Proceedings of the 1st International Workshop on

REVERSE ENGINEERING TO REQUIREMENTS (RETR'05)

Collocated with the Joint conference of WICSA & WCRE'05
Pittsburgh, PA USA
http://www.cs.toronto.edu/km/retr

Organizers

Yijun Yu
Yiqiao Wang
Sotirios Liaskos
Alexei Lapouchnian
John Mylopoulos
University of Toronto
Ying Zou
Queen's University
Marin Litoiu
IBM Canada, Ltd.
Julio C.S.P. Leite
PUC-Rio, Brazil

Program Committee

Periklis Andritsos
University of Trento, Italy
Nicolas Anquetil
Universidade Catolica de Brasilia, Brazil
Daniel M. Berry
University of Waterloo, Canada
Marsha Chechik
University of Toronto, Canada
Elliot Chikofsky
DMR TRECOM, USA
Luiz Marcio Cysneiros
York University, Canada
Steve Easterbrook
University of Toronto, Canada
Stan Jarzabek
University of Singapore, Singapore
Xiaoping Jia
DePaul University, USA
Kostas Kontogiannis
University of Waterloo, Canada
Ric Holt
University of Waterloo, Canada
Chang Liu
Ohio University, USA
Lin Liu
Tsinghua University, China
Jianguo Lu
University of Windsor, Canada
Paulo Cesar Masiero
Universidade de São Paulo, Brazil
Hausi Muller
University of Victoria, Canada
John Mylopoulos
University of Toronto, Canada
Julio Cesar Sampaio do Prado Leite
PUC-Rio, Brazil
Marin Litiou
IBM, Canada
Antonio Francisco do Prado
Universidade Federal de São Carlos, Brazil
Ladan Tahvildari
University of Waterloo, Canada
Kenny Wong
University of Alberta, Canada
Tao Xie
North Carolina State University, USA
Eric S. K. Yu
University of Toronto, Canada
Yijun Yu
University of Toronto, Canada
Ying Zou
Queens University, Canada

A full day workshop collocated with WCRE, Pittsburgh, Pennsylvania, USA (Carnegie Mellon). Date to be decided, will be one day in the week of 6 November 2005.
[Call for Papers | Topics | Submission | Important Dates | WCRE 2005 Home]

Reverse engineering aims at *extracting many kinds of information from existing software, such as requirements specifications, design documents*, *and system artifacts*, and *using this information in system renovation and program understanding* [WCRE].

Existing *reverse engineering* methods focus on recovering architecture and design of software products that are often represented in standard formats such as UML, GXL or ADL. However, few methods *recover requirements* such as goals of the various stakeholders, non-functional requirements, early aspects, variability tradeoffs and dynamic/emergent behavior of autonomic systems.

Therefore a forum is needed to discuss the issues related to recovering requirements. It can enable the reverse engineered software systems to continuously adapt to the evolving functional requirements, and to be reengineered to meet the non-functional requirements.

The goal of this full day WCRE'05 workshop is to bring together researchers and practitioners interested in developing methods and techniques for Reverse Engineering to Requirements (RETR). The objective of the workshop is to sketch the state-of-the-art of the RETR practice and to identify current trends and fields of interest, possible paths of collaboration and points of future research directions.

Topics

The workshop focuses on issues including, but not limited to, the following:

- Early Requirements
  - Aspects
  - Goals
  - Scenarios
  - Use cases
  - Variabilies
  - Viewpoints
- Non-functional Requirements and Qualities
  - Understandability
  - Maintainability
  - Performance
  - Reliability
  - Security
  - Privacy
  - Usability
  - Interoperability
- Evolution of Software Requirements
  - Mining and Clustering Software Repositories
  - Reconciliation of Requirements and Implementation
  - Adaptive Software for Autonomic Computing
  - Emergent Behavior in Software Integrations
- Traceability
  - Establishing Traceability
  - Maintaining Traceability
- Verification and Validation
  - Design Rationale and Impact Analysis
  - Requirements Testing
  - Empirical Case Studies

Submission Details

Participants of RETR 2005 are asked to submit papers relating to the scope of the workshop. Papers must be original and previously unpublished. The organizing committee will oversee the reviewing process. Selection will be based on originality, ability to stimulate discussion, and presentation quality. Authors of accepted papers are expected to participate in the workshop. Papers must be in WCRE 2005 submission format (i.e. IEEE Proceedings style in accepted Postscript or PDF form), in AT MOST 6 pages. You can submit your paper to retr(at)cs.toronto.edu. All accepted position papers will be available electronically without page limitation before the workshop.

Important Dates

Paper due: October 3, 2005
Notification to Authors: October 17, 2005
Camera-ready Papers due: October 25, 2005
Workshop date: the week of November 6, 2005

What's the mission of RETR'05

Three engineering research communities such as the International Conference on Software Engineering (ICSE), the Working Conference on Reverse Engineering (WCRE) and the International Conference/Symposium on Requirements Engineering (RE) are known to be highly involved with the software industry. They share largely common interests among researchers and practitioners, as indicated by an intersection of authorship found from the (reverse engineered) statistics based on the literature archive DBLP[1], see Table 1. It shows that exactly 20% of the unique authors in WCRE and RE have also published in the ICSE proceedings.

Table 1. Authorship distribution among engineering research communities

Venue	years	no. papers	no. authors	ICSE presence (%)	The 1st and 2nd most prolific authors
ICSE	27	1966	2921	2921(100%)	Victor R. Basili (22), Barry W.Boehm (21)
RE	13	378	662	133(20%)	Bashar Nuseibeh (12), Michael Jackson (8)
WCRE	12	344	491	102(20%)	Richard C. Holt (15), Kostas Kontogiannis(13)

Among all the 1142 unique authors who have published in either RE or WCRE, however, only 11 have published papers in both. As detailed in Table 2, interestingly, these 11 authors have published more papers in the RE proceedings than ever in the WCRE proceedings. Fortunately we have invited 3 of them to our workshop program committee. We will see more people joining the list after this year, e.g. Mike Godfrey has a paper in RE'05, which will break the ad hoc pattern that a selected author has more publications in RE than in WCRE.

Table 2. Distribution of publications among the authors who published in both WCRE and RE. RE'05 Publications are added to some of the authors. Three are in the programme committee of RETR'05.

no.	name	RE	WCRE	ICSE	RETR-PC
1	John Mylopoulos	2	7+2	5	Y
2	Julio Cesar Sampaio do Prado Leite	2	7+1	2	Y
3	Betty H. C. Cheng	2	5+1	4	N
4	Colin Potts	1	8	7	N
5	Margaret-Anne D. Storey	1	5	4	N
6	Harald Gall	1	5	3	N
7	Wojtek Kozaczynski	1	1	6	N
8	Mehdi Jazayeri	1	1	3	N
9	Stan Jarzabek	1	1	2	Y
10	Roland Mittermeir	1	1	1	N
11	Eleftherios Koutsofios	1	1	0	N
	Mike Godfrey	4	0+1	0	N

The above data are by no means complete, as some venues attract papers of both RE and WCRE areas. However, it does clearly indicate that few people (2.2% of WCRE and 1.6% of RE authors) did recognized work in both.

Thus, the mission for RETR is to bridge the gap between the two seemingly connected areas. We shall (1) aim WCRE **higher**: *e.g. reverse engineering to requirements, not just to architecture and design*; (2) ground RE **deeper**: *e.g. find traceability between requirements and implementations*; and (3) reveal **newer** potentials: *e.g. combine both requirements and reverse engineering in autonomic systems.*

[1]Retrieved from http://dblp.uni-trier.de/xml on November 2, 2005. Note that recent RE'05 and WCRE'05 papers are not listed in the DBLP records.

RETR: Reverse Engineering To Requirements

Yijun Yu, John Mylopoulos, Yiqiao Wang, Sotirios Liaskos, Alexei Lapouchnian, University of Toronto {yijun,jm,yw,liaskos,alexei}@cs.toronto.edu

Ying Zou, Queen's University, ying.zou@queensu.ca

Marin Litiou, IBM Canada, marin@ca.ibm.com

Julio C.S.P. Leite, PUC-Rio, julio@inf.puc-rio.br

Abstract

Reverse engineering aims at extracting many kinds of information from existing software and using this information for system renovation and program understanding. The goal of this full day WCRE'05 workshop is to identify methods and techniques for Reverse Engineering from software to Requirements (RETR).

1 Introduction

Reverse engineering aims at *extracting many kinds of information from existing software, such as requirements specifications, design documents*, and *system artifacts, and using this information in system renovation and program understanding* [1].

Existing *reverse engineering* methods focus on recovering architecture and design of software products that are often represented in standard formats such as UML, GXL or ADL. However, few methods *recover requirements* such as goals of the various stakeholders, non-functional requirements, early aspects, variability tradeoffs and dynamic/emmergent behavior of autonomic systems.

Therefore a forum is needed to discuss the issues related to recovering requirements from the software. It can enable the reverse engineered software systems to continuously adapt to the evolving functional requirements, and to be reengineered to meet the non-functional requirements.

2 Topics of interest

Topics of interest include, but are not limited to:

```
+ Early Requirements
  - Aspects
  - Goals
  - Scenarios
  - Use cases
  - Variabilities
  - Viewpoints
+ Non-functional Requirements and Qualities
  - Understandability and Maintainability
  - Performance, Usability and Reliability
  - Security and Privacy
  - Interoperability
+ Evolution of Software Requirements
  - Mining and Clustering Software Repositories
  - Reconciliation of Requirements and Implementation
  - Adaptive Software for Autonomic Computing
  - Emergent Behavior in Software Integrations
+ Traceability
  - Establishing Traceability
  - Maintaining Traceability
+ Verification and Validation
  - Design Rationale and Impact Analysis
  - Requirements Testing
  - Empirical Case Studies
```

3 Objectives of the workshop

The goal of this full day WCRE'05 workshop is to work on methods and techniques for Reverse Engineering from software to Requirements (RETR). The objective of the workshop is to sketch the state-of-the-art of the RETR practice and to identify current trends and fields of interest, possible paths of collaboration and points of future research directions.

References

[1] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17. 1990.

Workshop Schedule

Time	Event
8:30	Opening and Introduce ourselves
9:00	Invited talk: The Challenge to Recover 15 Years of "Why" Behind a Product Design by Kevin Dunipace
9:30	Invited talk: Learning from Past Trial and Error: Some History of Reverse Engineering to Requirements by Elliot Chikofsky
10:00	Presentation (pp. 5-11): Automatic Extraction of Abstract-Object-State Machines Based on Branch Coverage by Hai Yuan & Tao Xie
10:45	Presentation (pp. 24-28): Extracting Business Policy and Business Data from the Three-Tier Architecture System by Maokeng Hung & Ying Zou
11:30	Panel discussions on RETR from legacy software design
12:00	Lunch break
13:00	Break, you may attend Grady Boochs keynote speech at WICSA
14:30	Invited talk Autonomic Computing: Now You See It, Now You Don't by Hausi Muller
15:00	Invited talk by Kostas Kontogiannis and Ladan Tahvildari
15:30	Presentation (pp. 18-23): Requirements-Driven Configuration of Software Systems by Yijun Yu, Alexei Lapouchnian, Sotirios Liaskos & John Mylopoulos
16:00	Break
16:30	Presentation (pp. 12-17): Towards a Framework to Incorporate NFRs into UML Models by Sabrina Anjum Tonu & Ladan Tahvildari
17:00	Panel discussion on RETR for autonomic systems
17:30	Wrap up & conclusions on lessons learnt, next steps
18:00	End of the workshop

Enjoy!

Automatic Extraction of Abstract-Object-State Machines Based on Branch Coverage

Hai Yuan
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
hyuan3@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

Abstract

Some requirement modelling languages such as UML's statechart diagrams allow developers to specify requirements of state-transition behavior in a visual way. These requirement specifications are useful in many ways, including helping program understanding and specification-based testing. However, there are a large number of legacy systems that are not equipped with these requirement specifications. This paper proposes a new approach, called Brastra, for extracting object state machines (OSM) from unit-test executions. An OSM describes how a method call transits an object from one state to another. When the state of an object is represented with concrete-state information (the values of fields transitively reachable from the object), the extracted OSMs are simply too complex to be useful. Our Brastra approach abstracts an object's concrete state to an abstract state based on the branch coverage information exercised by methods invoked on the object. We have prototyped our Brastra approach and shown the utility of the approach with an illustrating example. Our initial experience shows that Brastra can extract compact OSMs that provide useful information for understanding state-transition behavior.

1 Introduction

The Unified Modelling Languages (UML) [15] provides a set of notations for describing requirements of artifacts in software systems. Among these notations, statechart diagrams capture state-transition behavior of a class or multiple classes. After requirements specifications are specified, developers can write source code to implement the specified behavior. Later when developers want to understand and maintain the source code, they can refer to requirements specifications besides directly inspecting the source code. In addition, developers can use specification-based testing tools [6, 8, 12, 17] to generate test inputs from the specifications and check the behavior of implementation with the behavior specified in requirements specifications. However, a number of legacy systems are not equipped with specifications. Understanding and testing these legacy systems present a challenge for developers. To address this challenge, researchers have developed various reverse engineering techniques [11] to infer various types of information from legacy systems.

This paper proposes Brastra, a new approach for automatically extracting object state machines (OSM) [21] for a class from unit-test executions. These OSMs describe state-transition behavior exhibited by invoking methods on objects of a class. An OSM is similar to a UML statechart diagram. In an OSM for a class, a state represents the state of an object at runtime. A transition represents method calls invoked on an object, transiting the object from one state to another. States in an OSM can be concrete or abstract. A concrete state of an object is characterized by the values of object fields transitively reachable from the object. Because a concrete OSM is often too complicated to be useful, our previous work [21, 22] has developed techniques to abstract concrete states to abstract states, which are used to construct abstract OSMs. Our previous observer-abstraction approach [21] represents an abstract state of an object with the return values of observer methods (methods whose returns are not void) invoked on the object. Our previous sliced-OSM-extraction approach [22] represents an abstract state of an object with the values of a specific field. In this paper, we have developed the new Brastra approach that does not require appropriate observer methods in class interface (required by our previous observer abstraction approach [21]) or appropriate object-field structure (required by our previous sliced OSM extraction approach [22]).

The Brastra approach represents an abstract state of an object with the branch coverage information produced by methods invoked on the object. OSMs produced by Brastra

capture program behavior exhibited by branching points in method body, complementing program behavior exhibited by observer methods or specific fields (captured by our previous approaches). We have implemented the Brastra approach and demonstrated its utility by applying it on an illustrating example. Our initial experience shows that OSMs extracted by Brastra are compact and useful for providing insights to state-transition behavior.

The rest of this paper is organized as follows. Section 2 presents an illustrating example. Section 3 introduces the formal definition of an OSM. Section 4 illustrates our new approach for extracting OSMs based on branch coverage information. Section 5 introduces our implementation of the approach. Section 6 discusses issues of the approach and lays out future directions. Section 7 reviews related work, and Section 8 concludes.

2 Example

As an illustrating example, we use a data structure: a `UBStack` class, which is the implementation of a bounded stack that stores unique elements of integer type. Figure 1 shows the class including two standard stack operations: `push` and `pop`. Stotts et al. coded this Java implementation to experiment with their algebraic-specification-based approach for systematically creating unit tests [16]. In the class implementation, the `max` is the capacity of the stack, the array `elems` contains the elements stored in the stack, and `numberOfElements` is the number of the elements and the index of the first free location in the stack.

The `push` method first checks whether the element to be pushed exists already in the stack. If the same element already exists in the stack, the method moves the element to the top of the stack. Otherwise, the method increases `numberOfElements` after writing the element into the `elems` array if `numberOfElements` does not exceed the stack capacity `max`. If the stack capacity is exceeded, the method prints an error message and makes no changes on the stack. The `pop` method first checks whether `numberOfElements` is greater than zero. If so, it retrieves the top element of the stack, decreases `numberOfElements`, and returns the retrieved element; otherwise, the method prints an error message and returns `-1` as an error indicator.

To generate tests for `UBStack`, we first manually configure `push`'s arguments to be 1, 2, 3, or 4.[1]. Given the bytecode of `UBStack` our previously developed Rostra tool [19] automatically generates 263 tests; these generated tests exercise 41 non-equivalent concrete object states (two concrete object states are non-equivalent if their concrete state representations are different).

[1]We can use some existing test generation tools such as Parasoft Jtest [13] or JCrasher [2] to automatically generate method arguments for `UBStack`, but these tools may not generate relevant method arguments.

```
public class UBStack {
  private int max;
  private int[] elems;
  private int numberOfElements;

  public UBStack() {
    numberOfElements = 0;
    max = 3;
    elems = new int[max];
  }
  public void push(int k) {
    int index;
    boolean alreadyMember = false;
    for(index=0; index<numberOfElements; index++) {
       if (k==elems[index]) {
          alreadyMember = true;
          break;
       }
    }
    if (alreadyMember) {
       for (int j=index; j<numberOfElements-1; j++)
           elems[j] = elems[j+1];
       elems[numberOfElements-1] = k;
    } else {
       if (numberOfElements < max) {
          elems[numberOfElements] = k;
          numberOfElements++;
          return;
       } else {
          System.out.println("Stack full, cannot push");
          return;
       }
    }
  }
  public int pop(){
    int ret = -1;
    if (numberOfElements > 0) {
         ret = elems[numberOfElements-1];
         elems[numberOfElements-1] = 0;
         numberOfElements --;
    } else {
        System.out.println("Stack empty, cannot pop");
    }
    return ret;
  }
}
```

Figure 1. A bounded-stack implementation that accommodates unique integer elements

3 Object State Machine

In our previous work [21], We have defined an object state machine for a class:

Definition 1 *An* object state machine *(OSM) M of a component c is a sextuple M = (I, O, S, δ, λ, $INIT$) where I, O, and S are nonempty sets of method calls in c's interface, returns of these method calls, and states of c's objects, respectively. $INIT \in S$ is the initial state that the machine is in before calling any constructor method of c. $\delta : S \times I \rightarrow P(S)$ is the state transition function and $\lambda : S \times I \rightarrow P(O)$ is the output function where $P(S)$ and $P(O)$ are the power sets of S and O, respectively. When the machine is in a current state s and receives a method call i from I, it moves to one of the next states specified by $\delta(s, i)$ and produces one of the method returns given by $\lambda(s, i)$.*

The object states in an OSM can be concrete or abstract. In a concrete OSM, states of an object are represented by its concrete-state representation. An object's concrete-state representation is characterized by the values of all the field transitively reachable from the object [19]. Because some object fields may be reference types and their values point to memory addresses (which can be different in different runs of the same test), we use a linearization algorithm [19] to collect the values of these reference-type fields so that comparing state representations takes into account comparing object-graph shapes but without directly comparing memory addresses. Two states are *equivalent* if their state representations are the same, and are *nonequivalent* otherwise.

For example, the generated tests for `UBStack` exercise 41 nonequivalent concrete object states. There are 142 transitions among these states. Figure 2 shows a concrete OSM exercised by generated tests and Figure 3 shows a detailed view of the highlighted area in Figure 2. The OSM is displayed by using the Grappa package, which is part of graphviz [5]. States in the OSM are shown as circles in Figure 3 and the labels inside these circles are the state representations, which include field names followed by ":" and corresponding field values (array-element values are separated by ";"). The three states in Figure 3 represent three full stacks. Although they have the same set of stack elements, these elements are stored in three stacks in different orders. Transitions in the OSM are shown as directed edges that connect circles (states). These edges are labelled with method names and arguments (for brevity, we do not show method return values in the edge labels).

We have observed that the concrete OSM is too complex to be useful in practice. Although we can zoom in to view details of object states and transitions among them, these details in such a large OSM are often not very useful for program understanding or test-result inspection.

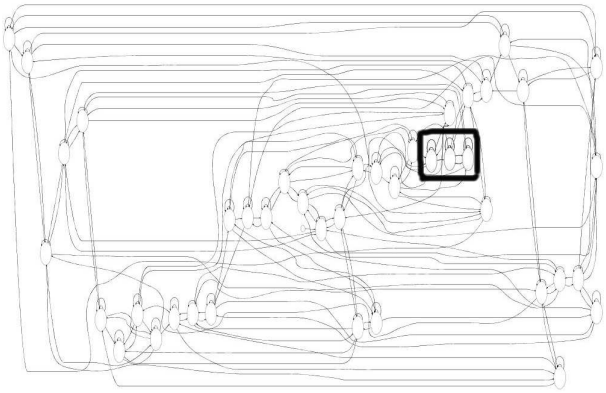

Figure 2. An overview of `UBStack` concrete OSM (containing 41 states and 142 transitions) exercised by generated tests

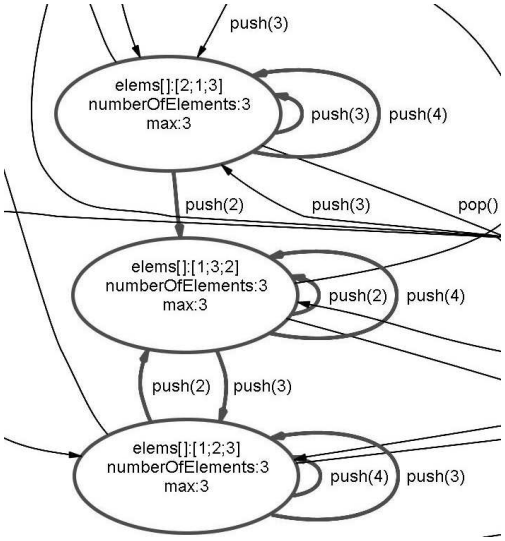

Figure 3. A detailed view of the selected area in `UBStack` concrete OSM

4 Approach

To reduce the complexity of an OSM, we can construct an *abstraction function* [10] to map concrete states to abstract states. Our Brastra approach constructs such an abstraction function by using branch coverage information. We first define the branch coverage we shall use in representing an abstract state of an object.

A method m is characterized by its defining class c, method name and method signature. Then we define conditional set for a method m.

Definition 2 Conditional set CS *of a method* m *are a set of strings, including all the conditional strings (together with their source-code-line numbers) that appear in the body of* m*,* m*'s direct and indirect callees.*

A method call mc is a pair $\langle m, a\rangle$ where m is a method and a is a vector of method-argument values.

Definition 3 *Given an object* o *of class* c *and a method call* mc*:*$\langle m, a\rangle$ *of* c*, assume* CS *is the conditional set of* m*,* branch coverage BC *of* mc *on* o *is a map from* CS *to* $\{true, false, both, n/a\}$*, where the map is defined based on whether a conditional's false branch, true branch, both branches, or neither branch is covered during the execution of* mc *on* o*.*

Definition 4 *Given an object* o *of class* c *and a set of* c*'s method calls* MC = $\{mc_1, mc_2, ..., mc_n\}$*, the abstract state of* o *with respect to* MC *is represented by* $\{BC_1, BC_2, ..., BC_n\}$*, where* BC_i *is branch coverage of* mc_i *on* o*.*

Then we construct an abstract OSM where all states are abstract states with respect to MC.

For example, assume MC for `UBStack` is {`pop()`, `push(1)`, `push(2)`, `push(3)`, `push(4)`} and consider the following tests:

```
Example Test:
   UBStack s = new UBStack();
   s.push(1);
   s.pop();
   s.push(2);
   s.push(3);
   s.push(4);
```

After the end of the constructor call, if we invoke `pop()` on s, the `pop` method execution covers the false branch of the following conditional: `(numberOfElements > 0)`. We represent the map of `(numberOfElements>0)` $\rightarrow$ `false` as `!(numberOfElements>0)`.

To simplify illustration, we do not display source-code line numbers for conditional strings. When a conditional `c` is mapped to `both`, which indicates both branches of the conditional are covered, we simply represent the mapping with two entries `c` and `!c`.

After the end of the constructor call, if we invoke any of `push(1), push(2), push(3), and push(4)` on s, the `push` method execution covers the following branches following the preceding notations:

```
!(index<numberOfElements)
!(alreadyMember)
numberOfElements < max
```

Figure 4 shows the abstract OSM extracted by Brastra based on branch coverage information. The top state is labelled as `INIT`, which indicates no state before invoking a constructor. Then we represent the abstract state after the constructor call as the second to top state of the abstract OSM shown in Figure 4. On the top part of the state, we display the object field values that are common to all the concrete states represented by the the abstract state. Then we display the branch coverage for `pop` (we put method name `pop` before the first line of branch coverage). Finally we display the branch coverage for `push`. To simplify the view, we do not display the method arguments or returns on the transitions in the OSM.

Interesting behavior occurs when we abstract the concrete states resulting from invoking `push(1)` or `push(2)` on an empty stack (note that in the example test, push(2) is actually invoked on an empty stack because its preceding method call `pop()` counteracts the effect of `push(1)`, transiting the state to an empty stack). On a concrete state resulting from `push(1)`, invoking `push(1)` again follows a path different from invoking `push(2)`, because the stack stores only unique elements. Therefore, we can observe in the middle state of Figure 4 there are two different branch coverage for `push`: one representing the case where the `push`'s argument has already existed in the stack and the other representing the case where the `push`'s argument does

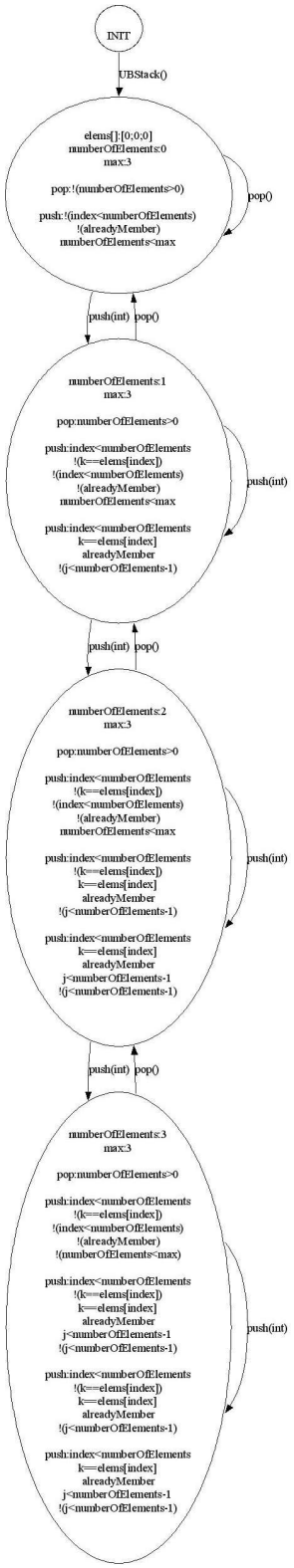

Figure 4. An overview of `UBStack` abstract OSM based on branch coverage (containing 4 states and 11 transitions) exercised by generated tests

not exist in the stack. The branches of `alreadyMember` and `!alreadyMember` from two sets of `push` branch coverage give us hints on these two cases.

The second to the bottom state has three sets of different `push` branch coverage, in addition to one set of `pop` coverage. The first set represents the case where the `push`'s argument does not exist in the stack, the second set represents the case where the `push`'s argument exists in the stack and the existing element is on the top of the stack (therefore, the element is not required to be moved to the top), and the third set represents the case where the `push`'s argument exists in the stack and the existing element is not on the top of the stack (therefore, the element is required to be moved to the top). In the example test, the concrete state of `s` after invoking `push(3)` falls into this abstract state.

The bottom state indicates a full stack; therefore, no `push` method call can further change the object state. Note that because a full stack with different concrete states can contain different elements; therefore, unlike in the second to top state, we do not display the values of the `elems[]` field. In the example test, the concrete state of `s` after invoking `push(4)` falls into this abstract state.

5 Implementation

Given a class, our Rostra tool [19] generates test inputs to exhaustively exercise object states iteratively. In particular, if users provide some sample method arguments, Rostra can use them; otherwise, Rostra uses Parasoft Jtest [13] or JCrasher [2] to generate method arguments. Then Rostra uses these method arguments to explore the object state space iteratively. Tool users can configure the maximum iteration number for Rostra to explore the state space. For `UBStack`, which has capacity of three, four iterations are sufficient to explore all possible states with the method arguments of `pop`, `push(1)`, `push(2)`, `push(3)`, and `push(4)`. Note that the Rostra's bounded-exhaustive test generation enables a better inspection of OSMs extracted from generated-test executions. For example, when tool users find out that an expected transition is missing in OSMs, it can have two reasons: a test that is required to produce that transition is missing or there is a bug in the program. Rostra's bounded-exhaustive test generation reduces the chance of the former case. In addition, Rostra's bounded-exhaustive test generation also facilitates our abstraction based on branch coverage. In order to abstract a concrete state, we need specific method calls to be invoked on the concrete state; these method calls are generated by Rostra. Note that when we invoke a method call on a concrete state in order to abstract the concrete state, the method call could modify the concrete state and later method calls on this concrete state need a reproduction of the concrete state; reproductions of concrete states are also supported by Rostra.

After Rostra generates test inputs and exports them into JUnit [4] test classes, we run these test classes with our previously developed Jusc [23] tool, a Java unit-test selection tool based on residual structural coverage [14], to output a path trace file after program executions terminate. We developed a tool to postprocess the collected path trace file to collect branch coverage information. Note that we collect branch hit coverage; therefore, when there are loop iterations during program executions, we do not count how many times a branch is hit nor collect execution orders among branches. This design decision provides further abstraction of states.

In addition, we also use the Daikon [3] Java frontend to run these test classes and collect object states exercised by these tests. Daikon [3] is a tool that dynamically detects likely program invariants in the program executions. It can collect object-field values during program executions, and reports properties that hold true on these fields during the executions. In our approach, we use Daikon to collect object states during program executions and later use these states to extract common field values among concrete states represented by an abstract state and then display the common field values in the state as an annotation.

6 Discussion and Future Work

Two main factors may affect our approach's usability in practice: methods' control flow graphs and generated test inputs. Branching points in control flow graphs take the role of abstraction functions [10]. Although different implementations of the same program behavior can have different control flow graphs, their implied behavior can be similar across different implementations. As is discussed in Section 7, we found that branch coverage information seems to more faithfully reflect interesting program behavior than our previous observer-abstraction approach [21] or sliced-OSM-extraction approach [22].

Besides the characteristics of control flow graphs, the executed test inputs can also affect the quality or complexity of an extracted OSM. Rostra's test generation has two controllable configurations: method arguments and the maximum iteration number. But comparing to previous approaches based on object-field values [22] or return values of observers [21], our new approach is less affected by the actual argument values in the generated tests inputs. But at the same time, choosing right argument values are also important. For example, if we choose only two different method arguments for `push` of `UBStack`, we can never reach a full stack state for `UBStack`. The maximum iteration number can have an effect if some boundary states are not exercised by a low maximum iteration number. For example, if we specify the maximum iteration number as

three, we cannot exercise a full stack state for `UBStack`.

There are several future directions for us to extend the Brastra approach. First, we plan to adapt the existing finite-state-machine-based testing techniques [9] or testing techniques based on UML statechart diagrams [6,8,12,17]. Extracted OSMs can guide further test generation to improve OSM extraction. These iterations form a feedback loop between test generation and specification inference proposed in our previous work [20].

Second, we plan to extend our specification inference for multiple classes instead of a single class. This may require adaptations of our diagram representations as well as inference algorithms.

Finally, we plan to investigate how human inputs can be used to improve the effectiveness of Brastra, which is currently developed as a totally automated tool. For example, when a Brastra-generated OSM is still too complicated to be understandable, developers can configure the state abstraction to be based on only the branches in a specified subset of public methods or the branches that are related to specified object fields. In addition, our Brastra approach is currently a dynamic analysis approach that focuses on functional behavior exercised by a class. There exists research on recovering non-functional requirements from legacy code such as the static analysis approach developed by Yu et al. [24]. In order to identify non-functional requirements, their approach requires some human manipulations of legacy code such as program refactoring. We plan to investigate how human inputs as well as static analysis can guide Brastra to extract non-functional behavior.

7 Related Work

The observer-abstraction approach was developed in our previous work [21]. The observer abstraction approach represents a state of an object by using the return values of observers invoked on the object. When we applied the observer abstraction on `UBStack`, we could invoke `pop`, the only observer, on an object and uses `pop`'s return value to abstract the state of the object. By considering `pop`'s semantic, we basically used the element on the top of the stack to abstract the whole stack. This abstraction is not helpful for us to understand `UBStack`'s behavior. The sliced-OSM-extraction approach was developed in our previous work [22]. It uses the values of an object's single field to represent the state of the object. For example, we can use the values of the `numberOfElements` field to represent states and the resulting OSM is similar to the OSM extracted by Brastra. But when we set the capacity of `UBStack` to be a large number such as 10, the size of the OSM extracted by using `numberOfElements` would grow linearly with iteration numbers, whereas the OSM extracted by Brastra keeps the original shape because loop iterations have been abstracted away by our mechanism of considering only branch hit coverage without considering how many times loop iterations are executed.

Kung et al. [7] statically extract object state models from a class's source code and use them to guide test generation. States in a object state model are defined by value intervals over object fields, which are derived from path conditions of method source; the transitions are derived by symbolically executing methods. Both their approach and our approach consider branches in method body, but their approach can exploit a limited types of conditionals (e.g., conditionals that compare an object field with a constant) and their approach statically extract state models with a limited capability.

From system-test executions, both Whaley et al. [18] and Ammons et al. [1] mine protocol specifications for component interfaces. They use sequencing order among method calls in the interfaces without using internal object-field values or structural coverage information. Both approaches usually require a good set of system tests for exercising component interfaces, whereas our approach generates test inputs to exercise component's object states in a small scope. Applying their approaches on our generated unit tests for `UBStack` would yield a circle connecting `push` and `pop`.

8 Conclusion

We have proposed Brastra, a new approach for automatically extracting object state machines (OSM) from unit-test executions. Because a concrete OSM extracted based on concrete states is often too complicated to be useful, Brastra abstracts the concrete state of an object by using the branch coverage exercised by methods invoked on the object. We have implemented the Brastra approach and demonstrated its utility on an illustrating example. Our initial experience has shown an OSM extracted by Brastra provides succinct information for understanding key program behavior of a class.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[2] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[4] E. Gamma and K. Beck. JUnit, 2003. `http://www.junit.org`.

[5] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, Sept. 2000.

[6] Y. Kim, H. Hong, D. Bae, and S. Cha. Test cases generation from UML state diagrams. *IEEE Proceedings- Software*, 146(4):197–192, 1999.

[7] D. Kung, N. Suchak, J. Gao, and P. Hsia. On object state testing. In *Proc. 18th International Computer Software and Applications Conference*, pages 222–227, 1994.

[8] Y. L. L.C. Briand, M. Di Penta. Assessing and improving state-based class testing: A series of experiments. *IEEE Transactions on Software Engineering*, 30(11), 2003.

[9] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proc. The IEEE*, volume 84, pages 1090–1123, Aug. 1996.

[10] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[11] H. A. Mueller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse Engineering: A Roadmap. In *Proc. 22nd International Conference on Software Engineering (ICSE 2000)*, 2000.

[12] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. 2nd International Conference on the Unified Modeling Language*, pages 416–429, October 1999.

[13] Parasoft Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.

[14] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proc. 21st International Conference on Software Engineering*, pages 277–284, 1999.

[15] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[16] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proc. 2002 XP/Agile Universe*, pages 131–143, 2002.

[17] M. Vieira, M. Dias, and D. Richardson. Object-oriented specification-based testing using UML state-chart diagrams. In *Proc. Workshop on Automated Program Analysis, Testing, and Verification at ICSE 2000*, June 2000.

[18] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis*, pages 218–228, 2002.

[19] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[20] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software*, volume 2931 of *LNCS*, pages 60–69, 2003.

[21] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods*, Nov. 2004.

[22] T. Xie and D. Notkin. Automatic extraction of sliced object state machines for component interfaces. In *Proc. 3rd Workshop on Specification and Verification of Component-Based Systems at ACM SIGSOFT 2004/FSE-12 (SAVCBS 2004)*, pages 39–46, October 2004.

[23] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.

[24] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite. Reverse engineering goal models from legacy code. In *Proc. International Conference on Requirements Engineering*, pages 363–372, October 2005.

Towards a Framework to Incorporate NFRs into UML Models

Subrina Anjum Tonu, Ladan Tahvildari
Software Technologies Applied Research Group
Dept. of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
{subrina, ltahvild}@swen.uwaterloo.ca

Abstract

Despite the fact that Non-Functional Requirements (NFRs) are very difficult to achieve and at the same time are expensive to deal with, a few research works have focused on them as first class requirements in a development process. We propose a framework to incorporate NFRs, as reusable components, with standard UML notations. Such a framework can also integrate those reusable NFRs with the extracted UML representations of legacy systems during the reverse engineering process. This novel research work uses standard XMI representation of UML models without proposing any extension to it. As a proof of concept, a small case study of a Credit Card System is presented.

1 Introduction

The demand for high quality software system is increasing day by day. Production of a highly organized software system requires separation of concerns [7] which is one of the basic engineering principles. On the other hand, production of a high quality software requires the implementation of all functional and non-functional requirements starting from the design phase to the end of the software life cycle.

As known, all these requirements are changing during the maintenance phase of any software system. The re-engineering of such software systems have gained significant attention in today's software industry. A few research works provide a re-engineering process that addresses such problems in order to incorporate any new or modified (functional and non-functional) requirements. Existing reverse engineering process can extract architectural design of the legacy systems which can be presented in UML model. UML tools also exist to automatically generate deployable source code from UML model specifications. In such an environment, the legacy systems can be modified by adding the NFRs with the extracted UML model out of such system and the source code can be re-generated automatically. In a nutshell, it is necessary to have an environment to attach the NFRs to the target system.

Tahvildari et al. proposed a quality-driven reengineering (QDR) framework which allows specific quality requirements for the migrant systems to be modelled as a collection of soft-goal graphs, and provides a selection of the transformational steps that needs to be applied at the source code level of the legacy system being reengineered [14]. They extended their work and proposed a framework of transformations that aims to improve error-prone design properties and assists in enhancing specific qualities of a software system using a catalogue of OO software metrics [13].

This research is an extension to that work by focusing on the extracted UML representation (from source code) of the legacy systems rather than AST. We also focus on making reusable NFRs and attaching them with the target model. In current practice, the join-point (where the NFR touches the target model) is defined as a part of the NFR itself. As a result, there is very little chance to reuse this NFR in other software design. This research work is a step to remove these shortcomings. First, we identify the functional requirements (FR) and non-functional requirements (NFR) of a legacy system which needs to be re-engineered. Second, we specify the new FRs with the appropriate UML diagrams and we specify a template of NFRs using NFR framework [4] and our proposed notations for creating standard UML diagrams. According to this approach, the NFRs do not have any hard coded join-points inside it. We used the concept of *dynamic parameterizations* described in [9]. We also use a knowledge-based concept for building a repository/library of those reusable NFRs. Finally, we integrate those NFRs with the target model where the necessary parameters for defining the join-points come dynamically during run time. We use UML as our design language as it is the most popular modelling language in research community, as well as a general purpose object-oriented language [1]. Our proposed framework is developed in a standard XMI environment.

Paper	Class Diagram	Use Case Diagram	Sequence Diagram	Interaction Diagram	State Diagram	Collaboration Diagram	Need Extension of UML
Lawrence Chung et.al [12]		√					√
Ana Moreira et.al [10]		√		√			√
Luiz Marcio Cysneiros et.al [5]	√	√	√			√	√
Evgeni Dimitrov et.al [6]		√		√	√		

Table 1. Summary of Related Works

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the framework while Section 4 applies the framework on a case study. Finally, Section 5 summarizes the contributions of this work and outlines directions for further research.

2 Related Works

The idea of integrating NFRs with FRs in design level is not a new one. As shown in Table 1 many researchers proposed extensions of UML model for the integration purpose.

Lawrence Chung et al. [12] proposed to integrate NFRs with FRs in UML use case model. They implemented the NFRs by their NFR framework [11, 4] and proposed to associate those NFRs with four use case model elements: actor, use case, actor-use case association and the system boundary. They named these associations as "Actor Association Point", "Use Case Association Point", "Actor-Use Case Association (AU-A) Association Point" and "System Boundary Association Point", respectively.

Ana Moreira et al. [10] proposed a model for integrating crosscutting quality attributes with FRs by UML use case diagram and interaction diagram. They proposed a template for quality attributes with some specific fields (such as description, focus, source, decomposition) and they integrated those quality attributes with FRs by using standard UML diagrammatic representations (e. g. use case diagram, interaction diagrams) extended with special notations.

Luiz Marcio Cysneiros et al. [5] proposed a systematic approach to assure that conceptual models will reflect the NFRs elicited. They uses a vocabulary anchor (LEL) to build both functional and nonfunctional perspectives of a software system. They also showed how to integrate NFRs into UML by extending some of the UML sublanguages, and they presented a systematic way to integrate NFRs into the functional models.

Evgeni Dimitrov et al. [6] described three approaches for UML-based performance engineering. The three approaches are: Approach-1) Direct representation of performance aspects using UML, Approach-2) Expanding UML to deal with performance aspects and Approach-3) Combining UML with formal description techniques. They proposed some extensions to UML use case and state transition diagram.

Our work is different from all these works in the sense that we do not apply our framework for a specific UML diagram, rather than we apply it in a general way for all types of UML model. Besides, we do not propose any extension to UML model, rather than we express the NFRs in a reusable way with the standard UML notations.

3 A Proposed Framework

Building of reusable NFR templates and the integration of them with the extracted UML models of the legacy systems requires a comprehensive framework to relate the integration process with the functional requirements of the target model. The focal point of the proposed research is to exploit the synergy between the area of software requirements analysis (both functional and non-functional) and software re-engineering.

We assume an extracted UML model is available from a legacy system after a reverse engineering process is applied. Our framework starts with such extracted UML model of a legacy system. It consists of three phases as depicted in Figure 1: 1) Identification of FRs and NFRs, 2) Specification of FRs and NFRs and 3) Integration of NFRs.

- **Identification of FRs and NFRs**

  From design documents, release notes, source code, extracted UML model and new user's requirements for a software system, we identify the new functional and non-functional requirements which need to be added to the system being re-engineered. Our focus is mainly on the desired non-functional properties of the software that it should meet to assure high quality software system.

- **Specification of FRs and NFRs**

  This phase consists of two parts. The first part is to specify the new FRs to be added into the extracted legacy model using standard UML notations. The second part is to search our knowledge-based NFR repository for any similar previously designed reusable NFR that the system may need. In case of the existing NFR design matches partially with the required NFR, the former one needs to be modified according to the required one and to be stored in the repository for future use. If no such NFR can be found, a new NFR template will be created according to the requirements.

- **Integration of NFRs**

  After specifying all non-functional requirements this phase just becomes a NFR weaver that weaves those desired NFRs with the FRs of the target system as shown in Figure 1. The following section elaborates

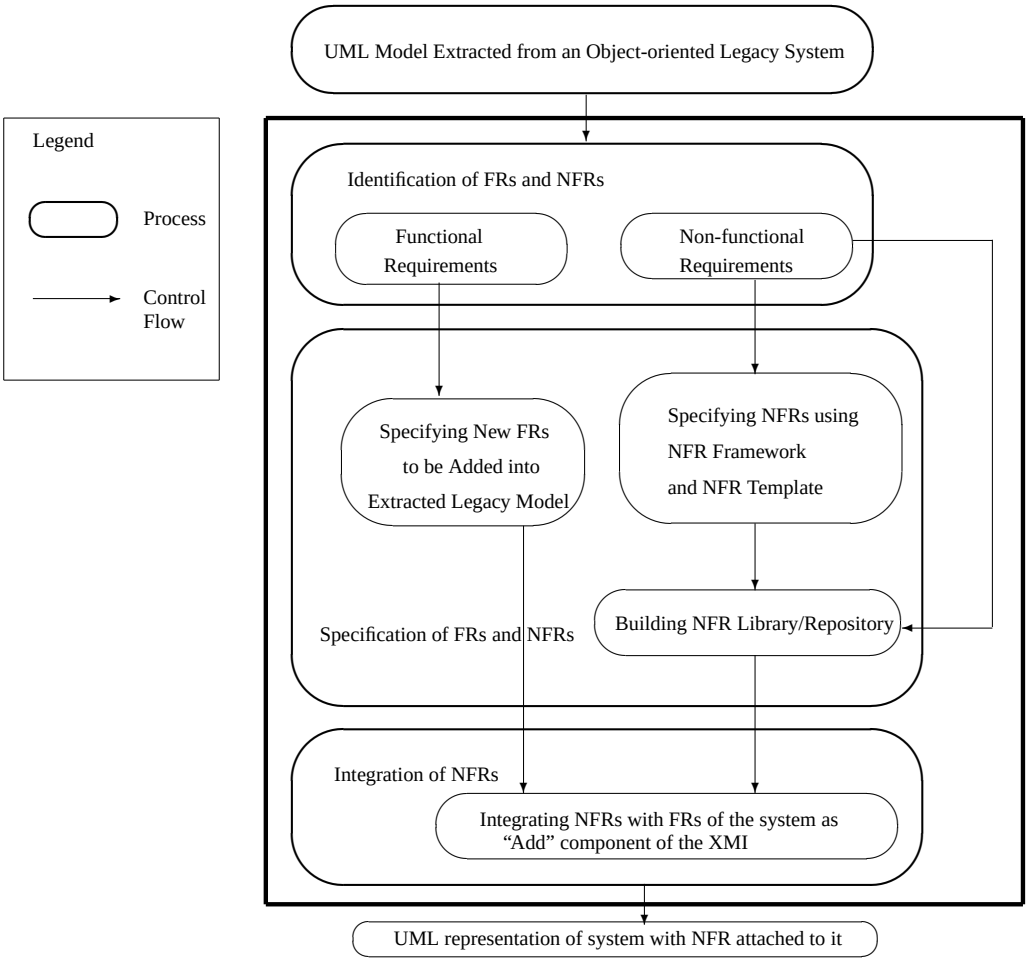

Figure 1. A Framework for Integrating NFRs

further on our proposed approach to make the reusable NFRs and to integrate them with the target model.

3.1 A Model for NFR Representation

A number of researchers and practitioners examined how a software or system successfully achieves quality attributes [2, 3, 4, 8]. To represent information about different software qualities, their interdependencies, evaluation of the NFRs upon the target system, detail techniques for specifying methods to arrive at the "target" or "destination" of the design process (operationalizing softgoal), we adopt the NFR framework proposed in [4]. The visualization of the operations of the NFR framework is done in terms of the incremental and interactive construction, elaboration, analysis, and revision of a *Softgoal interdependency graph (SIG)*.

Our work begins after evaluation of SIG. The evaluation procedure, defined by NFR framework, results in a subgraph of the SIG that needs to be integrated with the functional requirements of the target model. Finally, the *target system* in NFR framework describes the final solution of the particular NFR. Our framework maps this solution to achieve a particular NFR with standard UML notations and provides a weaver to weave it with the UML representation of a software system.

3.2 A UML Representation for NFR

We propose a high level notation to design the final solution of the target system to achieve the particular NFR with standard UML notations. Our proposed notation is based on the general actions that can be performed on any entity of UML model. For example, creating a class, inserting attributes to a class, inserting methods to a class, deleting a class, deleting attributes from a class, creating states, creating state-transitions, and so on. The goal of this notation is to provide all the actions which may be needed to create any type of UML diagram, and to specify a template for the UML representation of the NFR. The template is not executed at this point, rather it becomes an ordered set of instructions/commands (similar to a script in UNIX). While the framework is attaching the NFR to the functional requirements of the system, the template commands need to be executed to build the proper UML representation of the NFR.

3.3 A NFR Library/Repository

We use the same knowledge-based approach as NFR framework. We propose to build a NFR library/repository where the past experience, standard techniques and knowledge about particular NFR, the evaluated subgraph of SIG for achieving that NFR and the proposed NFR template to integrate it with the target model can be stored for future use. The library/repository is likely to evolve in the course of time and can help the developers in saving time by supplying previously designed NFR templates as reusable components.

3.4 Weaving NFR with the Target Model

A meta-model in UML describes the UML model by itself . Hence, the manipulation of the meta-model is same as the manipulation of any UML model. For this purpose, we have developed a meta-level NFR weaver where the weaver executes the weaving operations as specified in each NFR template applied onto the initial model. The weaver actually executes the commands specified in the NFR template and generates the corresponding UML representation of the particular NFR. The target model is also a UML model with NFR attached to it. For the compatibility of other UML tools, we are using the standard XMI to generate the UML model. In our framework, the NFR description resulting from SIG is a part of comment inside the XMI and the UML representation of the NFR is a part of the "Add" component of the XMI. By adding the NFR description resulting from NFR framework with the XMI of the model as a comment, we can store all the information for a particular NFR which can be further viewed using our weaver. The output becomes compatible with other tools as it still is in standard XMI format. By adding the NFR as UML "Add" component we can also separate the NFR from the main design of functional requirements of a software system and other operations can also be done on the added NFR. For example "deletion" and "modification"of the NFRs can be done without changing the main design of a software system.

4 A Case Study

A prototype has been written in Java programming language to implement the proposed framework in a semi-automated manner. Due to the space constraint, a small case study is presented as a proof of concept. The case study, we have chosen is a part of the Credit Card System described in [4]. Here is short summary of such system:

> "We consider an information system for a bank's credit card operation. A body of information on cardholders and merchants is maintained. In this highly competitive market, it is important to provide fast response time and accuracy for sales authorization. To reduce losses due to fraud, lost

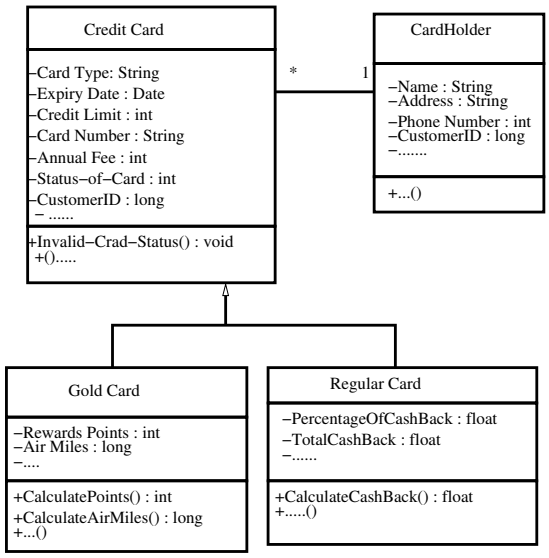

Figure 2. A Class Diagram of Credit Card System

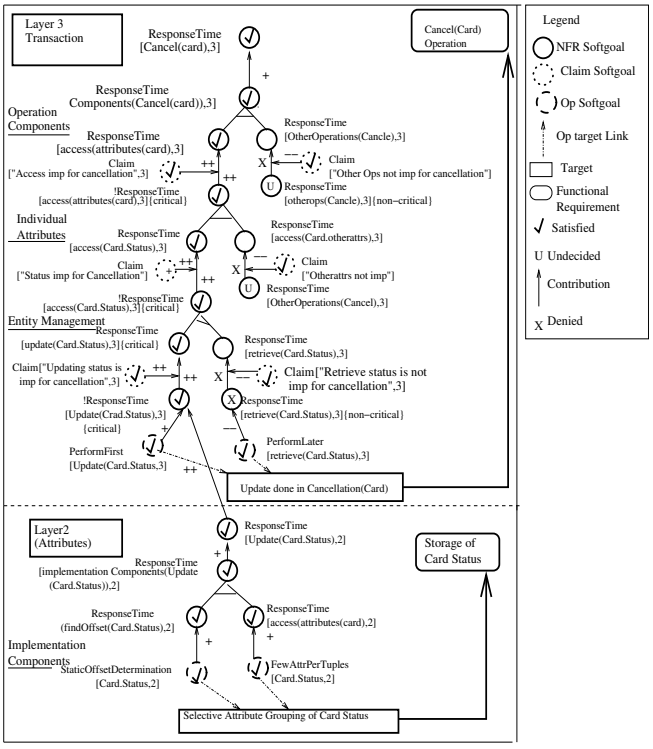

Figure 3. A SIG Performance for Credit Card System

and stolen cards must be invalidated as soon as the bank is notified."

The following sections elaborate further how each step of our proposed framework can be applied to the case study.

4.1 Identification of FRs and NFRs

In the selected case study, the functionality of Credit Card System includes maintaining information on *sales*, *card holders* and *merchants*. Transactions are authorized, and accounts are updated. Stolen cards are cancelled. The non-functional requirements of the credit card system may be performance and security of the transaction. Performance can also be divided as performance of card authorization and performance of card cancellation and so on. Here we consider the "Fast Response Time" to invalidate a card status (when it is lost or stolen) as the target NFR for the selected case study.

4.2 Specification of FRs and NFRs

Figure 2 shows a part of the class diagram of a Credit Card System specifying its FRs.

Figure 3 shows a partial SIG specifying the evaluation of impact of decisions after selecting operationalizing softgoals with respect to card cancellation operation. As shown in Figure 3, the "Performance" quality attribute of Credit Card System can be achieved by selecting attribute grouping of card status. One possible solution for this may be to physically store the card status separately along with few other attributes (card number for example). By applying such a mechanism, when a request comes for the cancellation of a particular card, the status of the card can be retrieved very quickly without any need to access irrelevant information (such as customer information, bonus points calculation) and the status can be updated to "invalid card" very easily. In the context of UML class diagram, one possible way to express this NFR is to create a new class (say "FastResponseTime") for the method Invalid-Card-Status() along with the attributes "Status-of-card" and the "Card-Number" (as primary key). This modification can result in deleting the attribute "Status-of-card" and the method "Invalid-Card-Status" from the original "CardHolder" class.

4.2.1 Buidling NFR Template and Repository

Figure 4 shows a possible set of NFR commands, from our proposed instruction set for class diagram, to specify the NFR "FastResponseTime". To create a template for the above NFR based on the solution discussed in the previous section, the following steps are required:

1. Accepting the join-point of the NFR "Fast Response Time" (here, method Invalid-Card-Status()).

```
Steps   Commands

0:      Input Parameters: Method(Single)<param0>
1:      AttributeList<result0> := getAttributesFromParentClass(<param0>)
2:      AttributeList<result1> := getSelectedInput("Attributes to Move",<result0>)
3:      AttributeList<result2> := getSelectedInput("Primary Keys",<result0>)
4:      ClassNode<result3> := createNewClass("FastResponseTime"
5:      <result3> := insertMultipleAttribute(<result3>,<result2>)
6:      <result3> := insertMultipleAttribute(<result3>,<result1>)
7:      <result3> := insertMethod(<param0>)
8:      deleteMethodFromParentClass(<param0>)
9:      deleteAttributesFromParentClass(<result1>)
10:     ClassNode<result4> := getParentClass(<param0>)
11:     createAssociation(<result3>,<result4>,1,1)
Output: (Step 7(result), Step 8(Operation), Step 9(Operation), Step 11(Operation)
```

Figure 4. NFR Template Commands

2. Selecting other attributes to move to the separate class for card status (here attributes Card-Number and Status-of-card).

3. Creating a new class (say, FastResponseTime) and building an association with the parent class.

4. Inserting those attributes defined in Step 2 and the method defined in Step 1, into this new class.

5. Deleting the attribute Status-of-card from the original class.

```
public class FastResponseTime{
        public FastResponseTime(){
        }

        void constructTargetSystem(){

            expectParameter(Method);
            MethodName param0 = getParameter();
            AttributeList result0 = getAttributesFromParentClass(param0);
            AttributeList result1 = getSelectedInput("Attributes to Move", result0);
            AttributeListresult2 = getSelectedInput("Primary Key", result0);
            ClassElement result3 = createNewClass("FastResponseTime");
            result3 = insertMultipleAttribute(result3,result1);
            result3 = insertMultipleAttribute(result3,result2);
            result3 = insertMethod(result3,param0);
            deleteMethodFromParentClass(param0);
            deleteAttributeFromParentClass(result1);
            ClassElement result4 = getParentClass(param0);
            createAssociation(result3,result4,1,1);
        }
    }
```

Figure 5. Java Class of NFR Template

In our prototype, we use the advantage of *dynamic class loading* feature of Java. We have built an NFR interpreter which interprets the commands in NFR template and generates the corresponding Java source code for the template. Each template is stored as a Java class in the NFR library/repository. The desirable template would be translated into the following piece of Java code as shown in Figure 5.

4.3 Integration of NFRs

The last step of our framework is to incorporate this NFR template with target design in UML notation. In order to attach the NFR with a method (e.g. Invalidate-Status-of-Card()in the design), the designer needs to supply the corresponding method as a parameter to the template. During the weaving, he/she needs to provide the necessary dynamic parameters to complete the process.

Figure 6 shows the new class diagram after the integration of NFR "FastResponseTime". After the integration of NFR "FastResponseTime" with the class diagram of Credit Card System, a new class named "FastResponseTime" is created with the attributes "Status-of-Card", "Card-Number" and the attribute "Status-of-Card" is deleted from the original class "CardHolder".

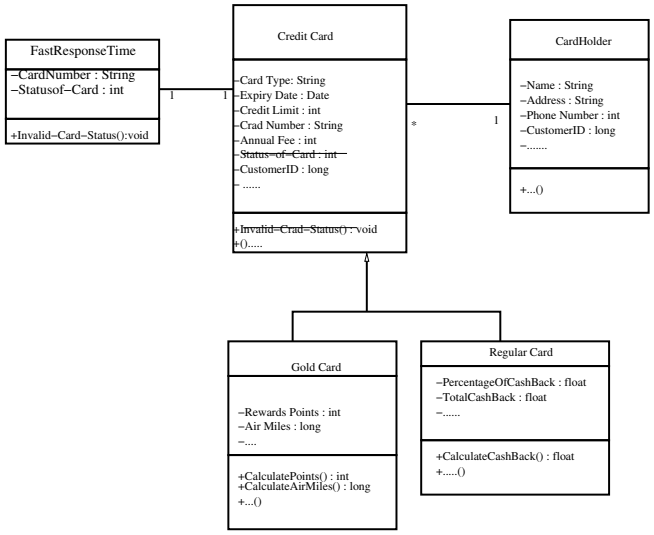

Figure 6. Adding NFR FastResponseTime

5. Conclusion and Future Work

We propose a novel framework for integrating non-functional-requirements with the UML design of a software system which can be applied during the re-engineering process of such a legacy system. The framework can also be used during forward engineering if the developers follow the standard XMI during their model design. Currently, we have built a prototype of the whole framework where the weaver supports the NFR design with class diagrams. The prototype also provides the advantage to draw the Softgoal-Interdependency graph (SIG) and to store the NFR template in the NFR library/repository along with the NFR information that comes from the NFR framework. We are now working on extending the prototype to incorporate all types of UML diagrams.

References

[1] Object management group. Unified Modeling Language Specification version 1.3 bata 1, 1999. available at uml.shl.com.

[2] J. Bergey, M. Barbacci, and W. William. Using quality attribute workshops to evaluate architectural design approaches in a major system acquisition : A case study. Technical Report CMU/SEI-2000-TN-010, Software Engineering Institute, Carnegie Mellon University, 2001.

[3] B. W. Boehm et al. *Characteristics of Software Quality*. Elsevier North-Holland Publishing Company, Inc., 1980.

[4] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishing, 2000.

[5] L. M. Cysneiros and J. C. S. do Prado Leite. Nonfunctional requirements: From elicitation to conceptual models. *IEEE Trans. Softw. Eng.*, 30(5):328–350, 2004.

[6] E. Dimitrov and A. Schmietendorf. Uml-based performance engineering possibilities and techniques. *IEEE Software*, 19:74–83, January/February 2002.

[7] W. Hürsch and C. V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, February 1995.

[8] International organization for standardization (iso). Technical report. Information Technology, Software Product Evaluation, Quality Characteristics and Guidelines for Their Use, ISO/IEC 9126, 1996.

[9] J.-M. J*ézé*quel, N. Plouzeau, T. Weis, and K. Geihs. From contracts to aspects in uml designs. In *Aspect-Oriented Modeling with UML, AOSD Workshop, Enschede, Netherlands*, April 2002.

[10] A. Moreira, I. Brito, and J. Arajo. Crosscutting quality attributes for requirements engineering. In *The fourteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, pages 167–174, July 2002.

[11] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: a process-oriented approach. In *IEEE Transactions on Software Engineering*, volume 8, pages 483–497, June 1992.

[12] S. Supakkul and L. Chung. Integrating frs and nfrs: A use case and goal driven approach. In *2nd International Conference on Software Engineering Research, Management and Applications (SERA'04),Los Angeles, CA*, pages 30–37, May 2004.

[13] L. Tahvildari and K. Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance*, 16(4-5):331–361, 2004.

[14] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *Journal of Systems and Software (JSS)*, 66(3):225–239, 2003.

Requirements-driven configuration of software systems

Yijun Yu Alexei Lapouchnian Sotirios Liaskos John Mylopoulos
Department of Computer Science, University of Toronto
{yijun, alexei, liaskos, jm}@cs.toronto.edu

Abstract

Configuring large-scale software to meet different user requirements is a challenging process, since end-users do not know the technical details of the system in the first place. We present an automatic process to connect high-level user requirements with low-level system's configurations. The process takes into account different user preferences and expectations, making configuration easier and more user-centered. Since it reuses a software system's configuration mechanisms, the configuration process is transparent to the system development. Moreover, it is very easy to plug different reasoning frameworks into the configuration process. As a case study, we have reengineered the Mozilla Firefox web browser into a requirements-driven software system, without changing its source code.

1. Introduction

Hardware evolution is governed by Moore's law – *CPU speed doubles every 18 to 24 months* [1]; on the other hand, software evolution is governed by Lehman's laws – especially the 2nd – *increasing complexity* [2]. As a consequence, computer hardware is getting ever-cheaper, e.g., an average workstation is typically a Windows box, which costs no more than $1000. On the other hand, the cost for employing an average developer is more expensive than buying 50 workstations, per year.

As the gap is widening, software maintenance cost dominates the operation of a software company. Managing and using large-scale software systems is becoming a grand challenge, sometimes even a nightmare, as too many parameters are to be configured in order for the software to be working properly by different clients and users. Configuring these is a headache for everyday users: Eclipse IDE, e-mail clients and web browsers such as Mozilla Thunderbird and Firefox, which target at populous and diverse user groups, several Linux kernels and distributions, and, of course, popular commercial software such as Microsoft Windows and Office Suite. These software systems typically contain millions of lines of code. The needs for managing such complex software engender the research in autonomic computing [1, 3].

Figure 1 presents the "Options" dialog window from Mozilla Firefox. A user is asked to provide very low-level details, such as "use TLS 1.0" or "Use SSL 2.0" etc. As shown on the screen, they are related to "Security", but it is not clear whether one should select all of them, one of them, or some combinations of them and how this impacts the attainment of the "Security" goal. Furthermore, what will the side-effects of these selections be on other goals such as "Performance", "Convenience", etc.?

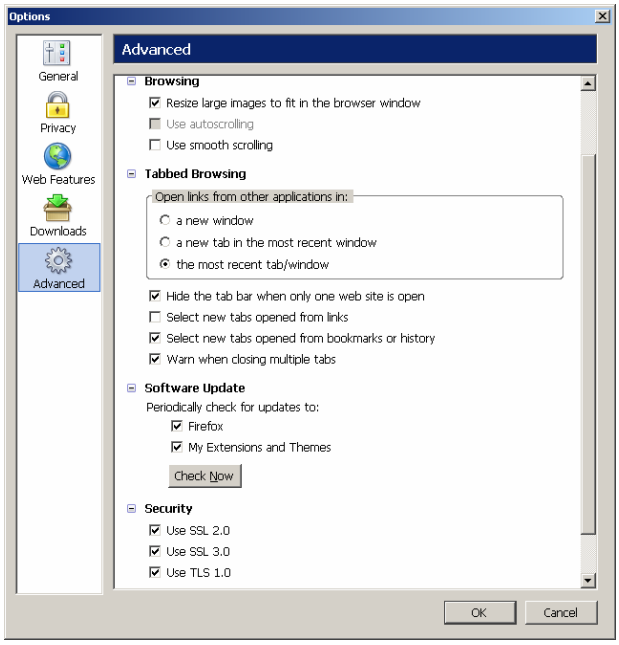

Figure 1. The Options dialog of Firefox

How do we reduce the overhead of controlling large-scale software systems to serve the clients better? How (in case the clients change their requirements) do we agilely reconfigure the software to fit the new client requirements? In this paper, we propose a way to tackle this problem by automating the configuration with goal

models [4]，which has been shown to be possible for a desktop application with an average number of configuration items [5]. Because we consider every individual's requirements in customizing large-scale software, the requirements-driven configuration process is strongly related to the concept of personal and contextual requirements engineering [6, 7]. In [5], for example, user's goals, skills and preferences are proposed as specific personalization criteria for customizing software and tailoring it to particular individuals. On the other hand, since requirements-driven configuration relies on the use of goals [8], a process for generating a goal model that appropriately explains the intentions behind an existing system needs to be considered [3]. In [9, 10], for example this is made possible through reverse engineering directly from the source code.

The process for such automated reconfiguration consists of two major steps. Firstly, one has to set up a goal model in order to connect the user's high-level requirements with the system's low-level configuration items. Secondly, the resulting mapping must be efficiently used by collecting user *preferences* over goals (one goal is more important than another) and *expectations* (a goal needs to be satisfied to a certain degree) and automatically carrying out the configuration.

Using this process, we have successfully configured the Mozilla Firefox browser and the Eclipse IDE for different types of users. The configuration step is fully automated and very efficient, making it well possible for the user to further analyze the resulting system by providing feedbacks.

The remainder of the paper is organized as follows. Section 2 explains the methodology in detail; Section 3 provides implementation details, and Section 4 discusses a case study of the requirements-driven configuration process on the Firefox Web browser. Section 5 discusses further work and concludes the paper.

2. Reengineering into requirements-driven configurable software

The aim of our process is to reengineer a legacy software system, such as Mozilla Firefox, into a requirements-driven reconfigurable system. Therefore, it calls for two necessary steps: (1) *reverse* engineering to understand the legacy system and (2) *forward* engineering to improve the legacy system.

In our case, a legacy system may or may not provide the source code to the reengineer. Thus, we use two kinds of reverse engineering techniques: (1) if the source code is available, the system can be reverse engineered to reveal the implemented goals or purposes of the programmer [11]; otherwise, (2) the system needs to be used and empirically examined in order to discover the alternative ways by which different users may customize the functionality of the system and consequently the alternative ways in which they may want their goals to be fulfilled [10].

Furthermore, once the goal-oriented requirements are obtained, an end user is simply asked to provide their preferences and expectations over the top-level abstract goals. This will drive the software configuration automatically. The degree of automation will depend on how advanced the user is and how much awareness of the low-level configuration details are demanded. Thus, advanced users may employ the method only to obtain a suggestion on how they should configure their system in order to better accommodate their preferences and expectations.

2.1 Reverse engineering for goal models

The objective of reverse engineering in our process is to detect traceability between the low-level implementation with the high-level requirements. Traceability between user goals and the implementation allows the users to understand the system and subsequently configure it in more abstract and less system-oriented way. It is also important to make the user aware of why the system makes certain choices.

In our approach, we do the reverse engineering in two steps:

1. Establish a goal model of the software system;
2. Associate the leaf goals with the configuration items.

A *configuration item* is a variable that can take certain values. A software system can be seen as a huge variability space induced by a large number of configuration items. Some of the configuration items are domain-specific, while others are domain-independent. For example, to configure the look and feel is a taste of the individual, whereas to configure the security task is subject to the software domain. A user's goal model can narrow down the search space by assessing the configuration items.

2.2 Forward engineering with goal models

Having identified the goal models, the objective of forward engineering in our process is to collect individual user preferences and expectations and translate them into software configurations. It is also done in a few steps:

1. **Querying.** Obtain user's preferences and expectations over the high-level goals;

2. **Reasoning.** Convert the user input into satisficing labels of the high-level goals and propagate them downward until leaf goals are reached; Note here the term *satisfice* was used by Herbert Simon [12] to denote the idea of partial satisfaction. The qualitative analysis of the NFR framework [13] is centered on the idea of satisfice.
3. **Configuring.** Convert the leaf goals satisficing labels into values of the configuration items.

Both steps 1 and 3 depend on the software being investigated. During the querying step, a user is asked to either directly provide the preferences and expectations over the goals, or to indirectly provide this information though answering an elicitation questionnaire. The configuring step associates each configuration item with a *default* value in order to attain a certain level of satisfaction for the leaf-level goals.

The reasoning step is independent of the domain of the system to be configured, and is based on the trade-off algorithms discussed in the following section.

3. Implementation

In this section, we briefly discuss the implementation of the methodology. We first describe the reverse engineering approach to establish a goal model. Then, the design of the tradeoff algorithms based on existing goal reasoning algorithms ([8, 14]) is explained. Finally, we show how the query and configuration steps are carried out automatically.

3.1 Reverse engineering

A goal model consists of a set of AND/OR decompositions that *refine* a high-level goal into a set of low-level subgoals. On top of these rules, a set of quantitative *contributions* shows how the satisficing of one goal influences the satisficing of the others. Such a quantification can have probabilistic semantics [8] or it may be cast into a framework of qualitative contribution links. Thus, we can use contribution links such as HELP (+), HURT (-), MAKE (++) or BREAK (--), to show how the satisfaction of the origin goal influences the satisfaction of the target goal.

The source of a goal model can be recovered from the system structure and behavior. In terms of structure, a system/subsystem decomposition paradigm, which follows the *divide and conquer* metaphor, is often a natural match for the AND/OR goal decompositions. For example, inheritance can be seen as the implementation of an OR decomposition of the subject whereas aggregation may be the implementation of an AND decomposition. In terms of behavior, the system achieves certain goals by performing transitions from one state to another. Here, the state/substate hierarchy that can be defined in a statechart has been shown to naturally map to the respective goal/subgoal decomposition graphs[15]. Static program analysis using program slicing techniques can reveal the system's implemented goals [9]. Observing the execution log/trace of the system can also reveal patterns in its dynamic behavior [10]. Combined with a testing framework one can make sure certain functional goals are indeed satisfied [9, 10].

Leaf-level goals may be associated with Boolean predicates on the value of one to many configuration items. For configuration items that are already Boolean, such as "use SSL 2.0" or "use SSL 1.0", such mappings are straightforward. For non-Boolean configuration items, such as a "keeping history record for N days" an extra step is required to find the default value of the configuration item that satisfies the goal. For example, we can represent the leaf-level goal "Keep a good record of my web surfing history" as a Boolean predicate "N >= 5", and associate the fully satisficed value of the goal with "N=10" and the fully denied value of the goal with "N=0". This way, a direct mapping is set from the configuration of domain-specific parameters to the configuration of the goal model.

3.2 Tradeoff algorithms

When a goal is decomposed into multiple alternatives (OR-subgoals), the contribution of each subgoal to the satisfaction of top-level goals can be compared with the expectations and preferences, in order to rate the choices and thus make decisions.

Bottom-up reasoning propagates the labels that describe the degree of satisfaction of leaf goals upwards to obtain the corresponding labels for the top-level goals [8]. This can be used to validate the requirements.

Top-down reasoning propagates the labels of the top-level goals downwards to obtain the labels for the minimal number of leaf-level goals [14]. This can be used to predict the minimal configuration that can satisfy the user's requirements. Since the top-down reasoning relies on a *satisfiability problem*[1] (SAT) solver [16, 17] which deals with binary propositions, it is important to design an encoding mechanism such that at least discrete labels (full/partial satisficing/denial) of goals can be translated into the binary propositions.

3.3 User interface and questionnaire design

An interface to the configuration system consists of a dialog and/or a questionnaire wizard. In the dialog, each

[1] That is, deciding whether a given Boolean formula in conjunctive normal form has an assignment that makes the formula "true."

top-level hard goal is presented as a *checkbox*, whereas each top-level softgoal (e.g. performance, security, usability) is presented as a *slider* by which the satisficing expectation is set. Preferences are shown by the order of the sliders from top to bottom. Although a slider-based user interface design can directly present the needed input, it is not guaranteed that all the user's expectations can be met by the system design *at the same time*. For example, a full satisfaction of performance, security, maintainability and usability goals is simply impossible. The interdependency and constraints among these goals are defined by the underlying goal model. Thus we also designed an alternative wizard to ask user a set of elicitation questions in order to derive the expectations and preferences with respect to the goals. In these questions, we avoid using technical terms, rather, using familiar terms to everyday user. For example, "Are you using the browser with a public-domain computer?" The simple Yes/No answer to such questions can lead to elicited preference such as whether "Privacy" is important or not. Thus for elicitation, we can use a goal model which connect the preferences/expectations of the high level goals with answers to concrete questions at the leaf level and use bottom-up label propagation to obtain the preference/expectation labels as an input for the configuration step.

3.4 Configuration step

The configuration of the system is done automatically. First, the software system is analyzed for its configurability in terms of whether there exists a persistent record of the configuration (if our configurator interacts with the subject software through a file interface) or an in-memory API for its configuration (if our configurator interacts with the subject software directly through APIs).

Based on the configuration in the goal model (the selected leaf-level goals), a script is generated to populate the configuration data with the default values associated with the leaf-level goal satisfaction labels. Since the reverse engineering step has already produced the appropriate mapping, this task is now quite straightforward. The last step is to automate the reconfiguration by running the script, either before restarting the subject software or during the execution of the software system.

4. Firefox: a case study

We represent user high-level requirements in an XML-based input language, as follows.

```
<input:model>
<soft name= "Performance">
          <rule op="AND"/>
          <soft name= "Browsing Performance"/>
          <soft name= "System Performance"/>
</soft>
<soft name= "Usability">
    <rule op="OR"/>
    <soft name= "Ease of Search"/>
    <soft name= "Convenient access to Information"/>
    <soft name= "User Tailorability">
        <rule op="OR"/>
        <soft name= "Programmability"/>
          <soft name= "User Flexibility"/>
    </soft>
</soft>
<soft name= "Security">
   <rule op="HURT" target="System Performance"/>
   <rule op="HURT" target="Browsing Performance"/>
</soft>
<soft name= "Allow changes in Content Appearance">
    <rule op="HELP" target="User Flexibility"/>
</soft>
<goal name= "Filter Advertisement/Spyware/Popups">
   <rule op="HELP" target="Performance"/>
   <rule op="HELP" target="Security"/>
   <rule op="HURT" target="Content Availability"/>
</goal>
</input:model>
```

In this input language, a *model* is given by a list of root *goal*s which are recursively decomposed in a nested XML element structure. A *soft*goal is a goal that can be satisfied to a degree less than 1. It usually represents quality attributes. A number of *rule*s show what kind of decomposition was used for a goal or softgoal, or which kind of contributions was used between a *source* hardgoal and a *target* softgoal.

Each user provides a profile including the preferences and expectations for the softgoals:

```
<input:profile>
<soft name="Security" rank="4" value="6" />
<soft name="Allow Interactive Content" rank="8" value="8"
/>
<soft name="Convenient Access to Information" rank="10"
value="10" />
<soft name="Performance" rank="9" value="1" />
<soft name="Content Availability" rank="1" value="10" />
<soft name="Allow changes in Content Appearance"
rank="6" value="4" />
<soft name="User Flexibility" rank="3" value="6" />
<soft name="Speed" rank="7" value="3" />
<soft name="Programmability" rank="3" value="8" />
<soft name="Modularity" rank="5" value="1" />
<soft name="Usability" rank="2" value="6" />
</input:profile>
```

For every root softgoal, a *rank* attribute represents the partial order among the preferences and a threshold *value*

represents the expectation from the user. The profile can be generated from a user interface dialog (Figure 2).

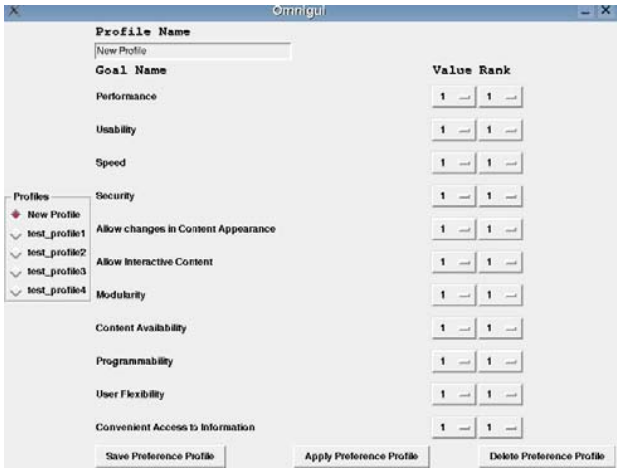

Figure 2. A simplified user preference dialog as the interface to the configurator

The reasoning algorithm is invoked by the configurator command automatically, to produce an output as follows:

```
<output:configuration>
<goal name="adFilterStrength" value="on" />
<goal name="tabBrowsingOn" value="off" />
<goal name="cookiesEnabled" value="off" />
<goal name="daysToCachePages" value="on" />
</output:configuration>
```

The goal model can be visualized as a goal graph and the reasoning can be invoked and its results shown in OpenOME [18], our requirements engineering tool, where both bottom-up and top-down goal reasoning algorithms are implemented and can be invoked by the two buttons on the toolbar (Figure 3). Behind the scenes, an XSLT script fully automatically generates the corresponding property configuration in the Firefox default installation directory.. The following JavaScript script code is an example of such property configuration:

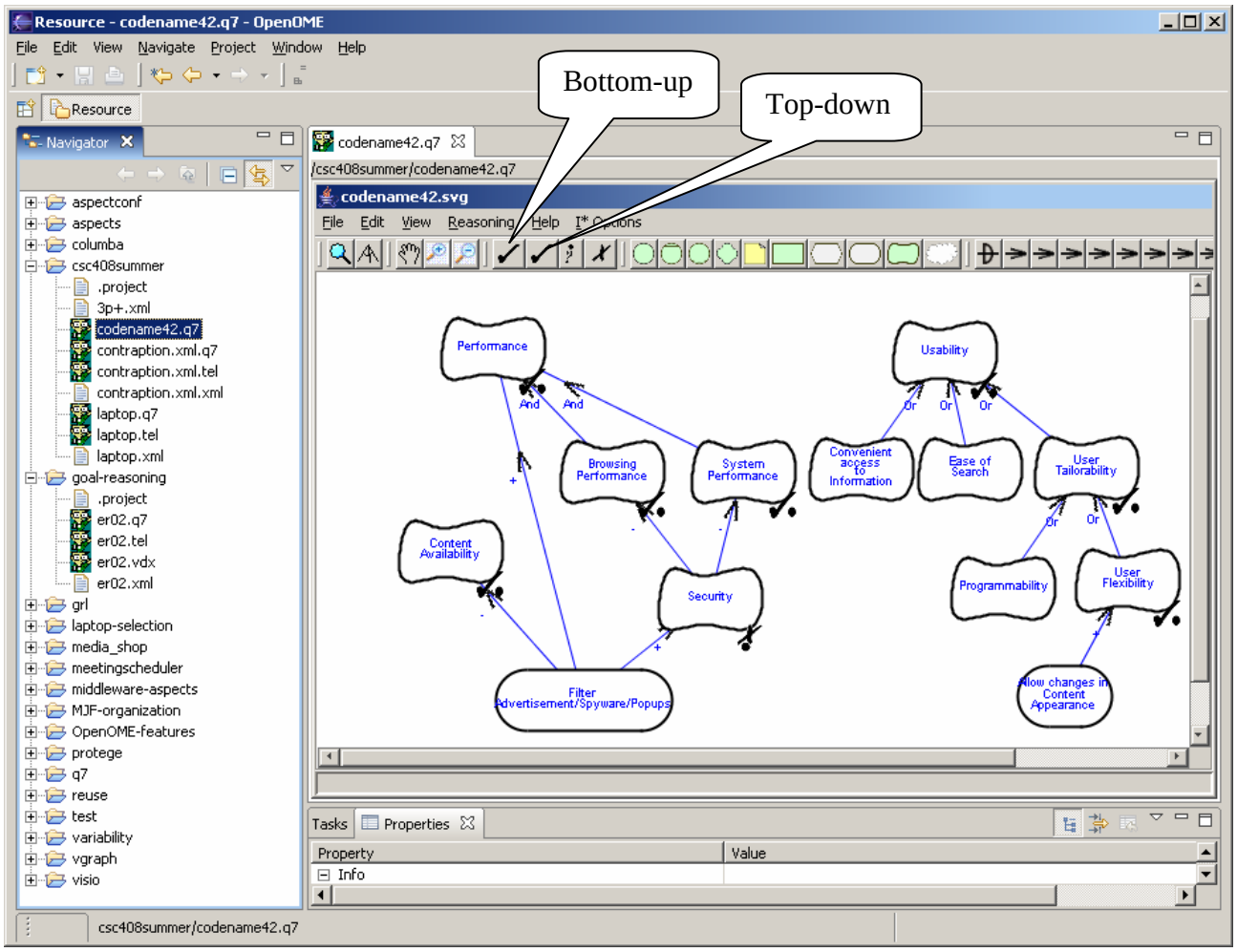

Figure 3. The goal model and its reasoning in OpenOME, an Eclipse plugin for requirements engineering

```
user_pref("network.image.imageBehavior", 2);
user_pref("network.cookie.cookieBehavior", 2);
user_pref("webdeveloper.disabled", false);
user_pref("browser.display.use_document_colors", true);
user_pref("javascript.enabled", false);
user_pref("webdeveloper.disabled", false);
user_pref("adblock.enabled", true);
user_pref("tidy.options.browser_disable", false);
user_pref("font.size.variable.x-western", 19);
user_pref("image.animation_mode", "normal");
user_pref("extensions.prefbar.display_on", 0);
user_pref("security.enable_java", false);
user_pref("security.default_personal_cert", "Select
Automatically");
user_pref("browser.cache.disk.enable", false);
```

5. Conclusion

Through the Mozilla Firefox case study we show how goal-oriented requirements can be used to guide the configuration process automatically. The goal models are provided by domain experts, the user profiles are obtained by the users directly through a simplified user interface, and the configuration is carried out without further human intervention. Currently, we are investigating how to apply the requirements-driven configuration mechanism to other applications and how to detect problems that reconfiguration may cause when it is performed while the software system is running. We also plan to implement a Firefox extension plugin to expose our tool to the larger user community and to solicit feedback from users.

6. Acknowledgement

Much of the implementation is done by all of our 26 undergraduate students in the Software Engineering course offered in the summer of 2005 at the University of Toronto [11]. Some examples in this paper are taken from the *codename42* project team including Dimitri Stroupine, Faiz Hemani, Hareem Arif, Sani Hashmi and Zia Malik. The authors would also like to thank Xin Gu for giving XSLT tutorial to the students.

7. References

[1] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Syst. J.*, vol. 42, pp. 5-18, 2003.

[2] M. M. Lehman and J. F. Ramil, "Evolution in software and related areas," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*. Vienna, Austria: ACM Press, 2001.

[3] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu, "Towards requirements-driven autonomic systems design," in *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*. St. Louis, Missouri: ACM Press, 2005.

[4] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," in *Selected Papers of the Sixth International Workshop on Software Specification and Design*: Elsevier Science Publishers B. V., 1993.

[5] B. Hui, S. Liaskos, and J. Mylopoulos, "Requirements Analysis for Customizable Software Goals-Skills-Preferences Framework," in *Proceedings of the 11th IEEE International Conference on Requirements Engineering*: IEEE Computer Society, 2003.

[6] A. Sutcliffe, S. Fickas, and M. M. Sohlberg, "Personal and Contextual Requirements Engineering," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering*: IEEE Computer Society, 2005.

[7] S. Fickas, "Clinical requirements engineering," in *Proceedings of the 27th international conference on Software engineering*. St. Louis, MO, USA: ACM Press, 2005.

[8] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, "Reasoning with Goal Models," in *Proceedings of the 21st International Conference on Conceptual Modeling*: Springer-Verlag, 2002.

[9] Y. Yu, Y. Wang, S. Liaskos, A. Lapouchnian, and J. Mylopoulos, "Reverse Engineering Goal Models from Legacy Code," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering*: IEEE Computer Society, 2005.

[10] S. Liaskos, A. Lapouchnian, Y. Wang, Y. Yu, and S. Easterbrook, "Configuring Common Personal Software: A Requirements-Driven Approach," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering*: IEEE Computer Society, 2005.

[11] Y. Yu, A. Lapouchnian, S. Liaskos, and X. Gu, "The Software Engineering summer course (CSC408H1S)," *http://www.cdf.toronto.edu/~csc408h/summer*, 2005.

[12] H. A. Simon, *The Sciences of the Artificial*: Massachusetts Institute of Technology, 1996.

[13] L. Chung, B. A. Nixon, E. S. K. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*. Boston Hardbound: Kluwer Academic Publishers, 1999.

[14] R. Sebastiani, P. Giorgini, and J. Mylopoulos, "Simple and Minimum-Cost Satisfiability for Goal Models," in *Proc. CAiSE*: LNCS, 2004.

[15] Y. Yu, J. Mylopoulos, A. Lapouchnian, S. Liaskos, and J. C. S. d. P. Leite, "From stakeholder goals to high-variability software designs," University of Toronto CSRG-509, 2005.

[16] S. A. Cook and D. G. Mitchell, "Finding Hard Instances of the Satisfiability Problem: A Survey " in *Satisfiability Problem: Theory and Applications*, vol. 35, *Discrete Mathematics and Theoretical Computer Science*, J. G. Dingzhu Du, and Panos M. Pardalos, Ed.: American Mathematical Society, 1997, pp. 1-17.

[17] D. Le Berre, A. Parrain, O. Roussel, and L. Sais, "SAT4J: A satisfiability library for Java," 2005.

[18] Y. Yu, E. S. K. Yu, L. Liu, and J. Mylopoulos, "The OpenOME requirements engineering tool," in *http://www.cs.toronto.edu/km/openome*, 2005.

Extracting Business Processes from Three-Tier Architecture 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

To minimize the overall expense and to reduce time to market, organizations either modify existing source code to meet the new requirements, or reuse existing components in new systems. Unfortunately, many software systems never have up-to-date documentation. Absence of good documentation increases the challenges for maintenance because the developers must read through the source code to understand the behaviors of the systems and to locate business logics manually. In this paper, we proposed an automatic method to generate the business processes for the three-tier architecture systems by identifying the business data and business policies in the source code.

1. Introduction

Software maintenance has become one of the most critical and longest stages in the life cycle of software systems. When the documentation of a system is lost, out-dated or unavailable, it is essential to extract important information, such as architectures, designs or requirements before maintenance or reuse can take place. Often, the only reliable source for such information is either in the mind of the developers or deeply buried in the source code. It is labor-intensive to manually scan the source code to extract the documentation.

A business logic is "a requirement on the *condition* or manipulation of data expressed in terms of the business enterprise or application domain" [1] and a business policy specifies the rules and conditions on when and where the business logic should be executed [2]. The business logics and policies form the business processes which organizations specify how they run their business. For instance, when a book is ordered from an online store, the business process consists of checking the availability of the books, restocking the inventory if the book is out-of-stock, validating the credit card and shopping to the customer; each of the individual tasks is a business logic and the condition to execute a specific task is the business policy (i.e. "book is out-of-stock" is the condition for "restocking the inventory"). It is, however, the nature of the business process to change fast in order to adapt to the market dynamics [3]. As a result, automatic or semi-automatic methods for business process extraction are essential for organizations to remain competitive by increasing the response rate to customers' demands and by reducing the costs of reengineering tasks.

For e-commerce systems, the three-tier architecture has many advantages over the one-tier and two-tier architectures in various ways. The separation of layers between user interface, business logic and database not only distinguish the functionalities between components but also provide additional information including the business data and the explicitly defined interfaces that used by business logics to communicate to the database. In this research, we propose a technique to automatically extract the business process by analyzing the communication channel and the information flow between the business layer and the database layer from the source code of the three-tier architecture systems. Based on the database operations and the information flow of the business data, we are able to identify business policies, logics and processes from the source code.

The paper is organized as follows. We discuss the benefits of the three-tier architecture and the relationship between the three-tier architecture and business logic extraction in Section 2. Section 3 discusses the method we proposed to extract business logics by business data and policies. Our approach identifies the business data and policies from the database operations explicitly invoked in the source code. We show a case study of our approach and the improvements over our previous work from [4] in Section 4. Finally, Section 5 gives the conclusion of this paper.

2. Software Architecture and Business Logic Extraction

Currently, most e-commerce applications adapt three-tier architecture. The three-tier architecture has higher maintainability than the traditional one-tier or two-tier architectures because the components are well separated and the interface between components is well-defined.

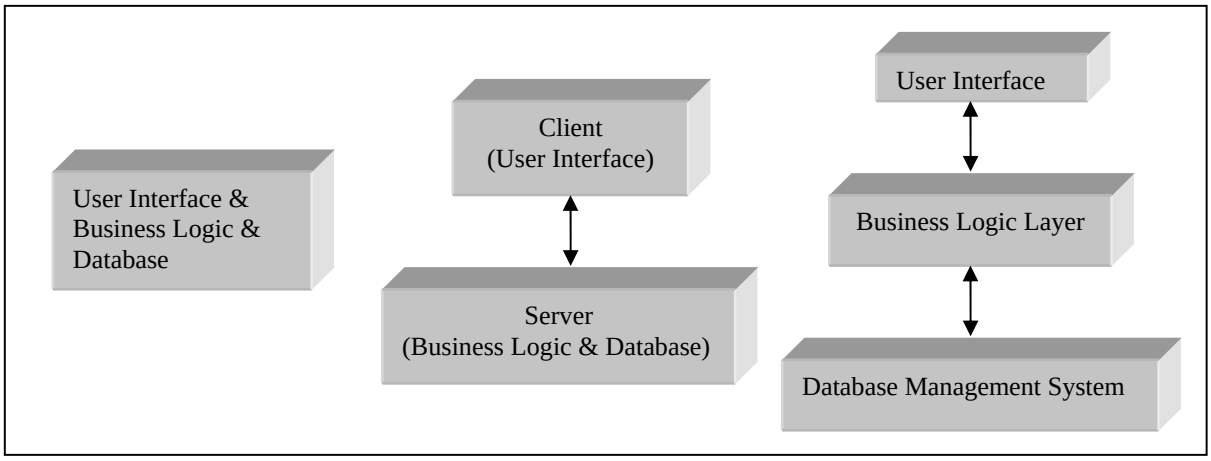

Figure 1 – One-Tier, Two-Tier and Three-Tier Software Architectures

In the three-tier architecture, user interface, business logic and database management are differentiated, as opposed to the mixture of functionalities in the one and two-tier architectures (Figure 1). Three-tier architecture provides numerous advantages over one-tier and two-tier architectures for reverse engineering and design recovery. In one and two-tier architectures, the source code entities implementing different components are interweaved with each other. It is therefore very difficult to separate the business logic components from others because of the absence of the clear partition. In three-tier architecture, the business logic and business policy components are implemented in a separate layer (the middle layer) and communicate with the user interface (UI) and the database management systems (DBMS) via external APIs. This explicit division between different functionalities not only partitions business data, policy and logics code in the source code but also clearly points out the communication of the business data and the corresponding fetch/update operations (such as J2EE EJB and embedded SQL) to the DBMS.

3. Business Process Extraction

To extract business logics from the source code, it is important to identify the relevant business entities that constitute business logics. The business entities include the business data and the business policies where the data are the inputs/outputs affecting the dataflow of the business logics and the policies control the execution path and control flow of the business logics. We employ forward and backward tracing to identify the exact location of the business logics. After the business logics are identified, the business process that determines the communications between the business logics and policies can be generated by static tracing as proposed in our previous work in [4].

3.1. Business Data

Organizations carry out their decisions based on the business policies, logics and data (input and output of the business logics). Often the input data of the business logics are fetched and the output data are updated to the DBMS. The explicit database fetch and update operations, such as the getters and setters of the J2EE® EJB objects respectively, identify the input and output of the business logics and therefore signal the presences of the business logics. Once the locations of the inputs and outputs are recognized from the source code, the static tracing technique is applied to identify where and how these data are used.

Fetch operations are often used to retrieve input data of the business logics from database. Centered on the business data, the execution of the business logics are dependent on the values of the data to make decisions and to compute the outputs of the logics. Therefore we apply the forward tracing, which analyzes the dataflow toward the direction of the execution, to the input data until the end of their lifetime. The usages of the business data of the fetch operations are discussed in the following:

a) Business policy conditions (also see Section 3.2) - Most of the decisions are made according to the information and data fetched from the database. For instance, the percentage of tax is charged differently to an online purchase according to the destination shipping address. As a result, the business data can indicate the business policy conditions and direct the execution of the business logics, as shown in Figure 2.a.
b) Computations of the output values of business logics - The output values are often computed from the business data. For instance, the final price for a purchase is calculated by the original price plus the tax. The calculated value is either used as

input of other business logics (i.e. to calculate the total) or updated to the database (i.e. to save the purchase history of a customer). Assignment statements with the business data on the right hand side is considered as an example of this computation, as shown in Figure 2.b.

c) Inputs of the user-defined functions - Method invocations are user-defined functions that perform specific tasks. Similar to b), a method can return a derived value based on the input data fetched from database; therefore, the method can be treated as one business logic where the business data are inputs and return value is the output of the logic, as shown in Figure 2.c.

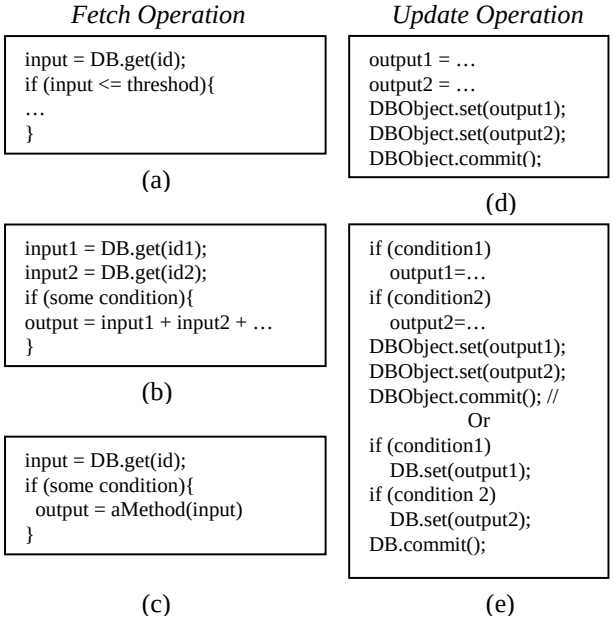

Figure 2 – Database Fetch and Update Operations and the Corresponding Business Logics

On the other hand, update operations store the result outputs of the business logics to the database. Therefore, we apply the backward tracing, which analyzes the dataflow toward reverse direction of the execution, to the parameters of the update operations. The backward tracing is able to identify the conditions and the locations where the outputs are computed or assigned. The usages of the business data of the update operations are discussed in the followings:

d) Computations performed under the same condition - If all of the parameters of the update operations are computed under the same condition (i.e. in sequence), the multiple values should be treated as a single composite data because the sequential derivations imply the grouping of the data that are updated to the database. As a result, the completion of the update operation (i.e. transaction committed to database) is the location of the business logic, as shown in Figure 2.d.

e) Computations performed under the different conditions - If any of the parameters of the update operations are computed under the different conditions, it implies that each value may or may not be present in the final transaction. Therefore, we can apply backward tracing to parameters of the update operations and identify where the computation is performed. We consider the individual computation as a business logic, as shown in Figure 2.e.

3.2. Business Policies

Business data are the center of the data flow of the business logics, and business policies determine the control flow of the business process and the execution of the business logics. In other words, business logics will be executed only when the current system states meet the conditions of the business policies. As a result, business policies signal the presences of the business logics.

However, not all conditions that guide the execution sequence of the business process are business policies. Many of the conditions are only specific to the programming languages. For instance, a program may check for current availability of the external services and it decides to wait (i.e. stay at the current state instead of moving to the next one if we treat the business process as a finite state machine). A program may also check the initialization of a variable and assign the variable with a default value if it is not initialized to avoid errors. Such conditions are only specific to program domains and have no real meaning in the business domains; thus they cannot be business policies. In our research, we define the business policies as the followings:

a) Business Policies specify the *constraints* that affect the behaviors, i.e. in an online shop, whether the purchased items can be downloaded, such as e-books or software, or it must be physically shipped, such as real books or computers.

b) Business Policies specify the *derivation of conditions that affect the execution flow*, i.e. the destination region of a shipping item according to the address and postcode.

c) Business Policies specify the *conditions* under which the computation is performed, i.e. the amount of tax to be charged to the item according to the regions.

According the above definitions, we can identify and extract business policies from the following scenarios:

1. Inside different branches of a CHOICE, an object invokes different methods or same method with different parameters, as shown in Figure 3.a. The conditions of the CHOICE are business policies.
2. Inside different branches of a CHOICE, a variable or variables are computed from different values, as shown in Figure 3.b. The conditions of the CHOICE are business policies.
3. A condition of a CHOICE is derived from business data in advance, as shown in Figure 3.c. Such condition is a business policy.

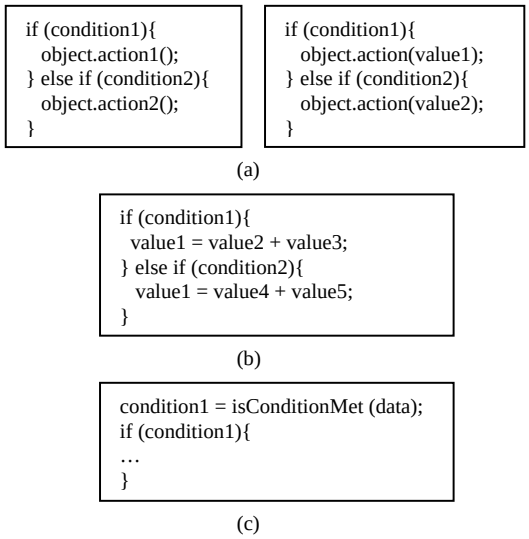

Figure 3 – Three cases of Business Policies

After business policies are identified from the source code, each execution sequence under the different conditions is grouped together as one business logic with the exception that other business logics are identified inside the execution sequence.

Business data and business policies have overlaps over each other in many occasions where the former handles the dataflow aspect and the latter handles the control flow aspect. Often, policy conditions are derived from business data. For instance, the shipping cost might be calculated based on the destination whose value, such as postcode, is fetched from the DBMS. Consequently, the overlap in the business logic identification is not redundant, but is an enforcement that strengthens our confidence in the business process and logic extraction.

4. Case Studies

To demonstrate the effectiveness of our proposed approach to identify the business logics, we performed case study on industrial e-commerce applications.

We analyze control conditions and entity behaviors to identify business data, policies and logics from the source code as discussed in Section 3. Furthermore, we generate complete trace records by applying static tracing technique that simulates the control flow of the source code [5] from the entry to the exit of the process (i.e. from *main* method of a program until this program terminates). The trace records contain the identified business policies and logics with the inputs and outputs as well as the execution sequence of the logics. One example of the extracted process, namely *Update TA Spending for Limit Check*, is shown in Figure 4 (simplified version of our original output).

```
1  <Decision expression="hasNext"/>
2  <Loop condition="yes" endline="236" startline="234">
3        <Task name="abRightToBuyTC.setInitKey_referenceNumber"/>
4        <Choice expression="strTCCurrency==null">
5        <Yes>
6              <Task name="abObligationToBuyTC.setInitKey_referenceNumber"/>
7        </Yes>
8        <Choice expression="bMultipleTradingIds">
9        <Yes>
10             <Decision expression="i<iabOrderItemArray.length"/>
11             <Loop condition="yes" endline="330" startline="320">
12             <Task name="dPurchaseAmount=dPurchaseAmount.add(getTaxAmountInEJBType())"/>
13             <Task name="dPurchaseAmount=dPurchaseAmount.add(getShippingChargeInEJBType())"/>
14             <Task name="dPurchaseAmount=dPurchaseAmount.add(getShippingTaxAmountInEJBType(
15             <Task name="dPurchaseAmount=dPurchaseAmount.add(getTotalAdjustmentInEJBType())"
16             <Task name="convertMonetaryValue"/>
17             </Loop>
18       </Yes>
19       <No>
20             <Taskname="convertMonetaryValue"/>
21       </No>
22       <Choice expression="spendingLimit!=null">
23       <Yes>
24             <Task name="findTradingPurchaseTotal"/>
25             <Task name="findTradingRefundTotal"/>
26       </Yes>
27       </Choice>
28 </Loop>
```

Figure 4 – "Update TA Spending for Limit Check" Process Extracted from Implementation

Compared with our previous work in [4], the new approach is able to identify more business logics accurately. We were able to identify four business logics from *Update TA Spending for Limit Check* by the method discussed in [4]; the number of business logics is increased to ten. This improvement has shown the effectiveness of the new approach.

In Figure 4, the business logics located on Line 3, 6 and 12~15 are newly identified. Furthermore, the pre-derived condition of the CHOICE on Line 8 is a business policy for the reason that the same method *convertMonetaryValue* was invoked in both branches, as discussed in Section 3.2. However, before the aid of business data, this policy had little significance because both branches contained only an identical logic, namely *convertMonetaryValue* (Line 16 and 20), whereas the *Yes* branch actually performs a number of additional computations. By considering business data, we realize the significance of the computations and capture them as the business logics (Line 12~15).

As afore mentioned, we are able to prove that business data and policies play important roles in the business processes in three-tier architecture systems. By considering business data and policies, we can identify business logic and generate complete and precise business processes from the source code automatically.

5. Conclusion

The three-tier architecture defines explicit interfaces to the DBMS and indicates the input and output data for the business logics. It offers good starting points for the business process and logic extraction. We identified the business data from the database operations and the business policies from the behaviors of the objects and the calculation of the outputs. Combining the business data and policies together, we automatically located the business logics. By utilizing static tracing technique, we generated complete records that outline the interactions between business logics to form business processes. Finally, our case studies demonstrated the improved effectiveness of our new approach by doubling the number of the identified business logics and by extracting more precise computations performed from the source code.

References

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

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

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

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

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