A Framework for Exacting Workflows from E-Commerce Systems

Maokeng Hung[1] and Ying Zou[2]
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, K7L 3N6, Canada
alex.hung@ece.queensu.ca[1], ying.zou@queensu.ca[2]

Abstract

For many organizations, reacting to fast changes in customers' requirements becomes keys to maintain the competitive edge in the market. However, developers must locate business logics manually in source code in order to add new requirements. This problem has caused maintenance costs to escalate while budgets and the corporate spending are shrinking. In this paper, we propose an automatic approach that extracts workflows from sources code and utilizes control structure information in as-specified workflows to refine the recovered as-implemented workflows from source code. By using the as-specified workflows to guide our extraction, we can limit the searching scope for business logics and locate explicit associations between artifacts in the as-specified and as-implemented workflows.

1. Introduction

Business logics can be considered as requirements and conditions on how to manipulate data in information systems [3]. As an example of business logic, a bookstore may have different rules to calculate the price of books sold based on a variety of coupons and special sales conditions. 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 books, restocking the inventory if needed, and validating the buyer's credit card. Typically, a workflow is a computerized representation of a business process that consists of a sequence of tasks that implement business logics (rules), control/data flows that link through tasks, participants, and resources required by tasks.

In order to achieve dramatic improvements in critical measures of performance, such as cost, quality, and speed [7], organizations change continuously and constantly to reflect the rethinking and redesign of business logics and organizational structures in business processes. Information Technology (IT) departments are faced with the growing challenges of supporting frequent requests for changing business logics. For many IT departments, the challenge is compounded by complex architectures and distributed computing infrastructures that were developed over the last 10-20 years. In particular, business logics are often hard-coded in such systems. The documentation of workflows is often lost, obsolete or never existed. Therefore, it is a challenging job for developers to manually locate the code blocks that implement business logics, and to modify the code to meet new requirements. This situation has caused system maintenance costs to escalate while budgets and corporate spending are shrinking.

In this context, a number of research 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. Consequently, 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 extraction process [6]. In our previous work, we proposed an approach to identify business logics from the source code based on code heuristics and to extract ***as-implemented workflows*** from the source code using Abstract Syntax Trees (ASTs) [1]. 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, while others may require to be merged.

The objective of this paper is to improve the precision of the workflow extraction and to reduce the requirement of the human intervention by utilizing the design information from the as-specified workflows, which serve as guidance to refine the extracted as-implemented workflows. A behavioral model is proposed to compare the structural and functional behaviors of these types of workflows (i.e., as-specified workflows and as-implemented workflows). The similarities in structures and behaviors of both types of workflows can indicate relevant code segments where business logics may be missing or over-identified.

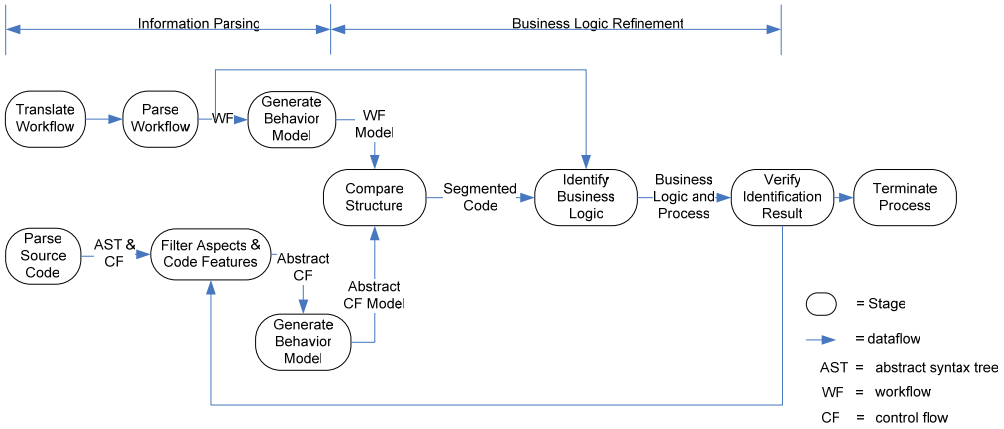

Figure 1 - The recovery consists of two major stages, information parsing and business logic refinement

2. Approach for Structure Guided Workflow Extraction

2.1 A Framework for Workflow Extraction

In most e-commerce systems, the workflows are implemented and executed without interacting with workflow engines and involving few human tasks. In such systems, software developers implement the workflow processes and their sub-processes in one system [1, 10]. Specifically, tasks are implemented as classes and methods. The business rules are hard-coded as the control and data flows in the source code. When updates are required, developers have to inspect the source code and determine the locations where changes must be made; it is our objective of this research to extract the workflows and business logics from the source code of the e-commerce systems to facilitate code updates for changing business requirements

Conceptually, workflows convey the execution of an application. An as-specified workflow can be considered as a high-level description of the control flow inside an application. On the other hand, an as-implemented workflow is extracted from the low-level implementation that realizes the as-specified workflow. Therefore, we can leverage the control structures of as-specified workflows to derive more precise as-implemented workflows at the appropriate abstraction. We proposed an approach that uses structural information for extracting workflows. The overall steps are illustrated in Figure 1.

In information parsing, we gather structural information from two artifacts – the as-specified workflows and the source code, as illustrated in Figure 1. First we generate the as-implemented workflows from the source code. Second, we raise the abstraction level of the generated as-implemented workflows by filtering programming specific features that are not in the business domain (e.g., checking object initialization). Third, we translate the workflows described by workflow description languages to as-specified workflows. Last, we convert both of the as-implemented workflows and as-specified workflows to an abstract behavioral model, which represents the control flow artifacts at an intermediate abstraction level and bridges the gap of the difference between the two artifacts.

In the stage of business logic refinement, we compare the behavioral models from as-specified and as-implemented workflows. We consider control constructs that affect the execution path of a workflow as critical nodes .e.g., Loop, Choice and Parallel). We collect a set of semantic equivalent critical nodes, compare their structural similarities, and establish the synchronization between the as-specified workflow and the as-implemented workflow in terms of segments.

To this extent, we can refine the extracted business logics within the scope of each synchronized segments. We can decompose coarse-grained business logics, or encapsulate a set of fine-grained business logics. Moreover, the precision of the extraction can be enhanced by the assistance of the data information collected from both types of workflows. In particular, we utilize the input and output data information in the as-specified workflows to determine the boundaries between business logics in source code

2.2 As-Implemented Workflow Extraction

Generating a complete trace record would produce a workflow graph that contains a large number of entities that are outside our interests. As a result, we adapted static tracing to identify business logics from source code using a set of heuristics to prune the number of code entities in the trace records. As discussed in [1], the heuristics include: 1) Utility code, 2) Java objects 3) Exception objects that should be filtered and 4) Data objects, 5) Task objects 6) User-defined class objects that are part of the business logics.

While code-heuristic recovery can be effective, various programming styles and the use of the encapsulation principle in software design has limited its usefulness to particular domains, and created challenges to extract business logics in an appropriate level of granularity. Therefore, we also apply the software metrics and documentary structure of the source code.

As discussed in [8] and [9], the documentary structures, such as the comments and white spaces, are arranged by programmers to differentiate the distance between different concepts. In practice, developers focus on a single task at a time and group the related statements into a code segment. They insert comments and white spaces between each function, (i.e. business logic). As the result, comments and white spaces are at the positions that separate different functions to help the developers understand and locate the code segments of their interests; especially the comments that occupied one or more lines in the source code, i.e. the comments that are not inside or after a statement. Therefore, documentary structures can be considered as candidates to separate the business logics in sequences. To refine a segment that represents on business logic, we further measure the properties of the separated segments using the software metrics, such as line of code in the segment, number of input and output, cohesion and coupling metrics, and name similarity. Since business logic is a function that computes the output based on the input and conditions, the statements that belong to the same business logic should be highly dependent on each other. These metrics check the interdependency between the adjacent segments. According to the outcomes of the metric measurements, the segments can be merged into a single logic if the segments are highly interdependent; else they will be left as it-is and treated as separate logics.

2.3 Intermediate Behavior Model for As-Specified and As-Implemented Workflow

An as-specified workflow has a higher level of abstraction than an as-implemented workflow. it is necessary to bring the both types of workflows to a common ground in order to perform the comparison between the two. In our research, we have developed an abstract behavioral model to represent control structures and their execution behaviors of as-specified and as-implemented workflows in a unified form.

As shown in Figure 2, there are three types of entities, namely *Control, Task and Data* to capture the control flow, the business logic or process, and input and output in the workflows, respectively. Each type contains the concrete entities for the low level behaviors in the workflows; for instance, *ControlEntity* contains *Choice* that models the alternatives in as-specified workflow and if-else statements in source code, and *DataEntity* can be either *Input* or *Output* of a task or a (sub)proess, as depicted in Figure 2. We aim to identify the concrete entities that represent the direct realization (e.g.,. a *Logic* is a task in the as-specified workflows) or indirect realization (e.g.,. a *Parallel* is implemented by threads in Java) in either as-implemented or as-specified workflows. This abstract behavioral model allows us to effectively compare both types of workflows.

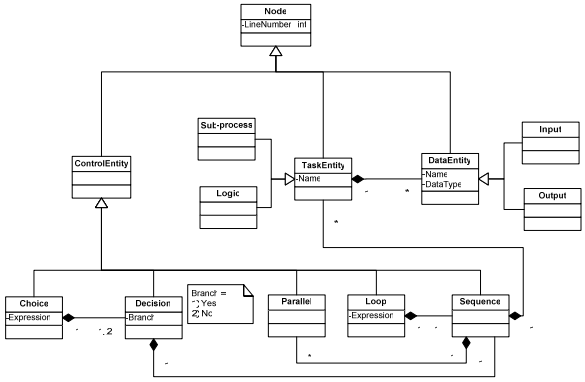

Figure 2 – Intermediate Abstraction Behavior Model

2.4 Comparison Algorithm

In our research, we developed an algorithm to compare the as-specified workflows with as-implemented workflows. Specially, the comparison algorithm attempts to match control nodes in the as-implemented workflows to the control nodes in the as-specified workflows into pairs of the matched critical nodes. The algorithm walks through both types of workflows (presented in intermediate model) until it matches the candidate critical nodes with the highest similarity. We measure the similarity of the critical nodes by the following properties:

1) Type: Are they the same type, ex. Choice nodes? The matched nodes must be the same type.

2) Order: Are they following or followed by the same types of control nodes? The nodes are more similar if their neighbor nodes are also related.

3) Depth: Does a critical node in the as-implemented workflow is equal or deeper in the nested scopes than the ones in the as-specified workflow? Since the abstraction level of the as-implemented workflows are lower than the level in as-specified workflows, the as-implemented workflows should contain more details and thus more nested scopes.

4) Inner contents: Do they have the same types of control or task nodes as children nodes? Two nodes are more similar if they both contain the same types of the control or task nodes

The pairs of the matched critical nodes serve as the boundaries of the business logics. Inside the matched

critical nodes, we map task nodes in as-implemented workflows to the task nodes in as-specified workflows, based on names, order and input and output of the tasks. Some business logics in the as-implemented workflows can be merged to increase the abstraction level. In some cases, business logics, which were not recognized in the requirement or newly added to the source code, can be identified from the comparison algorithm because no corresponding ones will be found in as-specified workflows.

3. Conclusions and Future Work

Currently, most research has focused on single source, i.e. source code, for extracting workflows from source code. In this paper we propose an approach to perform the workflow extraction by incorporating as-specified workflows modeled by business analysts. By extracting control constructs from the two artifacts and modeling them in a unified manner, we are able to measure the similarity between the as-specified workflows and as-implemented workflows. Based on this measurement, we are able to validate the conformance of the extracted as-implemented workflows, refine the abstraction without human intervention, and generate the updated as-specified workflow documents.

In the future, we plan to generalize the framework and to test it on systems in different domains. We are also interested in developers' mental models during the development and other software metrics that computes the similarity between code segments. These models and metrics will help us separate and merge the business logics from the source code with better accuracy.

References

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

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

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

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

[5] 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

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

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

[8] M. Folwer, "Refactoring: Improving the Design of Existing Progrmas", Addison-Wesley, 1999

[9] M. Van De Vanter, "The Documentary Structure of Source Code", to appear in Information & Software Technology, 2002

[10] D. A. Manolescu and R. E. Johnson, "A Micro Workflow Framework for Compositional Object-Oriented Software Development", in proceedings of the OOPSLA'99 Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems, 1999