An Approach for Extracting Workflows from E-Commerce Applications

Ying Zou and Maokeng Hung
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, K7L 3N6, Canada
{ ying.zou, alex.hung } @ ece.queensu.ca

Abstract

For many enterprises, reacting to fast changes to their business process is key to maintaining their competitive edge in the market. However, developers often must manually locate and modify business logics in source code, in order to meet new requirements. This situation has caused system maintenance costs to escalate while budgets and corporate spending shrink. In this paper, we propose an automatic approach that recovers business processes from source code and refines them using control structure information in as-specified workflows (a workflow is a computerized representation of a business process). By using the as-specified workflows to guide our recovery, we can limit the search scope for business logics in the source code and we can locate explicit associations between artifacts in the as-specified and as-implemented workflows. Our case studies illustrate the effectiveness of this structural based business process recovery approach.

1. Introduction

Business logics can be considered as requirements and conditions on how to manipulate data in software applications [3]. As an example of business logics, a bookstore may have different rules to calculate the price of books sold based on a variety of coupons and special offers. A business process communicates the knowledge of organizational structures, business policies, and business operations. For instance, when a book is ordered, the business process consists of checking the availability of the book, restocking the inventory if needed, and validating the buyer's credit card. Typically, a workflow is a computerized representation of a business process. A workflow consists of a sequence of tasks that implement business logics (rules), control/data flows that link through tasks, participants, and resources required by tasks. A business application (e.g., an online bookstore application) implements several workflows, such as buying or preordering books.

E-commerce applications have emerged over the past few years. Unlike navigational Web sites, which mainly allow access to huge amounts of information, e-commerce applications support and execute business processes for various business scenarios, such as Business to Business (B2B), Business to Consumer (B2C), supply chains between organizations, and hosting services for an organization. Commerce servers, such as, Microsoft Commerce Server [19] and IBM WebSphere Commerce [16] are commonly adopted to host and develop e-commerce applications. As a commerce server is purchased by an organization (e.g., a store), a commerce server is usually shipped with default implementation of e-commerce applications for specific business sector (e.g., an on-line store website). Such a default e-commerce application supports standard business processes in that business sector. However, organizations may further customize the existing e-commerce applications in order to reflect their own business model.

Information Technology (IT) departments are faced with the growing challenges of supporting frequent requests for changing business processes. In particular, business logics are often hard-coded in such applications. The documentation for the original workflows may not be available or may not conform to updated and constantly changing business processes for the customized scenarios. Therefore, it is a challenging job for developers to manually locate the code blocks that implement business logics, and modify the code to meet new requirements of organizations. This situation has caused system maintenance costs to escalate while budgets and corporate spending are shrinking.

In this context, a number of researches has been carried out to recover business processes from source code [2…6]. However, most of the proposed approaches focus on extracting business logics without generating the workflows (the underling structure) implemented by applications. Therefore, the ***as-specified workflows*** (i.e., documented business processes) cannot be updated. Moreover, these approaches either analyze the code syntactically or require significant human intervention to identify the semantic meanings of business logics during the recovery process [6]. In our previous work, we proposed an approach to identify business logics from the source code based on code heuristics and to

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

extract ***as-implemented workflows*** from the source code [1]. Furthermore, we maintain the linkage between the as-implemented workflows and the code blocks that implement business logics in the as-implemented workflows. Unfortunately, the extracted business logics vary in the level of granularity. For example, some identified business logics may need to be further decomposed into multiple business logics. Other identified business logics are too trivial to be meaningful in a business sense. For example, an operation such as getTaxRate, which is used in calculating the price of a product, is too trivial as an individual business task.

As an extension to our previous work, we utilize the design information embedded in the as-specified workflows as guidance to refine the recovered as-implemented workflows. However, the as-specified workflows and the source code that implements the as-specified workflows tend to evolve independently. For example, additions/deletions of business logics in the source code may not be reflected in the as-specified workflows. Therefore, checking the name similarities between tasks in both types of the workflows may not be a reliable technique to establish the correspondences of both types of workflows. In our approach, we aim to compare the structural features of both types of workflows (i.e., as-specified and as-implemented workflows) using an intermediate behavioral model. The similarities in structure of both types of workflows can indicate relevant code fragments where business logics may be missing or over-identified. Driven by the as-specified workflows, the business process recovery is carried out automatically. Therefore, we can reduce the dependency on the human assistance and maximize the automation during a business process recovery process.

The rest of the paper is organized as the follows: In Section 2, we discuss the architecture of commerce servers. In Section 3, we present our proposed approach which uses the as-specified workflow to guide the business process recovery. In Section 4, we describe an abstract behavioral model that serves as an intermediate common ground for representing as-specified workflows and as-implemented workflows in a consistent manner. Section 5 presents the techniques to convert as-specified workflows into the intermediate behavioral model. Section 6 discusses the transformation of the as-implemented workflow into the intermediate behavioral model. In Section 7, we discuss how to perform the comparison of an as-specified workflow with an as-implemented workflow. Section 8 presents our case studies. Section 9 describes the related work. Finally Section 10 concludes the paper and describes future work.

2. Architecture of Commerce Servers

Typically, the architecture of a commerce server can be broken down into three layers, namely business commands, business objects and data access objects.

- *Business commands* act as "facades" for business processes such as "purchase a book". These commands make use of business objects to implement a business process. A business command can be also composed of other business commands. For example, "add customer" might be a business command comprised of two atomic business commands, such as "validate customer information", and "store customer information into database". Furthermore, the business commands can be organized in several categories, such as order management, promotion management, and customer management.
- *Business objects* are components that represent atomic business tasks, such as "create an order" and "add a new category". Moreover, business objects can reside in the local commerce server, or connect to third-party applications.
- *Data access objects* provide abstract data access from the business object layer to the underlying database management systems. Database management systems provide storage for business data, user profiles, and catalogs.

Workflows capture high-level abstraction of execution flows of business processes. A task represents the lowest-level of details in a workflow and its functionality is implemented using a business object. In the previous example of a "book purchasing" business process, a task, such as checking inventory, is included in the workflow. However, the detailed processing steps required for checking the inventory are omitted from the workflow. In our research, we aim to analyze code in the level of business commands and objects in order to generate as-implemented workflows which describe business processes.

3. Overview of the Approach for Structure Guided Business Process Recovery

In our previous work, we adapted static tracing to identify business logics from source code using a set of heuristics [1]. The code-heuristic based business process recovery can be effectively used in a particular domain. However, various programming styles and the use of the encapsulation principle in software design and

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

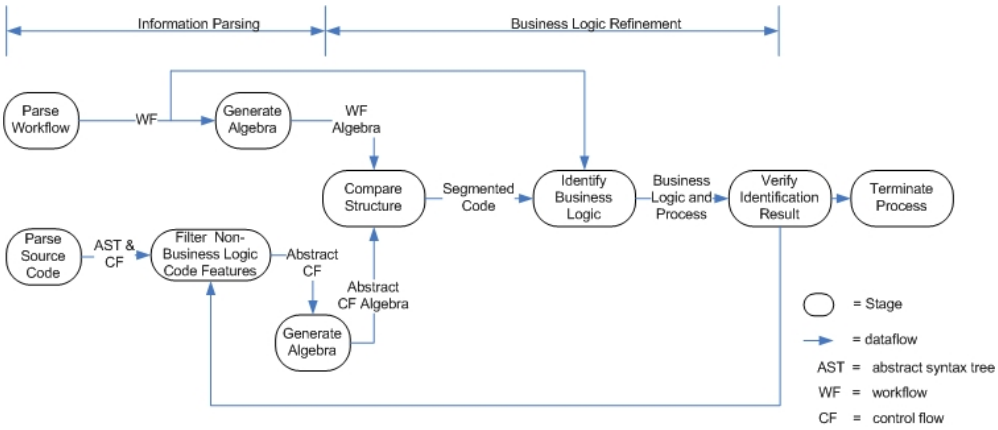


Figure 1 – Structural based business process recovery

development create challenges to extract business logics at an appropriate level of granularity without any guidance. In this paper, we leverage the control structures of as-specified workflows to derive a more precise as-implemented workflow from source code. We propose an approach which uses structural information for business process recovery. The overall steps are illustrated in Figure 1. The recovery process consists of two major stages, namely information parsing and business logic refinement.

In information parsing stage, we gather structural information from two artifacts – the as-specified workflow and the source code. Generally speaking, a workflow encodes a business process at a high level of abstraction from the perspective of a business analyst. On the other hand, source code contains the lowest level of details about a business process from a developer's point of view. To bridge the gap between them, we propose an abstract behavioral model that captures common control flow artifacts in as-specified workflows and as-implemented workflows. The behavioral model represents these control flow artifacts at an intermediate abstraction level. To achieve this, we perform information parsing in two ways, as depicted in Figure 1.

For the source code, we parse it, and generate the as-implemented workflows using code heuristics. Second, we raise the abstraction level of the generated as-implemented workflows by filtering programming specific features (e.g., nested if-statements). As a result, abstract control flows are obtained, and are further converted into our proposed behavioral model.

For the as-specified workflows, we parse the as-specified workflow represented in an XML format, extract the control flow from the specified workflow, and generate an abstract behavioral model for the as-specified workflow.

In the business logic refinement stage, we compare the behavioral models for as-specified and as-implemented workflows. Instead of measuring the structural similarities of the entire control flows, we map critical nodes in both behavioral models. We consider control constructs that affect the execution path of a workflow as *critical nodes*. For example, a decision node of a workflow is a critical node because such a decision node can generate several conditional execution paths. We collect a set of semantic equivalent critical nodes, such as decisions and loops from both types of workflows, and compare their structural similarities based on the types of the nodes, their relative locations, and orders. Once the mapping of the critical nodes between the two types of workflows is done, we can partition the as-implemented workflow into segments where each segment is bounded by the associated critical nodes of both workflows. In another words, we synchronize the as-specified and the as-implemented workflow in terms of workflow segments.

To this extent, we can refine the extracted business logics within the scope of each synchronized segment. We can then decompose identified coarse-grained business logics, or encapsulate a set of fine-grained business logics into coarser ones. Moreover, the precision of the business process recovery can be enhanced by the assistance of the data information collected from both types of workflows. In particular, we leverage the naming similarity and data flow information in the as-specified workflows to determine the boundaries between business logics in source code. Furthermore, the user can assess and verify the extracted workflow. The recovery process can loop back to filter non-business logic code, marked by the user, from the as-implemented workflows

4. Representation of Our Abstract Behavioral Model

We have developed an abstract behavioral model to represent control structures and their execution

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

behaviors for as-specified and as-implemented workflows in a unified form. We leverage the concept of Process Algebra [19] to capture the behaviors of control flows. For the purpose of comparing the as-specified and the as-implemented workflow effectively, our abstract behavioral model must satisfy the following requirements:

- Simplicity – Only essential structures should be included. Complex and unnecessary entities will complicate the comparison.
- Completeness – All semantically equivalent control structures from as-specified and as-implemented workflows should be captured.
- Generality – Various as-specified and as-implemented workflows could be ideally represented.
- Uniqueness – For any structurally equivalent workflows, there should be only one algebraic representation.
- Automatic transformation – The algebra generation from as-specified and as-implemented workflows should be performed automatically.

The grammars of our proposed algebraic behavioral model, shown in Figure 2, describe the control flows of as-specific and as-implemented workflows as derivations of the process algebra. In particular, the entire control flow of a workflow is denoted as a *process* entity, a high level initial entity which can be derived into a combination of lower level entities (e.g., non-terminals and terminals). Furthermore, the non-terminals can be broken down into other non-terminals or terminals. The derivation stops until all terminals are reached. The grammar is also recursively defined to allow a *process* entity to include other process entities.

The set of non-terminals describes the common control structures between the as-specified workflow and the control flows of source code. More specifically, the non-terminals include: 1) *process* – the high level entity that represents an entire workflow or segments of a workflow, 2) *task* – the lowest level unit of a business logic, 3) *decision* – the routing rules that a sequence follows. It is followed by either binary (Yes and No) choice. 4) *parallel* – the tasks that act simultaneous or at random order (independent of each other), 5) *sequence* – a number of tasks connected in a sequential order without any other control constructs, 6) *loop* – the repetition of a sequence of tasks.

Several operators are defined to describe the different execution relations between non-terminals or terminals. The operators are listed in Figure 2. For example, a *composition* operator (i.e., ".") describes a set of entities in sequence. An *or* operator *(i.e., "+")* describes the choices of a *Decision entity*. A *grouping* operator encapsulates a set of entities into a single one.

More complex control flows, from either as-specified or as-implemented workflows, can be modeled using the above set of terminals, non-terminals and operators in the abstract model (as discussed in Section 5 and Section 6, respectively). In such a way, we are able to reduce the granularity gap between the as-specified and as-implemented workflows.

Grammar	Non-Terminals	Terminals	Operator
$P \rightarrow S \mid T \mid L \mid D \mid \varnothing$ $P \rightarrow PL$ $PL \rightarrow P\ (\parallel P)^{+}$ $S \rightarrow P \cdot P$ $D \rightarrow$ '(' $P\ (+ P)^{+}$ ')' $L \rightarrow$ '(' P ')' '*' $T \rightarrow a \mid b \mid \ldots \mid z$ (or tasks names)	P = Process PL = Parallel S = Sequence of Task T = Task L = Loop D = Decision	∅ (empty task) Name of business logics or tasks	· = composition + = or * = repetition $^{+}$ = one or more(…) = grouping | = alternation || = parallelism

Figure 2 – Grammars for the abstract behavioral model

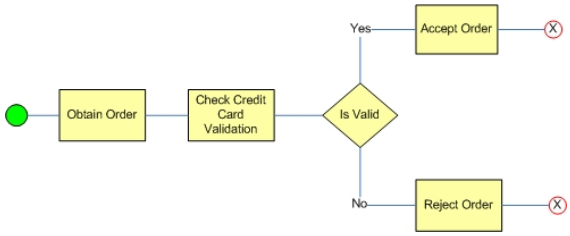


Figure 3 – A sample of the business workflow of online purchase using credit card

$$Workflow = A \cdot B \cdot (C + D) \quad \ldots\ldots\ldots\ldots\ldots\ldots (EQ\ 1)$$

Obtain Order	*A*
Credit Card Validation	*B*
Accept Order	*C*
Reject Order	*D*

Table 1 – The symbols for tasks in EQ1

As the example depicted in Figure 3, the workflow consists of two tasks in sequence, followed by a decision with two branches; and each branch contains one task. Utilizing the grammars specified in Figure 2, this workflow could be presented in simple text format as shown in EQ1 (the meaning of each operand is listed in Table 1). As depicted in EQ1, the workflow is composed by *Task A, Task B and a grouping* in sequence. In addition, the *grouping* contains *Task C* and *D* using an alternation operator.

5. Workflow Structure

Generally, workflows are represented in various specification languages, such as BPEL (Business Process Execution Language) or XPDL (XML Process Definition Language). Workflows can be modeled using modeling tools, such as IBM WebSphere Business Integration Workbench and IBM WebSphere Business Integration Modeler. We infer a meta-model that is

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

independent of a particular workflow specification language, as depicted in Figure 4. There are three types of constituents contained in a workflow: **tasks** that define how the work is actually done; **control flow** that defines sequence of task execution depending on control flow constructs, such as *decision*, *parallelism*, *alternative, loop* or *start/exit/goto* conditions; and **data flow** that specifies the input and/or output of a task.

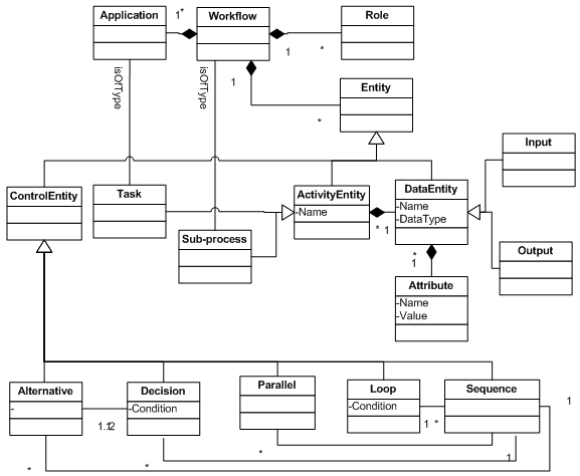


Figure 4 – Meta-model for workflow specification languages

5.1 Mapping Workflows to Abstract Behavioral Model

We are interested in inferring control flow information from a workflow and representing it in an intermediate format (i.e., our proposed abstract behavioral model), so that we can directly compare the as-specified workflow with the as-implemented workflow. Many of the control constructs of workflows, such as, *Decisions* and *Loops*, have functional equivalent counterparts in the control flow syntax of programming languages.

An *alternative* entity consists of multiple choices in a single *decision* entity, as shown in Figure 5. In a programming language, an *alternative* entity can be implemented using three types of statements, including 1) multiple nested *if-else* statements, 2) *if-elseif-else* statement and 3) *switch* statement. Essentially, these three types of statements are semantically equivalent. As a result, we can model the *alternative* entity in a workflow as nested-binary decision entities, as shown in Figure 5. There are also other possible nested-binary decisions that can be transformed with respect to the condition expression (e.g., six different possible ones in Figure 5); nevertheless, only one structure can be formed. In our research, we are primarily interested in the workflow structures. Therefore, this transformation allows us to have a unique structural representation to describe an *alternative* entity in the abstract behavioral model.

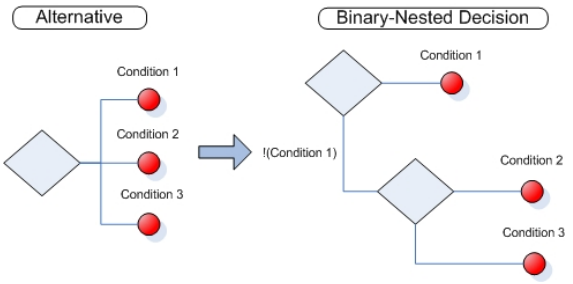


Figure 5 – Transformation from *Alternative* to nested-binary decisions

Parallelism occurs as tasks can be done in random order, such as collecting multiple data or outputting to the different destinations. For instance, both customer and inventory departments must be informed when an ordered item is out of stock in an e-commerce system. This could be implemented using concurrent programming threads or in sequence at random orders if the *parallelism* simply implies the independence between the business operation paths. In the former case, Java threads run as parallel business processes or tasks. However, the parallelism notation in a workflow is often not directly mapped to a control flow construct in source code. From our experience, parallel tasks are often implemented in sequential orders in source code. In this case, there are no data dependencies among these tasks.

Workflow Entity	**Entity in Abstract Behavioral Model**
Decision	*Decision*
Loop	*Loop*
Business logic	*Task*
Data	Attributes of *Task*
Constraint	Attributes of *Task*
Parallelism	*Parallel*, but can be treated as Sequence
Alternative	Binary-Nested *Decision*

Table 2 – The correspondence of the entities between the workflow and abstract behavioral model

Task entities representing business logics may take input data, generate output and change system states according to the input data. We can associate input data and output data as attributes of a *Task* entity. Moreover, a Task entity can also have constraints, such as timing and its execution privileges. Neither data nor constraints affect the control flow structure of a workflow; thus the attributes and constraints are not included in the abstract behavioral model. Table 2 summarizes the workflow control flow entities and their corresponding entities in the abstract behavioral model.

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

6. Abstract Control Flow Structure

The control flow of source code, extracted using static tracing, often contains programming language specific control constructs. To reduce the complexity of the structure of the control flow and lift its abstraction level, it is crucial to remove source code artifacts that have no contributions to business logics. We propose an abstract control flow graph to represent a simplified version of the detailed control flow graph emitted from compliers. We also filter irrelevant functionality, and merge structural details into a higher level of description. We discuss the details in the following subsections.

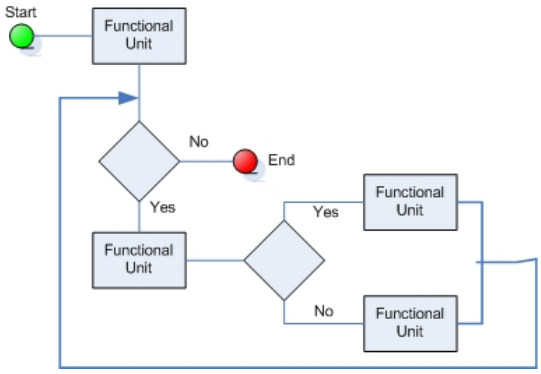


Figure 6 – An example for an abstract control flow with a loop and a decision

6.1. Abstract Control Flow Graph

The abstract control flow graph contains primitive control flow constructs and functional units, as the example shown in Figure 6. The functional unit is a code fragment that contributes business logics in the execution trace. It might contain other control constructs in the control flow where these control constructs do not contribute to business logics. Primitive control flow constructs include loops (e.g., *for*-loop, *while*-loop and *do-while*-loop), and decisions (e.g., *if-else*, *if-esleif-else* and *switch* statement). In this context, the three variations of loops (i.e, the *for*-loop and *do-while*-loop have the equivalent *while*-loop representations) can be modeled using the process algebra *loop* entity. As discussed in Section 4, the three types of *decisions* (e.g., *if-else*, *if-esleif-else* and *switch* statement) can be modeled using one representation, which corresponds to the *decision* entity in the algebra. Other programming language concepts, method invocations and error handlings, indicate changes in execution paths. In this sense, we convert these concepts into their corresponding primitive control flow constructs.

A method invocation is a routine call that jumps to a predefined location, executes a sequence of statements and resumes from the original positions after the completion. Static tracing traverses the control execution path during the method invocation. Therefore, method invocations can be represented directly as a sequence of primitive control flow constructs.

Error handling is important to build robust software. Once an expected or unexpected error occurs, the system quits the current execution and jumps to a specific location where the error can be handled and system state can be recovered, for example, rollback to previous system state if a transaction is failed. Error handling is not normally described in business processes since it is the responsibility of developers to anticipate programming specific errors. However, the error handling in source code may encapsulate other decision branches. As a result, we examine the code blocks inside *try-catch* clauses.

In summary, the proposed abstract control flow graph contains *decision, loop,* and *functional unit*. Table 3 summarizes the correspondence between constructs in abstract control flows and the entities in abstract behavioral model.

Constructs in Abstract Control Flow	**Entity in Abstract Behavioral Model**
Decision (*if-else*, *if-esleif-else* and *switch*)	*Decision*
Loop (*for*, *while* and *do-while*)	*Loop*
Functional Unit	*Task*

Table 3 – The correspondence between the abstract control flows and the abstract behavioral model

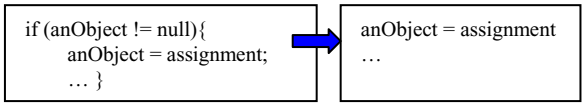
6.2. Filtering non-Business Logics

To reduce structural complexity of an abstract control flow graph, we further filter irrelevant functionality or merge detailed control constructs into a high-level representation. Non-business logics include the Utility Class, Java Class, Exception Class, and trivial methods (such as getters and setters). Their filtering has no impact on the recovered business logics. For example, the Utility classes, such as transaction manager, transaction commitment, and file I/O operations, perform a set of internal basic operations. Their functionality is served as foundations to build other methods. Therefore, the functionality of Utility classes is reflected in other higher grained methods that use these Utilities classes.

There are many programming-specific features particular to information system domain, for example, object initialization check and *null* value check. The checks complicate the control flow but have no meanings to business processes. Once identified, we remove the decisions with such checks from the abstract control flow graph. As shown below, the condition

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

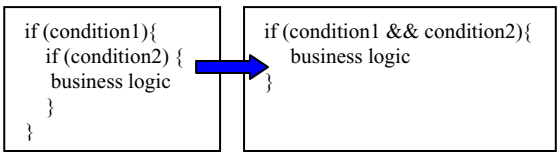
check against the *null* value is considered nonexistent in the control flow.



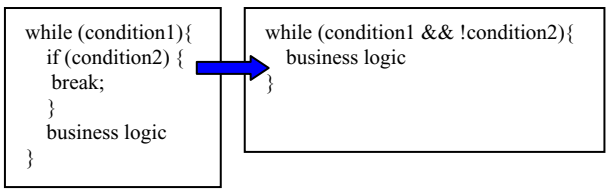
6.3. Merging Decision Control Constructs

Nested decision control constructs often contain other decisions in the code body. For example, an if-statement may encapsulate several other if-statements in its if-body part. We aim to restructure nested decision constructs, and merge their conditional expressions. In this way, we can reduce the complexity of the abstract control flow graphs and ease the comparison difficulties. Furthermore, we strive to avoid altering the overall structures and the semantics of the program during the merging of nested decision control constructs. While there are a number of methods to alter control constructs without altering their semantics, we consider the following two of the most straightforward types of nested decision control constructs.

Nested decisions: A set of nested if-statements, each of which has only the if-body part, may occur in control flow graphs. An example is illustrated below. After filtering non-business logics, the code blocks that are irrelevant to business logic are removed from the outer layers of the if-statement. Only the inner if-statement has code blocks that implement business logics. In this case, we can merge the condition expressions in outer layers of if-statements into a single layer of conditional expressions. In this example, we merge the two if-statements into one statement in the abstract control flow graph. This is also applied to nested loop-statements.



Break statement in loop: Business logics appear outside the inner decision that contains the *break* statement, as illustrated below. We can merge the conditional expressions of the while and if-statements into one loop condition expression. This transformation has no impact on the business logics.



7. Structural Comparison between Workflows

7.1. Algorithm for Structural Comparison

By utilizing the algebra and the grammars discussed in Section 3, we can generate two algebras for the control flows from an as-specified workflow and an as-implement workflow respectively. Instead of matching each element in the two algebras, we define critical points that serve as initial points for structural similarity. From the structural point of view, we consider *Loop, and Decision* in the control flow constructs as critical nodes, because of their syntactic structural meanings. *Task* and *Process* might be also treated as critical nodes if the Task or Process entities of two algebras are matched using naming conventions.

```
Procedure FindCorrespondence (ai_node, as_node) {
 // ai_node is an algebraic node in as-implemented workflow
 //  as_node is an algebraic node in as-specified workflow
    FOR (ai_node and as_node node have more sub-nodes){
   Traverse both algebras until a critical node of the same type
    Break the ai_node into two segments, ai_sub_l and ai_sub_r
   Break the as_node into two segments, as_sub_l and as_sub_r
  If isTerminal(ai_sub_l&&ai_sub_r) or  isTerminal(as_sub_l&&as_sub_r)
         Return;
  If  isTerminal(ai_sub_l) and isTerminal(as_sub_l)
        ai_sub_l and as_sub_l are matched
        FindCorrespondence(ai_sub_r, as_sub_r)
        Break;
  If isTerminal(ai_sub_r) and isTerminal(as_sub_r)
        ai_sub_r and as_sub_r are matched
        FindCorrespondenace(ai_sub_l, as_sub_l)
        Break;
  If  ai_sub_l , ai_sub_r, as_sub_l and as_sub_r are non-terminals
        Continue;
    } ENDFOR
}
```

Figure 7 – Algorithm to match the as-specified (workflow) and as implemented (CFG) algebras

Figure 7 lists the proposed algorithm for comparing the structural differences in the two types of algebras. Accepting the two preprocessed algebras, the algorithm identifies the critical nodes and compares the two algebras. This is a recursive algorithm, as depicted in Figure 7, and works as follows: it walks through both algebras until it matches the candidate critical nodes of the same types, such as *Loop* and *Decision* entities, and partitions the algebras into segments at the point of the matched critical nodes. The segments are treated as pseudo-tasks, consisting of one or a sequence of business logics. These segments of the as-implemented workflows between the matched critical nodes are then linked to the corresponding entities in as-specified workflows. In the case of ai_node and as_node are not matched (for example, one contains two branches and the other contains only one), we advance to the next

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

ai_node in the as-implemented workflow and look for another probable matching.

Original	CFG = f · (g + (h + i))*	WF = a · b · (c + (d + e))*
Grammar	CFG → f · L_{CFG1} L_{CFG1} → (C_{CFG1})* C_{CFG1} → g + C_{CFG2} C_{CFG2} → (h + i)	WF → a · b · L_{WF1} L_{WF1} → (C_{WF1})* C_{WF1} → c + C_{WF2} C_{WF2} → (d + e)
Resulting Mappings	f ↔ a · b g ↔ c h + i↔ d + e	

Table 4 – An example of the comparison algorithm

Table 4 demonstrates a sample run of this algorithm. We list the structural similarities between structural entities from the as-specified workflow (i.e., WF in the example) and the structural entities from the as-implemented workflow (i.e., CFG). The comparison algorithm is performed as follows:

- The structural comparison algorithm starts from the initial process entities (i.e., CFG and WF) of both algebras. Each algebraic expression contains *Loop* entity as a critical node. The *Loop* entity breaks each algebraic expression into two segments. For example, CFG contains segments f and L_{CFG1}.
- f in CFG and a · b in WF are mapped to each other. There are no critical nodes, and no non-terminal entities in f or a · b. The structural comparison is terminated in this segment.
- L_{CFG1} in CFG and L_{WF1} in WF are mapped. L_{CFG1} can be derived as a repetition of C_{CFG1}, a non-terminal entity. Similarly, L_{WF1} can be denoted as a repetition of C_{WF1}, a non-terminal entity. The structural comparison is further applied to non-terminal entities, C_{CFG1} and L_{WF1}, and iteratively compares structural similarities between C_{CFG1} and L_{WF1}.
- A critical node, *Decision* entity (i.e., +), exists in C_{CFG1} and L_{WF1}. Therefore, *Decision* entity separates each of C_{CFG1} and L_{WF1} into two segments respectively.
- g and c are mapped to each other.
- C_{CFG2} and C_{WF2} are marked as mapped, and the structural comparison algorithm continues to apply for the non-terminal entities.
- h + i and d + e are mapped to each other. The two algebras contain terminals and operators that are structurally similar. It is worth to mention that mappings between terminals are not considered in our structural comparison, for the reason that h could be mapped to d or e, and i could also be mapped to d or e. Nevertheless, there is no structural ambiguity for this pair of algebras.

The output of the algorithm consists of a mapping from workflow entities to source code implementation segments. The mapping is established based on the structural similarities. However, it is often difficult to identify a one-to-one relationship solely based on the structures. The ambiguous mappings include one-to-many, many-to-one, many-to-many, or cross-reference mappings. This can be resolved by comparing the data flows in the two types of workflows. Even though the one-to-one relationship may not be achieved, the searching scope for the business logics and input/output can be limited, and precise. Instead of scanning the entire systems for business logics, only the structurally similar blocks are traversed. Furthermore, code heuristics [1] and the proposed structural based techniques can be recursively used to identify more detailed business logics within a task.

8. Case Studies

To demonstrate the effectiveness of our proposed technique to recover business processes from e-commerce applications, we performed case studies using two subsystems in IBM WebSphere Commerce (Order Management and Member Management subsystems). The as-specified workflows shipped in IBM WebSphere Commerce Server are documented online at [18]. An example workflow is at [23]. As discussed with the developers from the e-commerce development team at IBM Canada, each as-specified workflow is implemented by one or more business commands in the e-commerce server. The source code for each business command is evolved and enhanced over time to adapt to new requirements. Unfortunately, the documentation (i.e. as-specified workflow) is not updated to reflect the code changes.

Subsystems	Member Management	Order Management
Num. of As-Implemented Workflows	27	63

Table 5: Number of as-implemented workflows extracted from two subsystems

The as-specified workflows, used in our case study, are modeled using IBM WebSphere Business Integration (WBI) Modeller in Eclipse environment. The as-specified workflows are stored using the schema of XMI and EMF (Eclipse Metamodel Framework) in IBM WBI Modeller. Moreover, workflows described in various specification languages can be imported into IBM WBI Modeller. We developed a parser that translates as-specified workflows stored in the schema of XMI and EMF into our behavioral model. In our case study, we analyzed business commands and objects to generate as-implemented workflows. We then compared our extracted as-implemented workflows to the documentation (i.e. as-specified workflow). Table 5 summarizes the number of as-implemented workflows extracted from each subsystem. Developers verified the results of seven as-implemented workflows and generated mappings. Table 6 lists the percentage of tasks in the as-specified workflows, which our

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

technique mapped correctly to tasks in the as-implementation workflows. In our case studies, the mappings are established by considering the critical nodes, and positions of tasks relative to the critical nodes, data input/output, and naming conventions of tasks in both types of workflows. As shown in Table 6, the percentage for the first three as-specified workflows is relatively low. A manual analysis of the source code by the developers reveals that this is due to missing tasks that are not implemented in the source code.

As-Implemented Workflow	As-Specified Workflow	Percentage of Identified Tasks
PickBatchGenerateCmd	Create pick batch	20%
ReleaseShipConfirm	Fulfill orders	33%
ReturnItemComponentDisposeCmd	Perform disposition	67%
ProcessBackordersCmd	Process backorders	100%
ReleaseToFulfillmentCmd	Release items to fulfillment	100%
PaySynchronizePMCmd	Process pending payment auths	100%
ReleaseExpiredAllocationsCmd	Release expired allocations	100%

Table 6: Comparisons between as-implemented workflows and as-specified workflows

Discussion: Comparing the naming similarities may not be feasible when workflow specifications and the development of business commands are independent. In particular, the as-implemented workflows may contain programming specific names for business tasks. This may not be easily understandable to business users. Our structural comparison technique provides an efficient indication of the boundaries of the matching among both types of workflows. Furthermore, we maintain the linkages among as-implemented workflows and code blocks in the source code and the linkages among both types of workflows. The changes in business requirements can be easily located in the source code through such linkages.

Some tasks in business processes are designed to be manually preformed by users. Therefore, manual tasks are not implemented in the e-commerce applications. In this case, we need to specify the manual tasks in the as-specified workflows.

When as-specified workflows are completely out of date from the code, it may be desirable to depend only on code heuristics instead of using the structural information in the as-specified workflow to extract business processes and rebuild documentation. In the subsystems we analyzed, we haven't found parallel tasks defined in the as-specified workflows. We plan to examine other systems and develop techniques to identify workflow parallelism in source code.

9. Related Work

This research is related to another three research areas:

Business Logic Identification. To determine the business logics inside the source code, Huang *et. al.* [2] and Sneed and Erdos [3] define business logics as functions with conditions, and consider 1) output and variable, 2) data flow and dependency, and 3) program stripping to reduce the searching scope in the source code. In [4], code restructuring and use case identification were used to extract business logics from COLBOL programs. Earls *et. al.* proposed a manual approach to extract the business logics from the source code and showed the effectiveness of human identification [5]. As a result of the above researches, human assistance is required to guide the identification. In our work, we utilize the documents, such as workflow diagrams and entity information, to extract business logics at the appropriate granularity level while minimizing the need for human intervention during the recovery process.

Traceability between Documentation and Implementation. The reflexion model reconciles high level models with high level code dependencies, in contrast our approach links low level source code entities and dependencies to business workflows entities [7]. Ivkovic and Kontogiannis introduce a methodology to trace changes to the software model using model synchronization [17]. Meta-models are defined from the source code and the documentation to establish the traceability between these two meta-models. Marcus and Maletic have used latent semantic indexing to establish the traceability links of the code with the external documents [9]. Chen and Rajlich present a manual search process to map the domain knowledge to the implementation using abstract system dependence graph [11]. Automated tools update the dependence graph while it is the developer's responsibility to make decision on the next node to visit in the graph. In our work, we compare the structural properties of the two models and establish the mapping between the critical nodes in order to limit the search scope for business logics.

Architecture Recovery. Bowman *et. al.* suggest the need for automated tool combined with human interaction to reverse engineer the architecture for a large but undocumented system before it becomes unmaintainable [10]. Several researchers proposed methods to recover architecture by identifying feature models [14, 15, 22]. The features can be extracted from informal information using concept and cluster analysis, and system expert can selectively filter the non-related

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

features, and ascertain the traceability between features and the architecture. Our work focuses on generating the workflow used by both the business and technical staffs for evolving business processes.

10. Conclusion

In this paper we propose an approach to recover as-implemented workflows from source code. We utilize as-specified workflows modeled by business analysts to guide the recovery process. By extracting control constructs from the as-specified workflow and source code, we can represent control constructs from as-specified and as-implemented workflows in a unified manner. A structural comparison algorithm is used to identify the structural similarities between both types of workflows, and to associate code blocks with workflow tasks in the as-specified workflows. This linkage limits the scope of searching for business logics in the source code and provides an explicit separation between business logics. Our case studies illustrate the effectiveness of the structural based business process recovery approach on large commercial code. In the future, we plan to study the e-commerce systems using workflow engines and examine our workflow extraction approach on such systems.

Acknowledgement

We would like to thank IBM Centers for Advanced Studies at IBM Toronto Laboratory for their valuable contribution to this paper.

References

[1] Y. Zou, et al, "Model-Driven Business Process Recovery", in Proc. of the 11th Working Conference on Reverse Engineering, 2004

[2] H. Huang, et al, "Business Rule Extraction from Legacy Code", in Proc. of 20th Conference on Computer Software and Applications", 1996

[3] H. Sneed and K. Erdos, "Extracting Business logics from Source code", in Proc. of 4th International Workshop on Program Comprehension, 1996

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

[5] A. B. Earls, et.al., "A Method for the Manual Extraction of Business logics from Legacy Source Code", BT Technology Journal, Volume 20, 2002

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

[7] G. Murphy, et al. "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models", FSE '95, October 1995.

[8] F. Balmas, "Using Dependence Graphs as a Support to Document Programs", in Proc. of the Second IEEE International Workshop on Source Code Analysis and Manipulation

[9] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", in Proc. of International Conference on Software Engineering, 2003

[10] I. Bowman, et.al, "Linux as a Case Study: Its Extracted Software Architecture", in proceeding of 21st International Conference on Software Engineering, 1999

[11] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph", in Proc. of the 8th International Workshop on Program Comprehension, 2000

[12] G. Kiczales et al, "Aspect-Oriented Programming", in Proc. of the European Conference on Object-Oriented Programmin (ECOOP), 1997

[13] T. Eisenbarth et. al. , "Static Trace Extraction", in Proc. of 9th Working Conference on Reverse Engineering, 2002

[14] I. Pashov, et. al., "Supporting Architectureal Restructuring by Analyzing Feature Models", in Proc. of 8th European Conference on Software Maintenance and Reengineering, 2004

[15] I. Pashov and M. Riebisch, "Using Feature Modeling for Program Comprehension and Software Architecture Recovery", in Proc. of 10th IEEE Symposium and Workshops on Engineering of Computer Based System, 2003

[16] Architecture Overview of IBM WebSphere Commerce, http://www.redbooks.ibm.com/abstracts/tips0247.html?Open

[17] I. Ivkovic and K. Kontogiannis, "Tracing Evolution Changes of Software Artifacts through Model Synchronization", in Proc. of 20th IEEE International Conference on Software Maintenance.

[18] http://publib.boulder.ibm.com/infocenter/wchelp/v5r6/topic/com.ibm.commerce.business_process.doc/concepts/processOrder_management.htm.

[19] Microsoft Commerce Server Technical Overview, http://www.microsoft.com/technet/prodtechnol/comm/commerce-server-technicaloverview.mspx

[20] J.C.M. Baeten, "A Brief History of Process Algebra", available at http://www.spatial.maine.edu/~worboys/processes/baeten%20history%20PA.pdf.

[21] A. Nigam and N.S. Caswell, "Business Artifacts: An Approach to Operational Specification", IBM Systems Journal, Vol. 42, No. 3, 2003.

[22] R. Koschke, *Atomic Architectural Component Recovery for Program Understanding and Evolution,* Ph.D. Thesis, Institute for Computer Science, University of Stuttgart.

[23]http://publib.boulder.ibm.com/infocenter/wchelp/v5r6/topic/com.ibm.commerce.business_process.doc/concepts/processProcess_backorders.htm

0-7695-2601-2/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY