Recovering Workflows from Multi Tiered E-commerce Systems

Maokeng Hung
ASUS Tek Computer Inc.
Yonghe City, Taipei County 234
Taiwan (R.O.C.)
alex1_hung@asus.com.tw

Ying Zou
Dept. of Electrical and Computer Engineering
Queen's University
Kingston, ON, K7L 3N6, Canada
ying.zou@queensu.ca

Abstract

A workflow is a computerized specification of a business process. A workflow describes how tasks are executed and ordered following business policies. E-commerce systems implement the workflows of the daily operations of an organization. Organizations must continuously modify their e-commerce systems in order to accommodate workflow changes. However, e-commerce systems are often designed and developed without referring to the workflows. Modifying e-commerce systems is a time consuming and error prone task. In order to correctly perform this task, developers require an in-depth understanding of multi tiered e-commerce systems and the workflows that they implement. In this paper, we present an approach which automatically recovers workflows from three tier e-commerce systems. Given the starting UI page of a particular workflow, the approach traces the flow of control throughout the different tiers of the e-commerce system in order to recover that workflow. We demonstrate the effectiveness of our approach through experiments on an open source e-commerce system.

1. Introduction

E-commerce systems provide users with convenient electronic services, such as banking, investment and purchasing. Organizations use e-commerce systems in order to automate their daily business operations. A workflow provides a specification of a business process and describes how tasks are executed, and ordered following business policies. For instance when a payment is issued, the e-commerce system must execute a sequence of tasks, following the specification of an on-line payment business process (i.e., a workflow). The workflow would contain tasks, such as "*obtain credit card information*", "*authorize credit card*", "*approve order*", "*reject order*", and "*ship order*". In addition to tasks, a workflow contains business policies (i.e., conditions) to regulate the execution of tasks. In an on-line payment workflow, the validity of a credit card is a business policy (i.e., condition) which determines the next tasks to execute. For example, the workflow would perform the "*approve order*" task, if the credit card is valid; otherwise, it would perform the "*reject order*" task.

With evolving market strategies, organizations constantly enhance their business processes in order to provide better services to their customers and to maintain their competitive edge. E-commerce systems must be updated in order to reflect technical and business changes. However, the documentation for business processes (i.e., workflows) is rarely available and if available it seldom conforms to the constantly changing and customized business processes. Moreover, e-commerce systems are often developed without referring to specified workflows. Developers asked to enhance and maintain an application, face a challenging task of manually locating and modifying the appropriate source code blocks. This challenging situation has caused the costs of system maintenance to escalate while corporate spending budgets are shrinking.

To address the above issue, we propose an approach to recover workflows from multi-tiered e-commerce software systems. In particular, we focus on three-tier software systems, including UI, Application Logic and Database. We identify the structure of workflows by starting from an initial page in the UI of the system. Furthermore, we trace the navigation flow of the UI through the different UI pages. We examine the source code of the different components that implement the functionality delivered through the UI pages. Moreover we recover information from the glue code (i.e. controller code) which integrates databases and back-end components in the three-tier architecture.

The recovered workflows are usually complex and contain many tasks. The complex structure may hinder the understanding of the system functionality. Therefore we present the recovered workflow using a hierarchical view. The view hides trivial business processing steps and gives a better glimpse of the structure of a workflow. For instance, the "*verify a credit card*" task usually consists of several simple tasks such as "*verify credit limit*", "*verify owner's name*", and "*verify owner's address*". In order to reduce clutter and to give a high-level view of a "*purchase item*" workflow, which contains a "*verify credit card*" task, we do not show the detailed simple

tasks. Instead we group the simple tasks under a sub-process called "*verify credit card*". Furthermore, our recovered workflows are imported into the IBM WBM (WebSphere Business Modeller) [12]. Developer can use the modeling tool to view and study the recovered workflow. Developers can delve deeper into a particular task in order to inspect its low-level processing tasks. We demonstrate the effectiveness of our techniques through experiments on open source e-commerce systems.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 gives an overview of the workflow recovery process. Section 4 discusses our approach for recovering high-level workflow views from UI and controller code. Section 5 presents the techniques used to recover low-level workflow views from the source code of the backend components. Section 6 presents the application of our approach on an e-commerce open source system. Finally, Section 7 concludes the paper and discusses our future work.

2. Related Work

To determine business tasks implemented in source code, Huang *et al.* [3] and Sneed and Erdos [1] define business logic code as functions with conditions, and consider 1) output and variables, 2) data flows and data dependencies, and 3) program stripping to reduce the search scope of their approach. In [4], code restructuring and use case identification were used to recover business tasks from COLBOL programs. In [1], Earls *et. al.* proposed a manual approach to recover business tasks from source code. In [1], static data and dynamic data are use for feature identification. In the above research, the authors focused on business tasks from source code and often human assistance is required to guide the identification.

In previous work, we applied static tracing techniques on the Java source code of a software system in order to automatically recover implemented workflows [4][9]. Our prior work was limited to the workflows implemented within a single source code file. For many large e-commerce systems, a multi-tiered architecture is used. The functionality of a software system is implemented and divided across various tiers and components. A component implements a collection of tasks and many components interact together in order to fulfill a business process. Simply analyzing the source code of a particular component is not sufficient.

To improve our previous work and to remove the limitation, we utilize different components in the e-commerce systems. Such components include the User Interface (UI) code, the source code of the components which implement that application logic and the databases. These components and their interaction are examined closely in order to produce a complete view of the implemented workflows.

3. Our Workflow Recovery Approach

Most e-commerce systems are nowadays constructed using a three-tier architecture, as shown in Figure 1. The architecture consists of three tiers: user interface, application logic and database tier. In the user interface tier, a user interacts with an e-commerce system by filling and submitting requests using a web browser. In the application logic tier, controllers act as integrators which coordinate the execution of different components in order to fulfill a business process. More specifically, a controller captures user requests and dispatches them to the appropriate components for processing. During the processing of a user request, a component may communicate with the database tier. Consequently, a component may retrieve data from the database, update data in the database, or return the response to the controller. Subject to the conditions required in a workflow, a controller determines the next step by either invoking another component to execute a task or request additional user input by loading another UI Page.

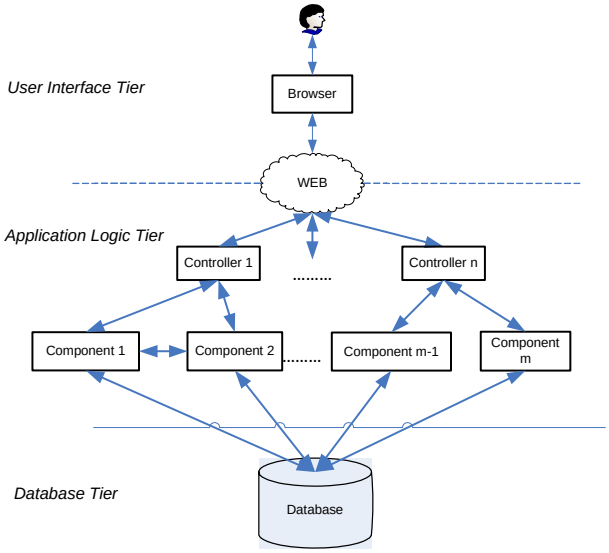


Figure 1: Data Flows in a Three-Tier Software Architecture for E-Commerce System

3.1 Analysis of E-Commerce System

In order to gain a complete and good understanding of the implementation of a particular workflow, all tiers of an e-commerce system should be analyzed. Simply studying the code of the UI pages is not sufficient. Given an on-line payment business workflow, if we solely study the UI code, we would only recover the tasks that require input from users, and miss the tasks that are directly performed by backend components. Therefore, we must trace the flow of navigation between the different UI pages and the interactions between UI pages and application logic tier in order to achieve a complete view of the workflow. Given a starting UI page of a particular

workflow, we analyze it in order to determine the navigation flow of the workflow. Many e-commerce systems are highly dynamic; therefore, subsequent UI pages are displayed by generating the new UI pages on the fly. We must also examine the controller code, which integrate various backend components, to understand the execution order of components and the execution constraints. Moreover, we need to analyze the source code of the backend components which provide the functionality required by these UI pages. In short, we must leverage all the available information sources, such as UI pages, controllers, components and database communications.

However, a complete control flow would be too large to view and of little value in improving system understanding. To address the possible shortcomings we employ these two techniques:

1) We trim the recovered control flow graph by removing all non-business related entities, such as tasks and business polices. For example, we eliminate access to utility functions. Moreover, we statically trace data reads from databases and user input data in order to determine whether a particular method is accessing business relevant data.
2) We use hierarchical views in order to provide a high-level view of the major processing steps (i.e. tasks) for a workflow, while offering access to low-level details on how a major step is achieved. The high-level view is based on the workflow information recovered from the control flows in controllers. The low-level view is based on analyzing the source code of the methods in components which are invoked by the controllers. All high-level tasks which are implemented in a component are called sub-processes and a developer can delve into these high-level sub-processes in order to understand the low-level structure of these sub-processes.

3.2 Representation of Recovered Workflows

We represent the recovered workflow using a unified format. Essentially, the recovered workflow consists of four elements.

- Tasks describe the lowest level of details needed to achieve a business function, for example, "*Check credit limit*", or "*Get credit card number*".
- Sub-processes are a grouping of interrelated simpler tasks. For example, the "*Verify credit card*" is composed of several simpler tasks, such as "*Verify credit limit*", "*Verify owner name*", and "*Verify owner address*".
- Control flows determine the execution order of tasks or sub-processes. More specifically, a set of tasks that can be executed in different orders, such as sequence, alternation and loop.
- Data flows describe the input/output of a task.

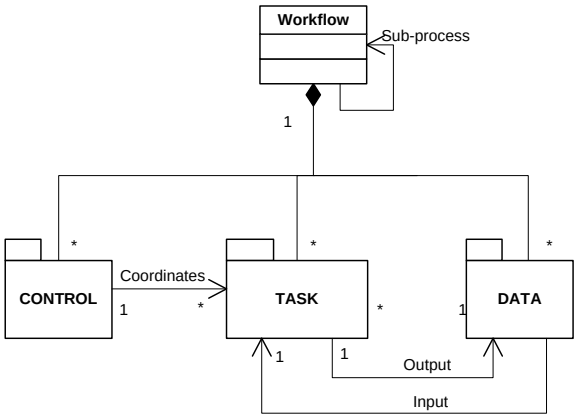


Figure 2: Schema for Representing Recovered Workflows

We store a recovered workflow in an XML document using the schema depicted in Figure 2. To visualize a recovered workflow, we convert our internal workflow format into a format used by commercial workflow modeling tools. By presenting the recovered workflows using industrial standard commercial tools, we can ease the adoption of our work in industry. In our research, we use the Eclipse-based IBM WBM (WebSphere Business Modeller) to display our recovered workflows. Figure 3 shows an example of a recovered workflow visualized using IBM WBM. The top part of the Figure shows the high-level view of the recovered "*Manage Content*" workflow. The bottom part of the Figure shows the internals of a particular sub-process (the "*Create Content*" sub-process). As depicted in the high-level view of the recovered workflow, once the "*Add Content*" task is performed, there are three alternative tasks: "*Create Content*", "*Edit Content*", and "*Update Content*". Based on our analysis of the e-commerce system, we discovered that the implementation of the "*Create Content*" and "*Update Content*" tasks resides in different methods of two backend components. Therefore these two tasks are considered as sub-processes. On the other hand, the "*Edit Content*" task is implemented as a UI page. Therefore, we render it as a task. We further analyze the backend components which implement these two sub-processes in order to produce a low-level view of the implementation of these sub-processes. The bottom part of Figure 3 shows the recovered view for the "*Create Content*" sub-process.

4. Recovering a High-Level Workflow View from the UI and Controller Code

In order to recover a high level view of a workflow, we must analyze the UI and controller code. The UI code is used to recover the navigation flow from the given initial page as a starting point of a workflow. The UI and controller code are used to recover the coordination

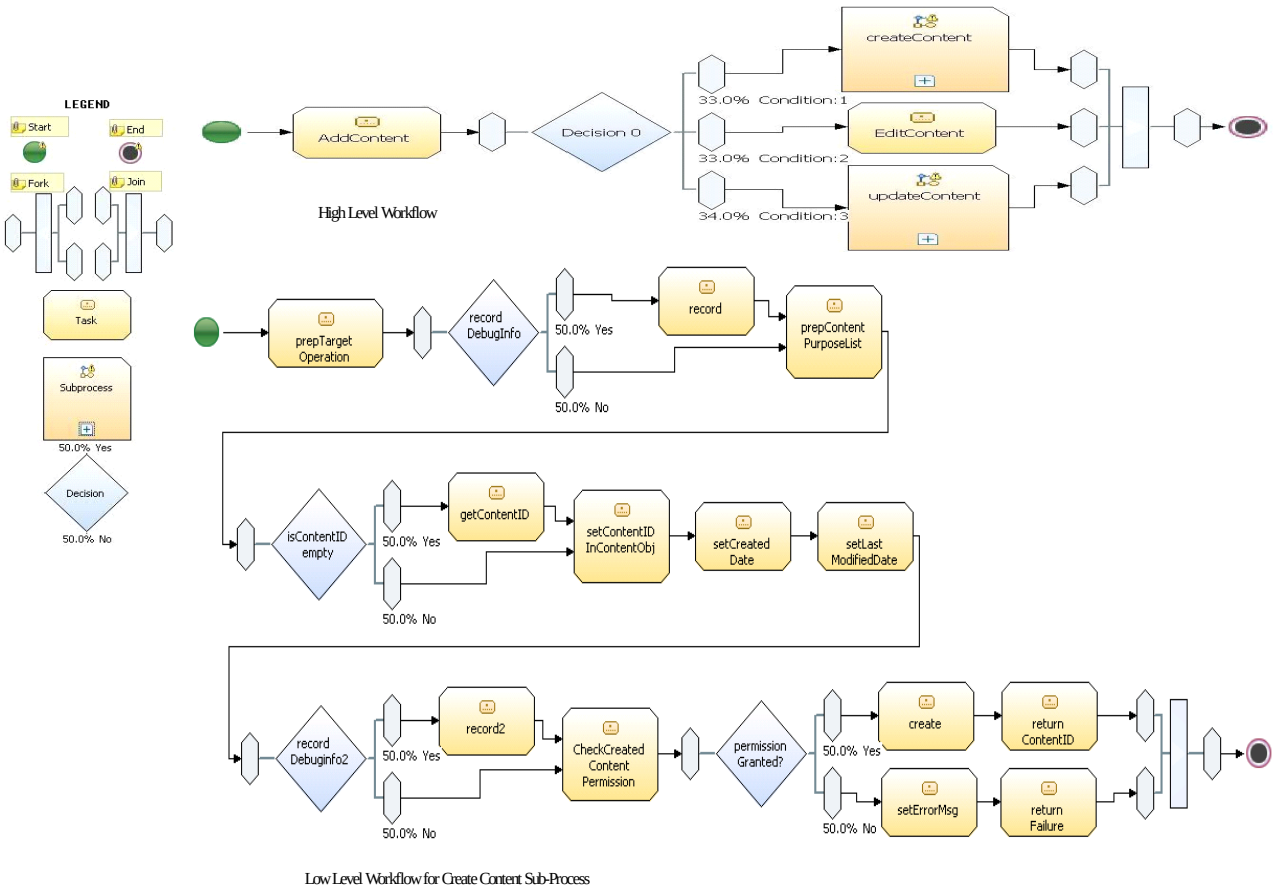


Figure 3: Example High-Level Workflow vs. Low– Level Workflow Visualized in IBM WebSphere Business Modeller

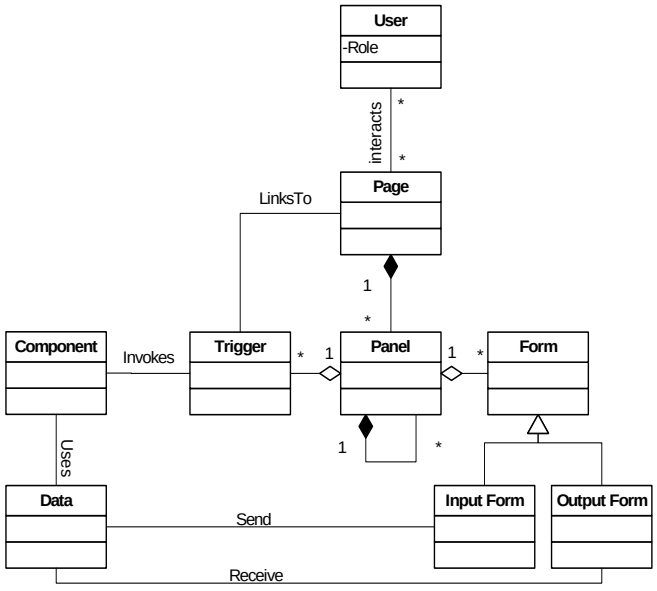


Figure 4: An Abstract Model for UI Pages

between the different tasks that are implemented in UI code and components.

4.1. Recovering the Navigation Flow from the UI

Given the starting UI page of a workflow, we can determine the controller which provides services for that page. The controller code details how the various buttons (i.e. triggers) in the UI are mapped to the backend components. Using this mapping information, we can recover the navigation flow of a user by tracing the progress of a workflow across the UI pages and their corresponding controllers. This tracing process continues until we re-visit the same page or there are no more triggers to visit. During the tracing process, we identify invoked methods in components as sub-processes. These methods take the data gathered from a UI page as input or generate results which are later shown on the page. We leverage the page names of UI buttons or hyperlinks to assign names for the recovered tasks.

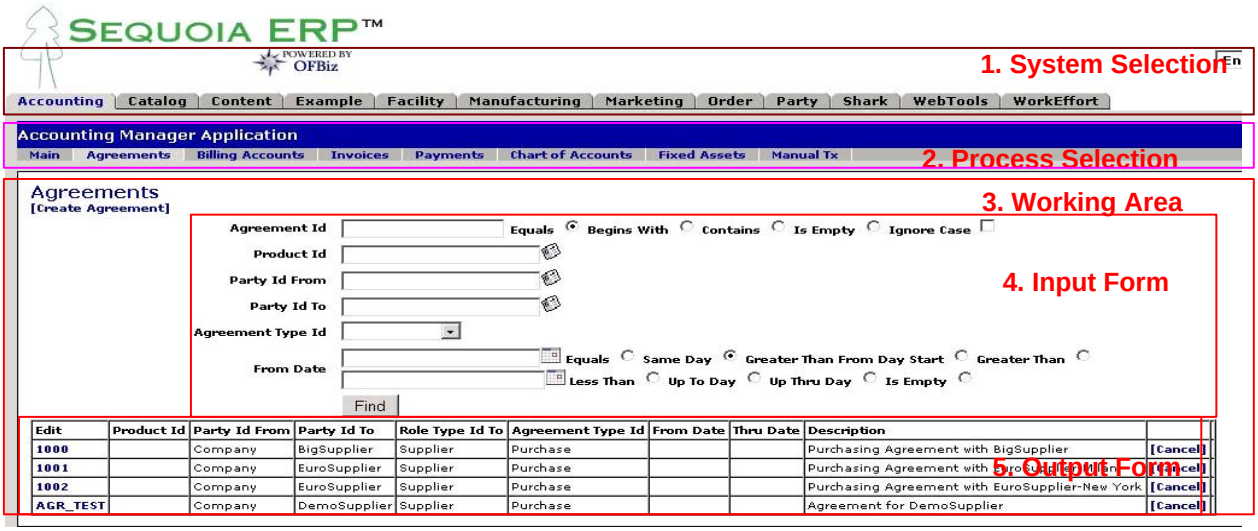


Figure 5: A Screenshot of an Accounting Subsystem in OFBiz Project

We developed an abstract model for representing the UI as illustrated in Figure 4. Figure 5 is a screenshot of a starting UI screen for the "*Manage Agreement*" workflow. The workflow is implemented as part of the OFBiz (Open For Business) project[10]. The user interface consists of a set of UI pages. UI pages are developed using web technologies. Each web page can be divided into a number of panels (i.e., screen areas), which can contain other panels or forms. Figure 5 is a screenshot of a starting UI page for the "*Manage Agreement*". As shown in Figure 5, the page contains three panels: 1) a system selection panel which allows a user to select a subsystem, such as accounting, catalog and content, to work on; 2) a process selection panel which permits a user to choose a workflow in a selected subsystem; and 3) a working area panel which contains an input form (indicated as 4 in Figure 5) and an output form (indicated as 5 in Figure 5). A form encapsulates a collection of widgets which allow a user to interact with the e-commerce system in order to conduct tasks. For example, a text field widget allows user to input strings. In addition, a user can click on a trigger, such as a button or a hyperlink. A button will invoke backend components. A hyperlink will proceed to next page. For instance as illustrated in Figure 5, a user can populate information about an agreement in the form and click on Find button as a trigger in order to invoke a backend component which will process the input data from the form. The result is displayed in an output form. For our example, this is a table indicated as 5 in Figure 5. A user can also click on a hyperlink, such as "Create Agreement", to navigate to a new page where he or she can create a new agreement.

4.2 Identifying Task Coordination

In addition to recover the tasks or sub-processes which form a workflow, we must identify the interaction and coordination between these tasks or sub-processes. Each UI page may have multiple triggers to invoke components that implement tasks (or sub-processes). The tasks (or sub-processes) can be coordinated by control flow constructs such as sequence, loop, and alternative. We determine each of the control flow constructs from UI source code, using the following heuristic rules.

SEQUENCE: To reduce the number of clicks in a page and avoid transferring small pages over Internet, a developer may group a sequence of forms into a page. Each form contains a button or a hyperlink that triggers a component or a new page. Following this design principle, we consider that widgets contained in one form contribute to a task. The sequence of the tasks is derived from the sequence of forms placed in one page.

LOOP: If a page is re-visited, a user is required to repeat the task he/she has completed. This repetition is terminated by the user. The sequence of tasks being. repeated is encapsulated in a *LOOP* control flow construct.

ALTERNATIVE: Multiple triggers, such as outgoing hyperlinks and buttons, can be displayed on one page. However, a user can select only one trigger to continue his or her work among several triggers. The selection is controlled by the widgets used in the UI. For example, a radio box can be used to allow a user to select between two shipping methods, either regular or express shipping. Moreover, each trigger invokes one component that fulfills the selected shipping method. As a result, the tasks performed by the corresponding components are considered as alternatives.

For example, as shown in the screenshot of Figure 5, this page is the initial page for "*Manage Agreement*" process and contains one trigger, a "*Find*" button for

finding an agreement. By following the outgoing hyperlink of the button, we identify that the *"Find Agreement"* task leads to a new page where a user can select from three tasks, including *"List Agreement"*, *"Find Agreements"*, and *"Edit Agreement"*. As shown in Figure 6, we show that these three tasks are coordinated by an alternative control construct. The *"List Agreement"* task displays a list of the agreements for a user to read. The *"Find Agreements"* task allows a user to find other agreements. This task leads the outgoing hyperlink to the starting UI page, *"Find Agreement"*. Therefore a *LOOP* is identified. If the user chooses the trigger which invokes the *"Edit Agreement"* task, then there are two possible alternative actions: *"Update Agreement"* or *"Create Agreement"*. The low-level details of tasks, such as *"Edit Agreement"* and *"Update Agreement"*, are recovered by analyzing the source code of the related components.

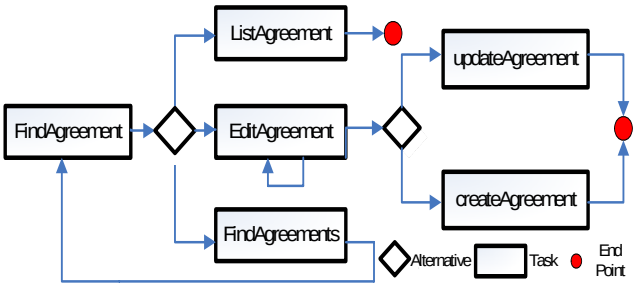


Figure 6: An Example Recovered High-Level Workflow from User Interface

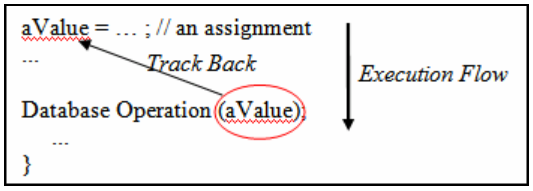
5. Identifying Business Relevant Control Flow and Tasks

It is challenging to distinguish if the identified tasks and control flows are relevant to business objectives. For example, *"Create a Shopping Cart"* task is relevant to a business operation, since this task operates on business data (i.e., shopping cart). However a method which creates a table in a database is a non-business function. Most of business data, such as catalog, product, and customer account are stored and maintained in databases. In our research, we use database accesses as criteria to detect business related tasks and control flows. To derive tasks performed in databases, we utilize data flow analysis to trace the input parameters for database invocations and result returned from database invocation.

5.1. Identifying Business Relevant Tasks

A task manipulates input data and generates outputs. Generally, business data are retrieved from and stored in the databases. If the input and output of a method are derived from business data, then there is strong evidence that the identified task is relevant to business operations. Therefore, we examine the interactions between code blocks and two types of database operations, namely fetch and update. The fetch operation retrieves data from the database and the update operation stores data in the database.

To locate the tasks that use fetch operations, we apply a static tracing technique, which analyzes the dataflow toward the direction of the execution. We analyze all the variables populated by a fetch operation, and trace them till the end of their lifetimes. Update operations store the outputs of tasks to the databases. To identify tasks that compute the outputs, we apply a backtracking technique, which analyzes the data committed to the databases, and backtracks to the uses of the data by following the reverse direction of the execution. As shown below, the database operation has an input, *aValue*, and we identify where this variable is assigned prior to this database operation. In some cases, *aValue* can be assigned multiple times, but only the final assignment determines the value of the variable that is stored to the databases. Therefore, our analysis backtracks till the last assignment of a variable (i.e., the first assignment statement in the backtracking). Moreover, we include this assignment statement as a task as it involves updating business data. The task name is taken from the name of the assignee with a prefix. We prefix is defined according to the method name and the operation the variable is involved. For example, if an assignment is the result from evaluating a Boolean typed expression, the task name is named as "*eval_aValue*".



We summarize the following usage patterns to identify tasks based on database fetch and update operations.

a) Computations which manipulate business data are considered business relevant tasks. The business data is derived from the result of database operations. The manipulation includes extracting information from business data and calculating mathematical expressions using the extracted information. For example as illustrated in the code below, the *customer* object is a business relevant object. The object contains the information related to a customer and payment methods. The *pm* variable stores the payment method extracted from a customer object. We trace the uses of the *pm* variable in the source code, and the uses of its derivations (e.g., the *discount* variable is initialized using *pm* variable). In this example, lines 1, 3, and 4 are identified as a business relevant task. The name of the task (i.e.,

cal_finalPrice) is derived from the name of assignee (i.e., finalPrice) with "cal" as a prefix since this task is involved in a mathematical calculation. The prefix can also include the name of the method which the code block belong to.

```
1. Payment pm =  customer.getPaymgnetMethod();
2....
3. discount = pm.getDiscountRate();
4. finalPrice = originalPrice * taxRate * discount;
```

Moreover, the back tracking of the database object can be used to identify tasks. For the code example shown below, *amount* is the result of the calculation at line 1. The *invoiceItem* object contains business relevant data and it uses the *set* method to issue a database update operation using the value of the *amount* variable. By backtracking, the value of the *amount* object is assigned by the assignment statement in line 1. Therefore, lines from 1 to 6 are considered as a business relevant task. Following the same naming convention, we name the task as "cal_Amount" with the name of a method as an additional prefix.

```
1. amount = ((baseAdjAmount / divisor) * multiplier);
2. ...
3. if (amount != 0.0) {
4.   …
5.    invoiceItem.set("amount", new Double(amount));
6.    invoiceItem.set("description", adj.get("description"));
```

b) Tasks can include a user defined method that takes business data as a parameter. The business relevance of such user defined methods is evident from the manipulation of business data. Similarly, tasks can be identified from a user defined method that returns business data as an object. The value of the returned object is stored in the database. The static tracing is used to identify if the method parameters and return values are related to database operations. In the high-level workflow view, the user defined functions/ methods are considered as a sub-process. The detailed low-level workflow is recovered from the body of these methods.

5.2. Identifying Business Relevant Control Flow Constructs

Control flow constructs are used to coordinate tasks, and determine the execution paths of tasks. The conditional expressions, used in if-statements and switch statements, are control flow candidates in the recovered workflow. However, not all conditional statements should be part of a recovered workflow. For instance, a conditional statement may check the initialization of a variable in order to avoid errors. Such conditional statements are specific to programming languages and are not meaningful in business domains; thus such control flow constructs should not be shown in the recovered workflow. We have developed two heuristics to recognize business relevant control flow constructs.

a) We examine if the result of evaluating a control expression determines the execution of a polymorphic method. For the example of an online bookstore, different shipping methods can be implemented using polymorphism. One polymorphic behavior allows a user to download a digital version of the book. Another polymorphic behavior ships a hardcopy to the user. Both methods have identical method names, but with different types of parameters. We treat each polymorphic method as an individual task, and the execution condition as a control construct. In the code example shown below, *order* (data object) invokes two "*addTax*" methods with different parameters in each conditional branch. As a result, the conditional expression and its derivation (postCode) are control constructs. To distinguish the two tasks generated from the same polymorphic method, we combine the method name with the parameter names. For example, the tasks obtained from the "*addTax*" method are called addTax_Interational, and addTax_National.

```
String postCode = (String) response.get("postCode");
    …
if (postCode.equals("INTERNATION")) {
    order.addTax(INTERNATIONAL);
    } else {
    order.addTax(NATIONAL);
}
```

b) To examine the meaningfulness of the candidate control constructs, we must test their relevance to business operations. Similar to filtering non-business relevant tasks, the usage of business data in expressions acts as key indictor for identifying business relevant control constructs. For example, a database fetch operation returns an object that contains business data, such as shopping cart, or an order. Similarly, an object, which stores its content using a database update operation, is considered as business relevant object. To locate business relevant control constructs, we start by analyzing the database operations (e.g., fetch and update). Furthermore, we use the data flow dependencies in order to trace the definition and uses of each business relevant object. A control flow construct is derived from a conditional expression which directly or indirectly uses information retrieved from a business relevant object. For example, the tax rate needs to be adjusted according to the region of residence. Hence, the total price of a product

is calculated using the province of the destination. In the code example below, the shopping cart object is retrieved from the database. The *order* object references a list of items placed in a buyers' shopping cart. The conditional statement at line 5 is a candidate for a business relevant control construct. By tracing the data dependencies of the *province* variable, we identify that the *province* variable is derived from the business relevant object (*order*). In other words, the *order* object is indirectly used in the conditional expression at line 5 and its value affects the execution of the code. It is safe to say that control construct at line 5 is a business relevant one.

```
1.  order = shoppingCart.getOrder(orderId);
2. ...
3.   Boolean province =
4              order.getDest("destination").equals("ON");
5.   if (province) {
6.       ...
7.      // Calculate taxes and prices.
```

6. Applying our approach on a large system

To demonstrate the effectiveness of our proposed approach, we used our approach to recover workflows from an open-source system, called *Open For Business* Project (OFBiz). It contains several subsystems that provide functionalities for Enterprise Resource Planning (ERP), Customer Relationship Management (CRM) and E-Commerce. The controller code is written as configuration files in an XML format. These configuration files specify inputs and outputs of UI pages, component and class invocations, and execution conditions for components or classes. In the OFBiz project, components are implemented in Java and a proprietary scripting language. In this case study, we select an application called Sequoia ERP [11] from the OFBiz project.

We have developed a prototype tool as an Eclipse plug-in to automatically recover workflows:

1) To recover high-level workflows, we have developed a parser to analyze the scripting code for the UI pages and the controllers.
2) To recover low-level workflows from source code written in Java, we have developed a parser to analyze the source code of the components, using Eclipse Java Development Toolkit.
3) To visualize the recovered workflows using commercial business process modeling tools, such as the IBM WBM, we have developed a converter that automatically transforms the format of our recovered workflows into the format used in IBM WBM.

In order to determine the effectiveness and accuracy of our approach, we count the number of misidentified and missed tasks in the recover workflows. We then calculate the *precision* and *recall* of our approach. We want to achieve high precision and high recall for our approach. In the following subsections, we present the results of our experiments. We first study the performance of our approach on recovering the high level workflows. We then examine the performance on recovering low level workflows (i.e. workflows within sub-processes).

Table 1: Recovered High-Level Workflows from Sequoia

Systems	# of Workflows	# of Tasks
Accounting	3	18
Content	7	34
Order	2	18
Party	3	16
Manufacturing	2	6
TOTAL	17	72

Table 2: Precision and Recall for High-Level Workflows

High-Level Workflow	# of Tasks	# of Mis-Identified Tasks	# of Missed Tasks	Precision	Recall
Add Content	4	0	0	100%	100%
Find Agreement	5	0	0	100%	100%
Find Quote	5	0	0	100%	100%
List Assets	4	0	0	100%	100%

6.1. Results for High-Level Workflows

Table 1 summarizes the results of the high-level workflows recovered from the Sequoia ERP system. We list the number of workflows we recovered from five subsystems and the total number of identified tasks. As shown in Table 1, we recovered 17 workflows from five subsystems. We recovered 72 tasks in total. As the original specification of workflows is not available, we cannot verify the total number of workflows implemented in these subsystems. However, we manually recovered the same number of workflows and tasks from the UI code and controller code. 27 tasks in total are sub-processes that are recovered from backend components. The reminder of the tasks is recovered from the following sources in the controller code:

1) Direct interaction with databases – Users enter data in the input forms, and the data is directly sent to the database without being processed by any back-end components. For instance, a user may specify a query to find information stored in the database. The database query operations are specified in the

Table 3: Evaluation of Recovered Low-level Workflow from Sequoia ERP System.

Sub-process	# of Identified Tasks	# of Misidentified Tasks	# of Missed Tasks	Precision	Recall
Create Content	13	4	2	70%	85%
Update Content	11	1	1	91%	91%
Create Content Assoc	22	4	3	82%	86%
Update Content Assoc	16	3	3	81%	81%

controller code instead of in the back-end components.

2) Direct Web page generation. A controller may create a new page for the next task in a workflow. We consider that the generated page is a task.

6.1.1. Evaluation of the High-Level Workflows

To measure the effectiveness of our approach for recovering high-level workflows, we manually evaluate the UI code and controller code using documents, the running system and the code. We manually recover the possible workflows. Such manually recovered workflows can be used to compare the workflows that are automatically recovered using our tool. We calculate the precision and recall for a set of the recovered high-level workflows.

$$precision = \frac{\#\text{ of identified tasks} - \#\text{ of misidentified tasks}}{\#\text{ of identified tasks}} \quad \textbf{(EQ 1)}$$

Precision measures the ability of approach in detecting non-business tasks and excluding these tasks from the recovered workflow. Therefore, misidentified tasks (i.e., non-business tasks) which are shown in the recovered workflow, reduce *precision*. EQ 1 is used to calculate the precision of a recovered workflow**.**

Recall measures the ability of our approach in identifying all business relevant tasks and not missing to show a business relevant task in the recovered workflow. Therefore, missed business relevant tasks reduce *recall*. EQ 2 is used to calculate recall.

$$recall = \frac{\#\text{ of identified tasks} - \#\text{ of missed tasks}}{\#\text{ of identified tasks}} \quad \textbf{(EQ 2)}$$

We manually evaluate the results against the UI and controller code in order to identify the number of misidentified tasks and the number of missed tasks according to documentation, comments and naming convention. The precision and recall of all recovered high-level workflows are 100%. Some sample results are illustrated in Table 2. As shown in Table 2, the number of missed tasks is 0 in all identified workflows, and therefore the recall is 100%.

6.2. Results for Low-Level Workflows

We recovered 84 low-level workflows from the Sequoia ERP System. We summarize the results for several recovered low-level workflows in Table 3. For instance, our approach identifies 13 tasks for the "*Create Content*" sub-process (as shown in the bottom part of Figure 3) and 11 tasks from "*Update Content*" sub-process.

We manually evaluate the results against the code in order to identify the number of misidentified tasks and the number of missed tasks according to documentation, comments and naming convention.

As shown in Table 3, "*Create Content*" has the lowest precision (70%) while other low-level workflows have precision above 80%. We review the recovered workflows and realize that most of the misidentified tasks are caused by two classes, namely *Timestamp* that records the current time and *Debug* that either checks errors or records error information. Additionally, in "*UpdateContentAssoc*", a "*returnFailure"* operation returns error message and a "*runSync*" operation performs synchronization are also incorrectly identified as tasks.

The recovered low-level workflows in Table 3 have reasonably high recalls. Through manual analysis, we realize that missed tasks are caused by methods with names similar to the names of utility functions. For instance, in the "*Create Content*" sub-process, the missed tasks are due to that our approach considers methods with "*put*" as utility methods.

6.3 Discussion and Limitations

Tasks in the high-level workflows may not be always implemented as separate components written in Java. For example, some tasks can be performed by user on a generated page or conducted inside a database.

In the best case scenario, the precision and recall should be equal to 1, indicating that all tasks are correctly identified. Most of our recovered low-level workflows have precisions and recalls ranged from 75% to 100%. The partial result is listed in Table 3. Through manually analysis of the source code, we identify the main sources for misidentifying are utility classes and operations that are used for non business relevant tasks such as 1) logging, 2) time recording, and 3) error handling. On the other hand missed tasks are often due to the following reasons:

1) We rely on database operations to determine the business relevance of tasks. Often, tasks without database access are missed during the recovery

because such intermediate tasks have no dependencies on the data from the databases;

2) We identify utility methods by the names and the packages that the methods belong to. When the names of components and the names of methods in a component have the similar naming conventions as the utility classes, our approach may misidentify such tasks as utility classes. In the future, we plan to improve our techniques for detecting utility classes.

We trace the navigation flows among pages in the UI to recover high-level workflows. However, tasks in different workflows may be associated with a common page in the user interface. Therefore, the recovered workflow may contain tasks in other workflows. In the future, we plan to examine the contextual information of navigation flows and detect the different contextual information for various workflows.

The user interfaces of e-commerce systems can be implemented in various technologies, such as HTML, JSP, Portals and XML configurations. Currently, our prototype tool supports XML and HTML based UI pages. We plan to enhance our tool to parse the UI developed using various implementation technologies. Nevertheless, the navigation flows are independent from UI implementation technologies.

In our current work, we start from the user interface to recover workflows which require the interaction of users. However, other workflows may not be initiated by a user using user interfaces. In this case, our workflow recovery approach can analyze the controller code only to recover high-level and low-level workflow views.

In this paper, the database access becomes key criteria to detect business relevant tasks. Moreover, the usages of data gathered from user interfaces in the code can be served as another indicator for detecting business relevance of an identified task when a workflow requires user interaction. In our future, we would like to examine the uses of input data from user interfaces to detect business tasks.

7. Conclusions

In this paper, we present techniques that recover workflows that contain a sequence of business tasks and their control constructs. from e-commerce systems. In addition, we consider the effect of three tier architecture. Especially, we aim to represent the recovered workflows at different hierarchical abstraction levels. To generate a high-level view of a workflow, we analyze the navigation flows in the user interfaces and controller code. To provide a low-level view of a workflow, we capture control constructs and detect tasks using data dependencies and database operations. We demonstrate the effectiveness of our approach by applying it on open source business software. We develop a prototype tool for automatic recovery of workflows. Our work integrates various sources in an e-commerce system to provide a complete view of an implemented workflow graphically. To prove the significance of the workflows generated by our prototype tool, we calculate the precisions and the recalls of the extracted workflows. Our analysis shows the precisions and recalls ranged from 75% to 100%. These numbers demonstrate the effectiveness on reducing the efforts required when software developers maintain the e-commerce systems.

In the future, we plan to examine more e-commerce systems and refine the criteria for checking the business relevance of tasks. We will as well investigate the possibility of applying dynamic tracing to recover workflows from running systems.

References

[1] G. Antoniol and Y. Gueheneuc Feature Identification: A Novel Approach and a Case Study Proceedings of IEEE International Conference on Software Maintenance , Budapest, Sept 2005.

[2] A. B. Earls, S. M. Embury and N. H. Turner, “A Method for the Manual Extraction of Business logics from Legacy Source Code”, BT Technology Journal, Volume 20, 2002

[3] H. Huang, et al, “Business Rule Extraction from Legacy Code”, in proceedings of 20th Conference on Computer Software and Applications”, 1996

[4] M. Hung and Y. Zou, “A Framework for Exacting Workflows from E-Commerce Systems”, in proceedings of Software Technology and Engineering Practice 2005

[5] D. C.C. Poo, “Explicit Representation of Business Policies”, in proceedings of Asia Pacific Software Engineering Conference, 1998

[6] J. Shao, C. J. Pound, “Extracting Business Rule from Information System”, BT Technol Journal, Vol 17 No 4, 1999

[7] H. Sneed, “Extracting Business Logic from existing COBOL programs as a basis for Redevelopment”, in proceedings of 9th International Workshop on Program Comprehension, 2001

[8] H. Sneed and K. Erdos, “Extracting Business Rules from Source code”, in proceedings of 4th International Workshop on Program Comprehension, 1996

[9] Y. Zou et al, “Model-Driven Business Process Recovery”, in proceedings of the 11th Working Conference on Reverse Engineering, 2004.

[10] http://www.OFBiz.org/

[11] http://www.sequoiaerp.org/

[12] http://www.306.ibm.com/software/integration/wbimodeler/