figure 7.

We create data access components in the data tier to support the communications between data access tasks and data repositories. We identify data access tasks from a source component to generate data access components by matching verbs in task names with the data access operations (e.g., insert, update and delete). Such operations are abstractions of fundamental database operations. Table 5 lists the heuristic mappings between task names and data access operations. The expression, T_d, extracts data access tasks from a source component. Data access components provide support for accessing persistent data without passing data between each other. Hence, we do not generate connectors among data access components.

Table 5: Heuristic mappings

Data access operations	Verbs in task names
Insert	Create, Add, Insert
Update	Update, Modify
Query	Find, Search, Select, Get
Delete	Delete, Remove

B. Generation of Connectors between Tiers

Beside connectors among components within each tier, three-tier software architectures also contain connectors between tiers:

- *Presentation-Business* connectors that describe communications between components in the presentation tier and components in the business logic tier.
- *Business-Data* connectors that capture communications between components in the business logic tier and components in the data tier.

Rule 4: Presentation-Business Connectors Generation Rule

$T_{p-b} = \{t \mid t.kind \rightarrow includes('\text{Human task}')$ and $t.kind \rightarrow includes('\text{Automatic task}')$

Where t is an instance of Task as defined in Figure 7.

Presentation-Business connectors transfer information between human tasks in the presentation tier and the automatic tasks in the business logic tier. If a task captures users' interactions and requires automated support, it is broken down to a human task and an automatic task. For tasks with this feature, we generate a presentation-business connector to transit requests and responses between the presentation tier and the business logic tier as specified in the expression T_{p-b}.

Rule 5: Business-Data Connectors Generation Rule

$T_{b-d} = \{t \mid t.kind \rightarrow includes('\text{Automatic task}')$ and $t.kind \rightarrow includes('\text{Data-access task}')\ \}$

Where t is an instance of Task as defined in Figure 7.

Business-Data connectors pass information between automatic tasks in the business logic tier and data access tasks in the data tier. If an automatic task encapsulates data access operations, it is divided into a data access task and an automatic task. For tasks with this feature, we generate a business-data connector to transmit data between the business logic tier and the data tier. The expression T_{b-d} extracts automatic tasks with data access operations for generating business-data connectors.

C. Application of Rules

To assess the quality impact of a model transformation rule, we compare the quality of software architectures before and after applying the transformation rule. If the impact confirms to specified quality requirements, *i.e.,* the modifiability, we deem that the model transformation is effective. Otherwise, we choose other transformation rules to improve the generated software architecture. Such an evaluation and improvement process is iterated until the modifiability is maximized.

To compare the coupling of the source software architecture, SA_{before} and the transformed software architecture, SA_{after}, we compare the coupling level by level, in the order from high to low and consider that SA_{after} has lower coupling if:

- its coupling strength is less than that of SA_{before} when both are in the same level of coupling,
- a high level of coupling is present in SA_{before} but absent in SA_{after} or, a lower level of coupling is absent in SA_{before} but present in SA_{after}.

Similarly, we compare the cohesion of SA_{before} and SA_{after} to evaluate the impact of a transformation rule on the cohesion of a software architecture. The comparison of function points of SA_{before} and SA_{after} is performed using median values as discussed in Section IV.A. We strive to generate software architecture design with high cohesion, low coupling, and low function points to maximize the modifiability.

VI. CASE STUDY

The *objectives* of this case study are two-fold: 1) evaluate the effectiveness of our model transformation rules in improving the modifiability of software architectures; 2) validate our extended metrics. The *context* of this case study is the system IBM WebSphere Commerce (WSC) server [17] which is a commercial platform for building e-commerce web sites and applications. This system implements business processes to support both B2B (business-to-business) transactions and B2C (business-to-consumer) transactions. The business processes for the system are available online [17]. Table 6 shows a summary of tasks in the system. These business processes are classified into five categories as shown in the first column of Table 6. The second and third columns of Table 6 describe the number of distinct tasks and the description of tasks in each category of business processes. All these business processes are modeled using WebSphere Business Modeler [1] and stored as XML documents. We developed a business process parser for XML documents to extract entities described in Section II.

Table 6: Summary of tasks in IBM WSC

Application	# Tasks	Description
Marketing	69	Facilitate marketing campaigns and activities
Merchandise management	66	Create, load, and manage products in online stores
Order management	204	Manage state of orders and components of orders
Customer management	17	Create and manage customer profiles and member resources
Customer service	22	Assist customer service representatives to provide services to customers
Total	378	

A. Evaluation Method

Transformation rules. To assess the effectiveness of our model transformation rules, we first generates functional software architecture designs using the approach presented in our prior work [21] and summarized in Section III.A. Then, we apply our transformation rules described in Section V to convert this functional architecture into the three-tier software architecture. To evaluate the effectiveness of our automatic quality improvement, we use our extended metrics described in Section IV and the state-of-the-art scenario based evaluation method, ALMA [2], to compare the modifiability of the architectures before and after the restructuring and assess the quality improvement.

Table 7: Elicited scenarios

ID	Description	Source
1	Advances in web technologies (e.g., Web 2.0) enable the development of more attractive user interfaces. The look and feel of the business application needs to be updated to improve customer experience.	Functional requirement
2	The organization establishes collaboration with a new shipping company. Consequently, a new shipping method needs to be added to the business application.	
3	Customers of the business application substantially increased. The throughput of the business application needs to be enhanced to reduce response time.	Quality requirement
4	After a period of execution, the maintainers find that the product repository is often queried in joint with the manufacturer repository. Therefore, they decide to merge the two repositories to improve the performance.	
5	The bank changed the interfaces to the payment validation system. Therefore, the business application needs to be modified to adapt to the new interfaces.	Operational environment

Metrics. To validate our extended metrics, we compare results of modifiability evaluations with metrics to those obtained by ALMA. A consistency of results between our metrics and ALMA would provide a quantitative validation for these metrics, since ALMA has already been validated through several cases studies [2]. To apply ALMA, we identify representative scenarios caused by three common sources of changes: functional requirements, quality requirements, and the operational environment of the application. Table 7 summarizes these scenarios. Changes in functional requirements result from technologic innovations or changes in business rules. Quality problems are often identified after the application is delivered. In this case, we need to modify the application to fix the quality problems. Changes in the operation environment are often manifested as changes in interfaces to external systems. Examples of external systems can be client systems and the payment validation system. We use the component modification ratio [2] to assess the impact of a scenario on a given software architecture. The component modification ratio is the ratio between the number of affected components and the total number of components in the given software architecture.

B. Study Results

This section reports and discusses the results of our case study. Figure 8 and Figure 9 show the two versions of software architecture of IBM WSC, respectively. Figure 8 illustrates the initial functional software architecture (i.e., the source architecture) that is generated from business processes, while Figure 9 shows the three-tier software architecture (i.e., the transformed architecture) produced by applying our transformation rules. By breaking the architecture into three tiers, we aim to improve the modifiability of the business application.

Table 8 shows the modifiability values of the two versions of software architecture of IBM WSC. Each row of the table shows the results of a metric. The second and third columns illustrate metric results corresponding to the initial functional software architecture and the three-tier software architecture, respectively. Column 4 shows the improvement rate in the metric result after applying our quality improvement transformation. We observe that: (i) the number of median function points in the three-tier architecture design is reduced. Therefore, components in the three-tier software architecture have fewer function points than those in the functional software architecture. Consequently, applying the three-tier software architectural style can reduce the effort required to modify components; (ii) the cohesion levels in both versions of the architecture are procedural cohesion. However, the strength of procedural cohesion of the three-tier architecture is **0.0646** which is higher than that of the functional software architecture. An improvement in cohesion strength helps prevent the propagation of modifications and increases the modifiability of the business application; (iii) both versions of the architecture present common coupling, control coupling, and data structured coupling. The coupling strengths of the three-tier software architecture are **0.0023**, **0.0009**, and **0.0005**, respectively. These values are lower than those of the functional software architecture. In addition, the three-tier software architecture designs further reduce coupling among components by introducing a new low level coupling, message coupling. We use N/A to denote that the functional software architecture designs do not present the message coupling. The introduction of message coupling isolates interactions among components and prevents propagations of modifications. Hence, applying the three-tier software architectural style can greatly improve the modifiability of a business application.

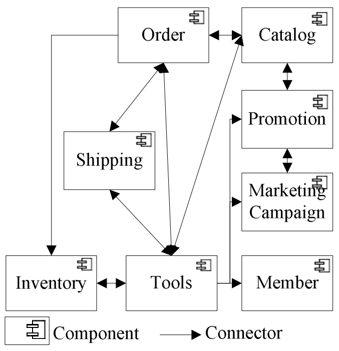

Figure 8: Functional software architecture of IBM WSC

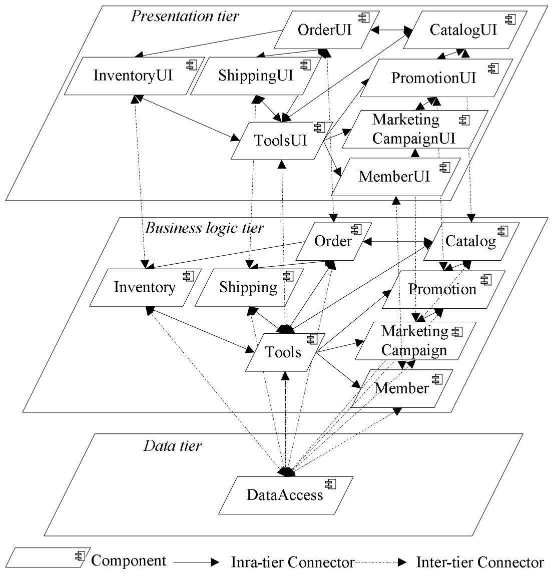

Figure 9: Three-tier software architecture of IBM WSC

Table 8: Metric values of IBM WSC

Metric	Functional SA	Three-tier SA	Improvement
Median function point	368	231	37.2%
Cohesion	(Procedural cohesion, 0.0467)	(Procedural cohesion, 0.0646)	38.3%
Coupling	(Common coupling, 0.0111)	(Common coupling, 0.0023)	79.4%
	(Control coupling, 0.0024)	(Control coupling,0.0009)	62.5%
	(Data structured coupling, 0.0024)	(Data structured coupling, 0.0005)	79.2%
	N/A	(Message coupling, 0.0238)	N/A

Table 9: Component modification ratios of IBM WSC

Scenario ID	Functional SA	Optimized SA	Improvement
1	8/8	8/17	52.9%
2	2/8	3/17	29.4%
3	8/8	8/17	52.9%
4	2/8	1/17	76.5%
5	1/8	1/17	52.9%

Table 9 summarizes component modification ratios of the IBM WSC. The improvement after architecture transformation exceeds **50%** in four scenarios (*i.e.,* Scenario 1, Scenario 3, Scenario 4, and Scenario 5). In Scenario 2, the component modification ratios of the three-tier software architecture designs are also lower than those of the functional software architecture designs. Hence, the modifiability of the three-tier software architecture designs is better than that of the functional software architecture designs.

Globally, the improvement rate of metric values falls in the range from 37.2% to 79.4%, and the improvement rate of the scenario based analysis estimates ranges from 29.4% to 76.5%.

We conclude that our proposed quality improvement approach effectively improves the modifiability of software architecture. Moreover, the consistency between the results of our metrics and ALMA provides a quantitative validation of these metrics.

VII. Related Work

We now discuss related literature on software architecture design and evaluation.

A. *Software architecture design*

The transformation of a requirement specification into a software architecture is a difficult challenge in the software engineering domain [4]. One part of this challenge lies in the lack of support for the derivation of software architecture from functional requirements. Functional requirements specify the problem to be addressed by a software system. Software architecture depicts the solution for resolving the problem. Due to the gap between the two perspectives, it is challenging to derive software architecture from functional requirements. A great deal of effort has been devoted to address the challenge. For example, the unified process [23] utilizes use cases to decompose functional requirements and produce software architecture. Rumbaugh et al. [24] present an object oriented modeling design approach that creates software architecture by composing objects identified from functional requirements. Hall et al. [25] use problem frames to analyze functional requirements and design functional software architecture by extending problem frames. The commonality of these approaches is that they rely on software architects to manually design software architecture following the specified process. Different from these approaches, we reuse business processes created by business analysts as requirement models and automatically derive software architecture from business processes. Our proposed approach can greatly improve the efficiency of software architecture design.

B. *Software architecture evaluation*

The fitness of a software design is often evaluated by two categories of approaches: quantitative measurement and qualitative analysis. Numerous product metrics have been introduced to quantify the quality of software designs; Fenton and Melton [18] and Dhama [19] propose metrics for measuring cohesion and coupling of software systems. Mancoridis et al. [9] present a modularization quality metric for evaluating the modularity of software systems. Blundell et al.

[14] present a collection of common quantitative metrics for measuring software design quality. However, the majority of these metrics require code level artifacts or detailed designs. Due to the lack of such low level details, quantitative measurement approaches are rarely used in the design of software architecture. Recent efforts toward software architecture evaluation are focused on using scenarios to qualitatively detect pitfalls of software architecture designs. The software architecture analysis method (SAAM) [7] is widely used to analyze various quality attributes of software architecture. The architecture trade-off analysis method (ATAM) [8] uses scenarios to aid the identification of risks in software architecture design and support design trade-offs. The performance assessment of software architecture (PASA) [26] and scenario-based reliability analysis (SBRA) [27] utilize scenarios to analyze performance and reliability related issues during software architecture design, respectively. However, scenarios are application and problem specific. Evaluators need to spend a large amount of effort in eliciting and evaluating distinct scenarios during software architecture designs. Scenario based approaches are labor intensive and time consuming [22]. In our generated software architecture, components are described using data items and tasks derived from business processes. We define metrics to automatically assess software architecture using properties of tasks and data items. Our approach reduces the cost of software architecture design by automatically generating software architecture with desired quality attributes from business processes.

VIII. Conclusion

In this paper, we present an approach that automatically generates software architecture with desired modifiability from business processes. We use tasks and data items in business processes to describe internal structures of architectural components. Moreover, we extend the traditional product metrics to measure the business process related artifacts instead of software code artifacts, and show that their evaluations are consistent with the ALMA scenario-based approach. Our approach provides a fast prototyping technique for software architecture design. It also helps software architects to automatically verify the achievement of modifiability requirements and evaluate impact of transformation rules. In the future, we aim to evaluate our proposed approach by consulting experienced software architects. In addition, we plan to extend our approach by building the goal models for other quality requirements.

References

[1] WBM, IBM WebSphere Business Modeler, http://www.ibm.com/software/integration/wbimodeler/

[2] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA), Journal of Systems and Software, vol. 69 (1-2): 129–147, 2004.

[3] F. Bachmann, L. Bass, and R. Nord, Modifiability tactics, Technical report CMU/SEI-2007-TR-002. Software Eng. Inst., 2007.

[4] J. Bosch and P. Molin, Software Architecture Design: Evaluation and Transformation. Proc. IEEE Conference and Workshop on Eng. of Computer-Based Systems, pp. 4-10, 1999.

[5] L. Dobrica and E. Niemelä, A Survey on Software Architecture Analysis Methods. IEEE Transactions on Software Engineering., vol.. 28 (7): 638-653, 2002.

[6] ISO/IEC 20926, IFPUG Functional Size Measurement Method, http://www.iso.org/iso/ catalogue_detail.htm?csnumber=51717

[7] R. Kazman, G. Abowd, L. Bass, and P. Clements, Scenario-Based Analysis of Software Architecture, IEEE Software, vol. 13(6): 47-55, 1996.

[8] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, The Architecture Tradeoff Analysis Method, Proc. 4th IEEE Int. Conf. on Engineering Complex Computer Systems, pp. 68-78, 1998.

[9] S. Mancoridis, BS. Mitchell , C. Rorres, and Y. Chen, Using Automatic Clustering to Produce High Level System Organizations of Source Code, Proc. the 6th Int. Workshop on Program Comprehension. pp. 45, 1998.

[10] JT. Nosek and P. Palvia, Software Maintenance Management: Changes in the Last Decade. Journal of Soft. Maintenance: Research and Practice, vol. 2(3): 157–174, 1990.

[11] RS. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill Higher Education, 5th edition, 2001.

[12] W. Stevens, G. Myers, and L. Constantine, Structured Design, IBM Systems Journal, vol. 13 (2): 115-139, 1974.

[13] Object Constraint Language (OCL), http://www.omg.org/spec/OCL/2.2/PDF

[14] JK. Blundell, ML. Hines, and J. Stach. The measurement of software design quality, Annals of Soft. Eng., vol. 4(1): 235–255, 1997.

[15] W. Li and S. Henry, Object-oriented metrics that predict maintainability. Journal of Systems and Software, vo. 23(2): 111–122 1993.

[16] F. Buschmann , R. Meunier , H. Rohnert , P. Sommerlad , and Mi. Stal, Pattern-oriented software architecture: a system of patterns, John Wiley & Sons, Inc., 1996

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

[18] NE. Fenton and A. Melton, Deriving structurally based software metrics, Journal of Systems and Soft., vol. 12 (3): 177-187, 1990.

[19] H. Dhama, Quantitative models of cohesion and coupling in software, Journal of Systems and Software, vol. 29 (1): 65-74, 1995

[20] J. Mylopoulos, L. Chung, and B. Nixon, Representing and using nonfunctional requirements: a process-oriented approach, IEEE Transactions on Software Engineering, vol. 18(6): 488-497, 1992.

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

[22] MT. Ionita, DK. Hammer and H. Obbink. Scenario-Based Software Architecture Evaluation Methods: An Overview. Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering, 2002.

[23] Jacobson, I., Booch, G., & Rumbaugh, J., The Unified Software Development Process, Addison-Wesley, 1999.

[24] Rumbaugh,J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. Object-Oriented Modeling and Design. Prentice Hall, 1991.

[25] JG. Hall, M. Jackson, RC. Laney, B. Nuseibeh, and L. Rapanotti, Relating Software Requirements and Architectures using Problem Frames, In Proceedings IEEE Joint International Conference on Requirements Engineering, pp. 137 – 144, 2002.

[26] LG. Williams. and CU. Smith, PASA: A Method for the Performance Assessment of Software Architecture, In Proceeding of ACM Internationl Workshop Software and Performance, pp. 179-189, 2002..

[27] S. Yacoub, B. Cukic and HH. Ammar, A scenario-based reliability analysis approach for component-based software, IEEE Transactions on Reliability, vol. 53(4), pp. 465-480,2004

[28] M. Hung and Y. Zou, Extracting Business Processes from Three-Tier Architecture, In International Workshop on Reverse Engineering To Requirements, 2005.

[29] 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.