Reverse Engineering Goal Models from Legacy Code

Yijun Yu[1], Yiqiao Wang[1], John Mylopoulos[1], Sotirios Liaskos[1], Alexei Lapouchnian[1],
Julio Cesar Sampaio do Prado Leite[2]
[1]Dept. of Computer Science, Univ. of Toronto, {yijun,yw,jm,liaskos,alexei}@cs.toronto.edu
[2]Dept. of Computer Science, PUC-Rio, julio@inf.puc-rio.br

Abstract

A reverse engineering process aims at reconstructing high-level abstractions from source code. This paper presents a novel reverse engineering methodology for recovering stakeholder goal models from both structured and unstructured legacy code. The methodology consists of the following major steps: 1) Refactoring source code by extracting methods based on comments; 2) Converting the refactored code into an abstract structured program through statechart refactoring and hammock graph construction; 3) Extracting a goal model from the structured program's abstract syntax tree; 4) Identifying non-functional requirements and derive softgoals based on the traceability between the code and the goal model. To illustrate this requirements recovery process, we refactor stakeholder goal models from two legacy software code bases: an unstructured Web-based email in PHP (SquirrelMail) and a structured email client system in Java (Columba).

1 Introduction

A software *reengineering* process follows a horseshoe model by first recovering lost abstractions (e.g., elements of design and/or requirements) through *reverse engineering* [2, 21] (also known as *design recovery* [19]), and then pushing these abstractions forward into low-level implementations through *forward engineering*. In the initial horseshoe proposal [18] and all subsequent research, the lowest level abstraction consists of legacy code, while the highest level abstraction consists of elements of a software architecture. The reverse engineering process amounts to architecture recovery, while the forward engineering process amounts to architecture-based development.

For more than a decade, the requirements engineering community has proposed and studied goal models [22, 7, 3, 31] as high level abstractions for modeling early requirements. Goals capture stakeholder intentions. By modeling and analyzing them, we can derive functional and non-functional requirements in a systematic and coherent fashion. The goal models developed in this early phase of software development tell us not only the origins of functional and non-functional requirements, but also the space of alternative solutions (operationalizations) that the requirements engineer needs to select from. The KAOS methodology defines the state-of-the-art on this thread of research [7, 31, 32]. Hui et al [17] propose an extended framework for developing requirements that includes modeling and analysis of user goals, skills, and preferences (GSP). The framework is intended for the design of generic, customizable (*high variability*) software, to be used by a community of users. The original case for this work involved users with traumatic brain injuries in Oregon State [11].

We are interested in using the GSP framework to reengineer legacy software into generic, high variability software. To meet this objective, we are developing techniques for reverse engineering goal models from legacy software that offers some service (e.g., email). These models can then be revised, refined and extended, so that they can serve as basis for generating an extended version of the legacy software system that supports the same service in multiple ways. This paper presents the reverse engineering phase during which a goal model is extracted from legacy code. In the sequel, we adapt the horseshoe model as shown in

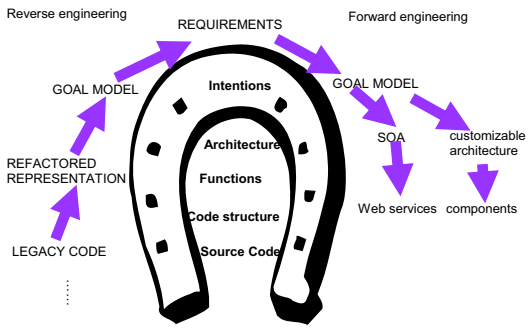

Figure 1. The horseshoe model

Figure 1. Instead of static architecture recovery, we aim

Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05)
0-7695-2425-7/05 $20.00 © 2005 IEEE

at discovering stakeholder goals from the behaviour of the system, thereby answering the most fundamental question about a software system: What is the system intended for?

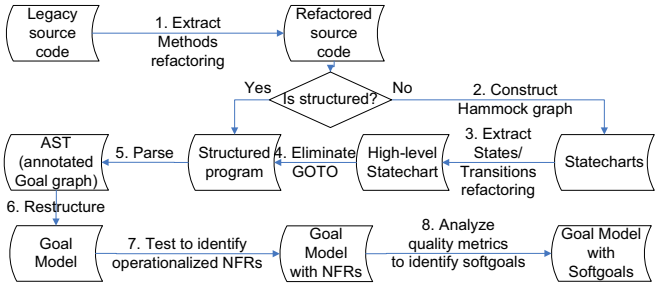

Figure 2. Major steps in our process

Our methodology converts structured and unstructured legacy source code into goal models following the steps illustrated in Figure 2: (1) refactor the source code by extracting methods [12] based on program slicing techniques [34] and programmer comments scattered in the code; if the refactored code is structured, then go to step (5); otherwise (2) convert it into equivalent statecharts [14], (3) further refactor the statecharts into higher-level statecharts by extracting states and transitions, and (4) convert the high-level statecharts into an equivalent program which is then structured by eliminating GOTO's [37]; (5) parse the structured program into an abstract syntax tree (AST) representing an annotated goal graph; (6) restructure the annotated goal graph into stakeholder goals; (7) identify non-functional requirements (NFR) in the resulting goal model by testing its traceable code in order to (8) derive softgoals from the NFR, using heuristics such as quality metrics.

The rest of the paper is organized as follows. Section 2 presents relevant concepts and the proposed reverse engineering methodology. Section 3 explains the refactoring of source code into its abstract form; Section 4 explains the extraction of goal models from the refactored programs. Both sections conduct a case study of open-source software (SquirrelMail [1] and Columba [9]), showing the independence of programming languages and structure of the legacy code. Section 5 discusses tool support that reduces the amount of time required for reverse engineering. Section 6 compares our proposal with related work, while section 7 summarizes results and sketches research directions.

2 Concepts and approach

Our proposal is based on well-founded concepts in software engineering. In particular, goal models [3, 31] arise during early requirements elicitation; program slices [34] are useful for program understanding and static analysis; while statecharts constitute a powerful representation for dynamic systems [14]. These concepts are integrated into a methodology through which the gap between goal models and source code is bridged. We first introduce these concepts in more detail, before presenting our proposed methodology.

2.1 Goal models

A *goal model* is a graph structure representing stakeholder goals and their inter-dependencies. A goal can be decomposed into subgoals through AND/OR refinements. In [22], softgoals are proposed as means for modeling and analyzing qualitative objectives. Unlike their vanilla cousins, softgoals (such as "improve profits" or "keep customers happy"), have no objective criterion as to whether they are satisfied or not. Goals and softgoals can be related to each other through MAKES, HELPS, HURTS and BREAKS relations. With goal models, requirements analysis proceeds by refining goals, identifying collections of leaf goals that together fulfill root-level goals, and assigning responsibilities for the fulfillment of leaf-level goals to either the system-to-be or external actors.

2.2 Statecharts

Statecharts constitute a concise visual formalism that captures the dynamic behaviour of a system [14]. Statecharts have been adopted in UML as one of the diagrammatic notations for modeling behaviour. Statecharts extend conventional finite state machine diagrams in several ways: a collection of sub-states can be abstracted into a super-state through AND or XOR composition; the number of states is visually reduced by zooming sub-states out; a transition from/to a super-state can abstract a number of transitions from/to its sub-states; parallel AND sub-states also reduce the number of combined states. Low-level statecharts capture both control and data dependencies of a program, and are close to implementation; high-level statecharts, on the other hand, hide implementation detail and abstract system behaviour. Although UML tools such as STATEMATE [15] can generate executable code from given statecharts, it is not yet possible to convert source code into statecharts. Our approach relies on program slicing and software refactoring techniques to do just that.

2.3 Program slicing

A *control flow graph* (CFG) is a directed graph of statement nodes and control transition edges. A *Hammock graph* (HG) is a subgraph of the CFG of the program that has a *single entry* n_e and a *single exit* n_x, i.e. all paths from a node in the CFG outside HG to a node inside HG must go

through n_e; all paths from a node inside HG to a node of the CFG outside the HG must go through n_x. The concept of Hammock graph serves two purposes: (1) unstructured programs can be structured using Hammock graphs [34, 37]; (2) a statechart can be constructed to associate a precondition entry substate to n_e and a postcondition exit substate to n_x. *Program slicing* [34] generates a slice P' of a program P based on a *slicing criterion* $< p, V >$ where p is a statement of P and V is a subset of the variables of P [30]. Static program slicing finds the statements that are either (control) dependent on p or (data) dependent on V. *Program dependence graph* (PDG) [25] and inter-procedural *system dependence graph* (SDG) [16] can be seen as results of program slicing, combining both control and data dependence information in a program [10]. In this paper, program slicing technique is used to separate the *live* variables needed by the rest of program execution on one Hammock graph.

2.4 Our approach

Our approach summarized in Figure 2 is inspired by software refactoring techniques [20]. *Refactoring* has been proposed [24] as a method for understanding and maintaining complex source code: it restructures and simplifies source code by *improving its internal structure without changing its external behaviour* [12]. In our proposal, source code is converted into more abstract form by recursively applying the refactoring operation *Extract Method* [12]. The scope of this process can be determined by delimiting comments, as these comments often indicate a semantic gap for program understanding [12]. The name of the unit method is either generated through heuristics or supplied by the software engineer. For example, the first five non-stop words can be concatenated as the identifier. Since a meaningful method name is helpful in understanding the program, the original comment is kept and moved into the refactored method.

For programs with too many comments, we deal with excessively fine-grain states and transitions to achieve higher levels of abstraction: the resulting code is subjected to new rounds of refactoring with a scope determined by Hammock graphs: we extend the *Extract Method* on an equivalent statechart representation of the program. As with *Extract Method*, *Extract States* replaces a sequence of states with a new super-state and *Extract Transitions* replaces a sequence of transitions with a new transition to the final state. For programs with too few comments, we get very high level statecharts and goal models. To understand the refinements of goals, we may consider other delimiters such as nested Hammock graphs (basic blocks). After these refactoring steps, the more abstract statecharts can be converted into an equivalent abstract program further structured through a GOTO elimination algorithm.

Goal models are then automatically constructed based on the resulting program which is by now both structured and abstract. An annotated goal graph is created from the program AST, and an AND/OR goal model is constructed from the annotated goal graph. Using the traceability between code and goal model, we identify non-functional requirements through function tests. By observing the effects on quality metrics through enabling/disabling the identified NFRs, we derive quality softgoals and create proper contribution links from the NFRs to them. The derived softgoals help bridge the gap between the actual implementation (source code) and its early requirements.

3 Refactoring for code abstraction

The proposed approach is illustrated with two open-source legacy software systems. The first system, SquirrelMail 1.5.0 [1], is an unstructured Web-based email client implemented in PHP. The second system, Columba 1.0 RC2 [9], is a structured email client implemented in Java. Our approach refactors goal models from both systems, despite the use of different programming languages and environments.

3.1 Extract Method using comments

We use *Extract Method* [12] as a refactoring technique to simplify the legacy code. *Extract Method* has the advantage that it is applicable to both object-oriented and procedural code. As illustrated in [12], *Extract Method* deals with statement blocks. Each block is determined by delimiting comments to reveal the programmer intentions. An implicit requirement for *Extract Method* is that the block must be a Hammock graph, have single entry and single exit [37]. Without loss of generality, consider just two statements $S_1(I_1, O_1)$ and $S_2(I_2, O_2)$ where I_1, I_2 are the sets of input variables and O_1, O_2 are the sets of output variables for the respectively numbered statements S_1 and S_2. Note that these statements may also have resulted from a previous application of *Extract Method*. If O has more than one live variable, then program slicing is conducted to separate each live variable slice into a separate method.

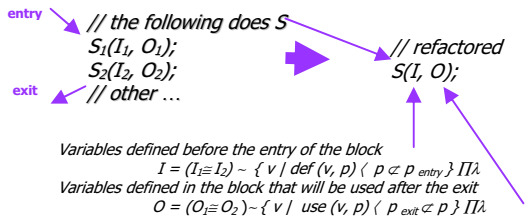

Figure 3. Illustration of *Extract Method*

The refactored statement $S(I, O)$ is obtained by program slicing. Specifically, as shown in Figure 3, S is the new

method named after the comment; I is computed as the set of input variables on entry of the block; O is computed as the set of output variables on exit of the block. We simplify the resulting statement by excluding internal variables of the block that are not used elsewhere in the program (thereby reducing the complexity of the code representation). The *Extract Method* refactoring step can be applied several times until all the commented blocks are replaced.

3.1.1 Refactoring SquirrelMail

SquirrelMail consists of 69771 LOC written in PHP, including 32249 (46.2%) lines of comments. The source code includes: (1) 37 main PHP programs responsible for generating the Web pages are located in the `src` subdirectory; (2) 20 PHP routine files called by the `load_theme` function for customizing different look-and-feel themes. They are located in the `themes` subdirectory; (3) 15 PHP utility class files located in the `classes` subdirectory; and (4) 34 PHP utility function files located in the `functions` subdirectory. In the rest of the paper, we refer to a program file only by its name; for example, the file `src/login.php` is referred to as `login.php`.

Since a dynamic PHP program generates an HTML page with hyperlinks to other dynamic PHP pages, the call to the other PHP program can be delayed until the user clicks at the link. We treat hyperlinks as GOTO statements in an unstructured program. Hence, dynamically generated Web-based programs are considered unstructured, even though PHP is a structured programming language. Therefore, after refactoring SquirrelMail in step 1 in Figure 2, we need to structure it through steps 2 to 5.

For example, the following Hammock graph in `login.php` has a variable definition `SM_PATH` exported to the remaining program, while no variable is imported.

```
/** Path for SquirrelMail required files. */
define('SM_PATH','../');
require_once($SM_PATH . 'functions/strings.php');
require_once($SM_PATH . 'config/config.php');
require_once($SM_PATH . 'functions/i18n.php');
require_once($SM_PATH . 'functions/plugin.php');
...
```

Hence `$SM_PATH` constitutes the only output variable of the block, whereas the set of input variables is empty. Note that even global variables that are not used outside the block will be hidden during the abstraction. After slicing analysis, it is safe to declare a new function `set_path` and call it in `login.php` as follows.

```
$SM_PATH=set_path ();
```

A complete listing of login.php has 185 LOC. The following lists `login.php` after *Extract Method* refactoring where all the comments from the original program have been removed. Several blocks have been replaced with method calls.

```
<?php /* login.php */
$SM_PATH=set_path ();
$SM_lang=setup_language();
$base_uri  = findout_base_URI();
$logindisabled = detect_imap_server($base_uri);
if ($logindisabled) { explain_situation(); exit;}
do_hook(login_cookie');
$header =onload_function('redirect.php');
display_header($header);
load_theme($theme[$theme_default]);
do_hook('login_top');
show_logo(); show_form($loginname, $mailto, $key);
do_hook('login_form'); do_hook('login_bottom');
?>
```

Similarly, such *Extract Method* refactoring delimited by comments can be applied to the other PHP files.

3.1.2 Refactoring Columba

Columba [9] is an open-source Email client that has 144916 LOC in Java, including 31472 (21.7%) lines of comments. The program is structured.

The input to our method is the main class of Columba: `org.columba.core.main.Main.run()`. The lengthy routine has 81 lines of code. Inside the routine, there are 22 code segments separated by 18 comments and 3 hammock boundaries. The first 15 lines are shown below.

```
class Main {
 public void run(String args[]) {
 1 ColumbaLogger.createDefaultHandler();
 2 registerCommandLineArguments();
 3 // handle commandline parameters
 4 if (handleCoreCommandLineParameters(args))
 5 {  System.exit(0); }
 6 // prompt user for profile
 7 Profile profile=
 8    ProfileManager.getInstance().getProfile(path);
 9 // initialize configuration with selected profile
10 new Config(profile.getLocation());
11 // if user doesn't overwrite logger settings ...
12 ColumbaLogger.createDefaultHandler();
13 ColumbaLogger.createDefaultFileHandler();
14 for ( int i=0; i<args.length; i++)
15 { LOG.info("arg["+i+"]="+args[i]); } ...
```

The Eclipse refactoring tool was used to extract 22 methods from the code.

```
public void run(String args[]) {
 1 ColumbaLogger.createDefaultHandler();
 2 registerCommandLineArguments();
 3 handler.registerCommandLineArguments();
 4 handle_commandline_parameters(args);
 5 Profile profile = prompt_user_for_profile();
 6 initialize_configuration_with_selected_profile(profile);
 7 initialize_default_logging(args);
 8 SessionController.passToRunningSessionAndExit(args);
 9 enable_debugging_repaint_manager ();
10 StartUpFrame frame = show_splash_screen();
11 register_protocol_handler();
12 load_user_customized_language_pack();
13 initialize_plugins(handler);
```

```
14 load_plugins();
15 set_look_and_feel();
16 init_font_configurations();
17 set_application_wide_font();
18 hide_splash_screen(frame);
19 handle_commandline_arguments_in_modules(handler);
20 restore_frames_of_last_session();
21 ensure_native_libraries_initialized();
22 post_startup_of_the_modules(handler);
}
```

3.2 Extracting states and transitions using hammock graphs

After the application of the *Extract Method* refactoring delimited by programmer's comments, the resulting code has no comments for further use. Moreover, due to possible lack of comments, the refactored code may still be at a lower level of abstraction than what we need to start deriving goals.

In SquirrelMail there are too many comments scattered in the routines, therefore *Extract Method* refactoring results in methods with too many short methods. To make it worse, the unstructureness of a Web-based system, such as SquirrelMail, limits the extraction of methods to individual routines. Thus, in order to obtain a more abstract representation, a behavioral view of the whole system needs to be extracted. In contrast, the result of *Extract Method* refactoring on the Columba system is less complex because Columba is structured and has well-written comments.

Static program analysis techniques can help us achieve more abstract program descriptions. In this section, we explain the use of Hammock graphs and statecharts to obtain an abstract view of system behavior.

Each extracted method has a single entry and a single exit (*Hammock graph*). At the entry and exit of the Hammock graph, pre- and post-conditions define allowable classes of input/output states. The transition between them is effected by the method. The states and transitions derived from a Hammock graph form a statechart. The statecharts of all hammock graphs are combined into a complete statechart by adding transitions according to the control flow.

For example, in Figure 4, we adopt the statechart notation used in [27]. The action at the transition `set_path` defines a variable `$SM_PATH`. Before the action `set_path`, the variable `$SM_PATH` is undefined. We model undefined variables as initial states. An event can also be put to the left of the slash in the transition label, to specify the triggering condition for the transition. Accordingly, we convert the refactored `login.php` code into an initial statechart (Figure 5). Note that two special functions `do_hook` and `load_theme` can make calls to other methods dynamically. Apart from the static calls, plugin routines registered for a hook name are called dynamically through `do_hook`. If there are no registered plugins for a hook name, then the action will be a NOP (nil operation). Similarly, the `theme` routines are called by a `load_theme` function based on a configuration parameter variable `$theme_default`. Note that in this statechart there are two exits, each leading to a different final state.

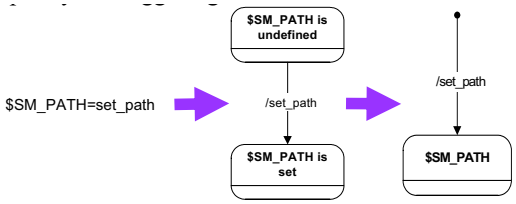

Figure 4. Statechart notations

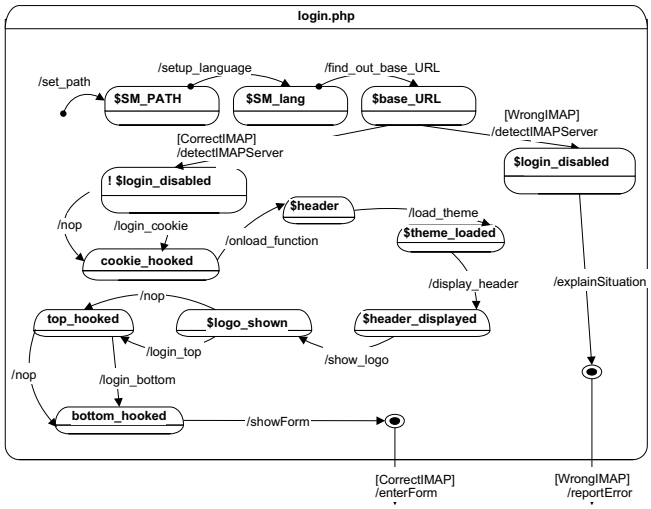

Figure 5. A statechart converted from refactored code

A statechart constructed from refactored code generally has too many states and transitions, and is hard to understand. We therefore need techniques to group states and transitions into more abstract, and fewer, super-states and super-transitions.

We accomplish this first by introducing layers: a group of states with single entry and exit are grouped together into one super-state. The new super-state replaces the group of the original states comprising it, thereby reducing the number of states. This refactoring step is called *Extract States*. In Figure 6, we illustrate how three states in the `login.php` (see Figure 5 and Figure 6a), namely `$SM_PATH`, `$SM_lang` and `$base_URL`, are grouped into a super-state named as `Globals` (Figure 6b). In Figure 7a, the sequence of three transitions in `Globals` statechart in Figure 6b are refactored into a single transition

(`set_globals`). To simplify the statecharts, a super-state with a single transition inside can be replaced with a new state by merging empty incoming/outgoing transitions to/from the superstate with its internal transition. This refactoring step is called *Extract Transition*. For example, in Figure 7a, `Globals` has an empty incoming transition; this transition is merged with `set_globals` which was inside `Globals` (Figure 7b). `Globals` is turned into a new state `globalsSet` with non-empty outgoing transitions. The result of applying *Extract States and Transitions* on `login.php` is shown in Figure 8. The abstracter view in Figure 9 puts together all top-level statecharts.

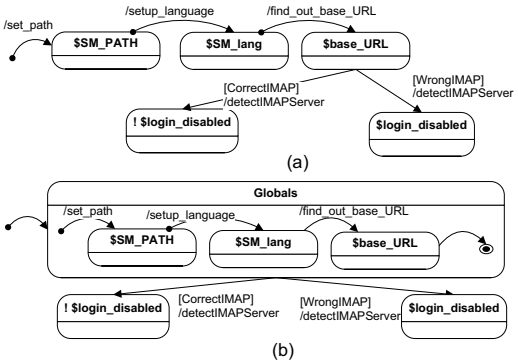

Figure 6. Extract States Refactoring part of the statechart in Figure 5 (a) into (b)

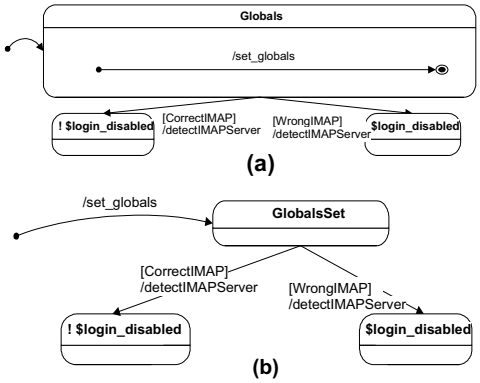

Figure 7. Extract Transitions Refactoring on the new super-state Globals in Figure 6b

3.3 Structuring the statecharts

The combined statecharts obtained from PHP programs in the previous steps are unstructured, even though a PHP program has no explicit GOTO statements. On the other hand, a goal model is formed by structured AND/OR decompositions. Before recovering a structured goal model (see Section 4), the statechart needs to be structured to contain only sequences, branches and loops.

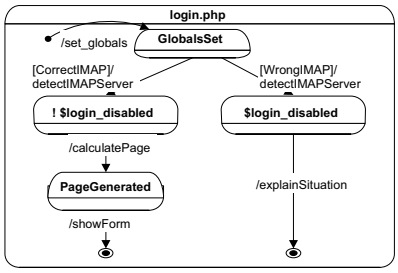

Figure 8. Refactoring on the initial statechart in Figure 7

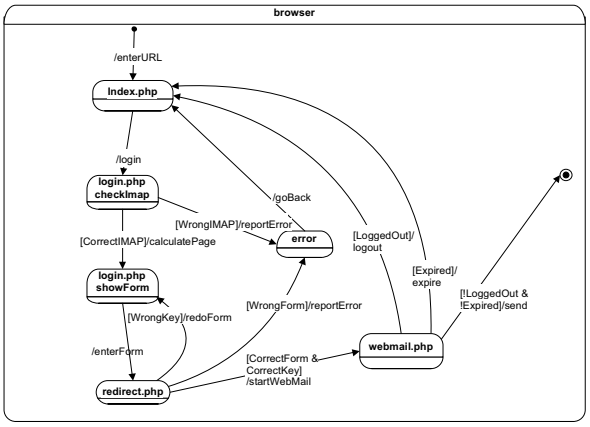

Figure 9. Top-level statechart of the browser

First, the high-level statechart (Figure 9) can be mapped into a program with GOTO statements. For convenience, in the sequel we use FORTRAN for such programs. Each state with more than one entry will be associated with a label, and each transition for an additional exit is associated with a GOTO statement following its activity statement. Secondly, to obtain a structured program, we adopt the GOTO statements elimination implemented in the FPT compiler [37]. It has been established using the theorem prover PVS [26, 28] that all the GOTO's can be removed through semantic preserving transformations, resulting in structured Hammock graphs [37]. As the technique eliminates GOTO's through hammock graph construction, it can be directly combined with our *Extract States and Transition* refactoring. For example, the statechart in Figure 9 can be converted into a FORTRAN program with GOTO statements (Figure 10a); the program is then structured (Figure 10b) using FPT [8].

```
      call EnterURL
10    call Login
    if (wrongIMAP) goto 30
20    call ShowForm
      if (wrongKey) goto 20
      call EnterForm
      if (wrongForm) goto 30
    call StartWebMail
    if (loggedOut) goto 10
    if (expired) goto 10
    call Send
    Stop
30  call ReportError
      call GoBack
    goto 10
    end
```

(a)

```
CALL EnterURL
REPEAT
 REPEAT
  CALL Login
  IF (.not.wrongIMAP) THEN
    REPEAT
      CALL ShowForm
    UNTIL (.not.wrongKey)
    CALL EnterForm
    IF(.not.wrongForm)THEN
      CALL StartWebmail
    ENDIF
  ENDIF
UNTIL (.not.loggedOut.or
 .not.expired.or.wrongIMAP
 .or.wrongForm)
IF(wrongIMAP.or.wrongForm)
THEN
  CALL ReportError
  CALL GoBack
ENDIF
UNTIL (.not.wrongIMAP.and.
          not.wrongForm)
CALL Send
END
```

(b)

Figure 10. Structuring the code converted from the statechart in Figure 9

4 Extracting goal models from abstract code

This section explains steps 5 to 8 in Figure 2. In the NFR framework [3], a goal has an intended function (intention), and an associated topic (subject matter). In a statechart, the function is found as the action of a state transition and the topic as the contextual state of the statechart. There are two basic modalities for goals: achieve or maintain [31]. In our process, it is easier to identify an *achieve* goal as a transition between different states whereas a *maintain* goal as a transition from a state to itself.

As shown in Figure 11, (a) a chain of state transitions designates an AND decomposition of a goal; the transitions correspond to a sequential composite statement. Here an ellipse denotes an unnamed goal. Furthermore, the parallel join of transitions from/to other states (b) designates OR-decompositions of a goal, corresponding to a branch statement e.g., IF-THEN-ELSE, with a condition derived from the event label on the transitions. Loops can also to be mapped to the goal model where the stop event s is converted into an intermediate goal along with the actions (c). This case results in a cyclic goal model, where the switching events on the transitions correspond to an OR decomposition. Having a structured statechart/program, we can view its abstract syntax tree (AST) as a goal model annotated with the control conditions, such as `IF(x)`, `REPEAT..UNTIL(s)`, etc.

4.1 Extracting goal models from SquirrelMail

The structured program in Figure 10b can be converted into an annotated goal model in Figure 12.

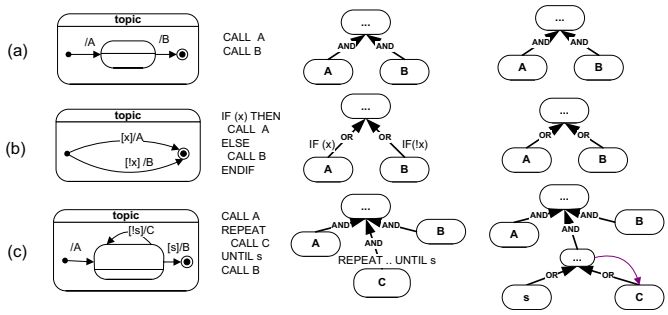

Figure 11. Patterns to extract goal models

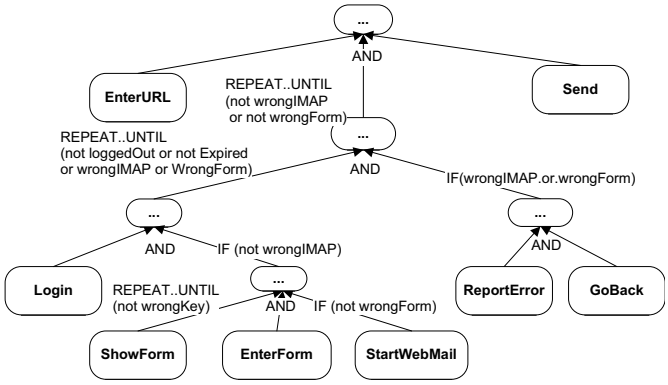

Figure 12. View AST as annotated goal model

Using the basic conversions (Figure 11) on the annotated control patterns (Figure 12), all the transitions are converted into goals in an AND/OR graph (Figure 13). Moreover, some tasks in the goal model are operationalizations of non-functional softgoals as they contribute to quality concerns modelled by the softgoals. For example, "Login" is a task contributing to the *security* concern. "ReportError" is another one that contributing to the *usability* concern (see Figure 13).

4.2 Extracting goal models from Columba

The AST of the refactored high-level Columba code generates 22 leaf-level goals in the annotated goal model. After applying the three transformation patterns (Figure 11) on the annotated goal graph, we obtain an AND/OR goal model that contains 22 leaf-goals. Among them, 13 goals were operationalizations of NFRs. These goals can be enabled by satisfying guard conditions in the `IF` statements, which are based on the quality metrics of 9 softgoals, including usability, maintainability, extensibility, etc. One can further categorize them into decomposing hierarchies.

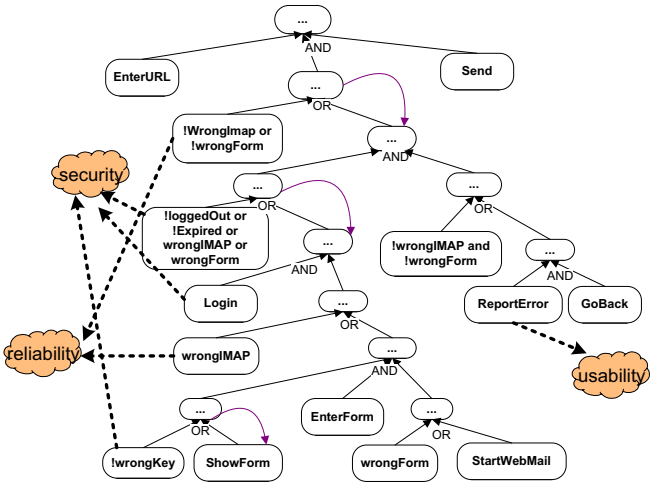

Figure 13. The *SquirrelMail* goal model

A complete description of these steps and verification results are discussed in [36]. We show the refactored code corresponding to the extracted goal model in Figure 14, where the NFR-operationalizing goals are separated into decomposition hierarchy for softgoals.

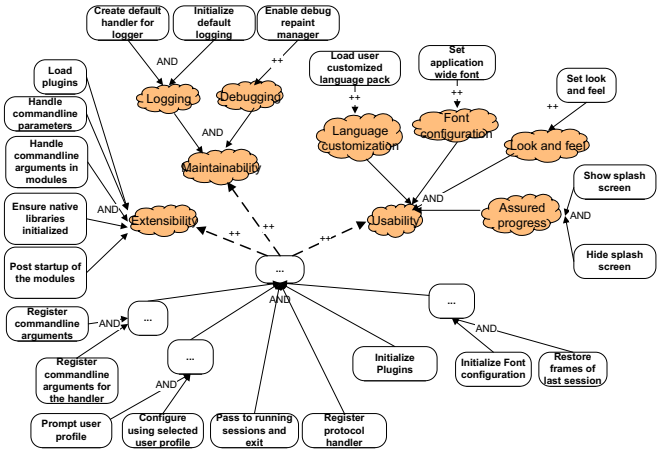

Figure 14. The *Columba* goal model

```
public void run(String args[]) {
 if ( maintainability_logging)
     ColumbaLogger.createDefaultHandler();
 registerCommandLineArguments();
 ComponentPluginHandler handler = register_plugins();
 handler.registerCommandLineArguments();
 if ( extensibility) handle_commandline_parameters(args);
 Profile profile = prompt_user_for_profile();
 initialize_configuration_with_selected_profile(profile);
 if ( maintainability_logging)
     initialize_default_logging(args);
 SessionController.passToRunningSessionAndExit(args);
 if ( maintainability_debugging)
     enable_debugging_repaint_manager ();
 StartUpFrame frame = null;
 if ( usability_assured_progress)
     frame = show_splash_screen();
 register_protocol_handler();
 if( usability_language_customization)
     load_user_customized_language_pack();
 initialize_plugins(handler);
 if ( extensibility) load_plugins();
 if ( usability_look_and_feel) set_look_and_feel();
 init_font_configurations();
 if ( usability_font_configuration)
     set_application_wide_font();
 if ( usability_assured_progress)
     hide_splash_screen(frame);
 if ( extensibility)
     handle_commandline_arguments_in_modules(handler);
 restore_frames_of_last_session();
 if ( extensibility)
     ensure_native_libraries_initialized();
 if ( extensibility)
     post_startup_of_the_modules(handler);
}
```

These softgoals fill in a gap between the code and its early requirements.

5 Discussion

In this section, we outline the implemented tool support for the reverse engineering process.

First, it is important to check the correctness of the refactoring steps to ensure that semantics is indeed preserved. A standard way for accomplishing this [12] is to test each refactoring step using available test cases. However, test cases can identify behavior changes, but can't ensure correctness. In order to prove that each step is semantics-preserving, we use program slicing techniques to ensure that both *Extract Method* and *Extract States and Transitions* are properly used. Also, the structuring of the statecharts is based on a well-established theory for GOTO eliminations, where the basic transformations have been proven correct. We can also annotate the state transitions in the statechart with the corresponding code. Therefore, the very detailed statechart is exemplified by the program code. Moreover, such traceability allows a change in the high-level abstraction to be easily reflected in the change of the code.

We can significantly improve the usefulness of our method by automating parts of it as follows:

- *Refactoring based on program slicing*. We can use the Extract Method as it is available for Java through the Eclipse IDE [35] (`Shift_Alt_M`). This refactoring is done semi-automatically by selecting statements delimited by programmer comments. Eclipse can only extract one method from a Hammock graph that has only one live variable. When there are multiple live variables, the Hammock graphs should be partitioned

into multiple disjoint program slices. We are developing an *Extract Method* tool for PHP and we are looking for a suitable case study in JSP that allows us to reuse existing Java refactoring tool support in Eclipse. Developing a tool for the *Extract States and Transitions* refactoring is also in our tool implementation agenda.

- *Statechart structuring*. Currently we deal with the problem by converting statecharts into an equivalent Fortran code with GOTO statements to leverage an existing Fortran compiler that has implemented the GOTO elimination algorithm [8, 37].

- *Extracting goal model*. The AST of the structured program is used to generate the annotated goal models. The format of these generated goal models conforms to the OMG XMI standard, which is exchangeable with other modeling tools such as EclipseUML or RationalRose. To this end, we used Eclipse modeling framework (EMF). Then we used the JDT API in Eclipse to convert any structured Java program into an annotated goal model, and further created an AND-OR goal model using the basic patterns. The annotation labels are automatically transformed into a purely AND-OR goal model using the basic patterns in Figure 11.

- *Identifying NFR-operationalizing goals by testing*. First, a set of functional test cases is prepared. Then every statement is guarded by an IF statement on a boolean condition. Falsifying the condition will skip the statement: if the modified execution still passes the functional test, then its corresponding goal is addressing NFRs. The necessary statements for the functional tests are restored to the unguarded form.

- *Linking NFR-operationalizing goals to softgoals by testing*. Once non-functional tasks are identified, one can use non-functional requirements test cases to associate the guard conditions to the NFR name and resort to the NFR framework [3] to categorize them with certain quality attributes, as softgoals in the extended goal model. The quality attributes answer why the non-functional tasks are present in the source code.

6 Related work

Initially, goal models [7] were proposed to capture requirements, i.e., the optative statements of the system-to-be [32]. Goal models have been extended to represent both functional and non-functional requirements [22]. [6] has developed a set of requirements eliciting tools to bridge the gap between NFRs and UML diagrams, where Class, Sequence and Collaboration diagrams are considered. According to the horseshoe model, this corresponds to the forward engineering phase. In this paper, we consider statecharts [14] as a suitable intermediate representation for the dynamic behavior of legacy code, as well as the abstract interface to the environment.

This paper is not the first attempt to discover goal models from sources other than requirements. In the KAOS project, goal models can be inferred from user scenarios [32]. However, scenarios generally do not cover all possible paths of program executions. And legacy software often comes with incomplete and inaccurate documents. Therefore, our methodology complements the KAOS approach, based on the idea of understanding-by-refactoring. The recovered goal model is not guaranteed to capture the intentions of the original stakeholders, but can be trusted to capture the implemented intentions of stakeholders, as understood by programmers. It is also more traceable from the code since each refactoring step is documented and is also invertible.

Goal models can also be seen as abstractions of system processes. Other literature details techniques for recovering process models from events collected during the software development process [4]. However, this work focuses on inferring the processes used to develop software, rather than the processes realized by the software itself. Program model checking [33] systems, such as Bandera [5], extract finite-state machines from Java source code. Although such systems have succeeded in finding counter examples for some programs, the combinatorial explosion of states ultimately limits their applicability in revealing intentions behind a large software system. Not surprisingly, we found a similar combinational barrier when stakeholder goal models are converted into state machines for model checking [13]. According to our case study, goal models can be built more concisely from statecharts.

Using execution event traces, an algorithm was proposed in [29] to compress state diagrams into UML state diagrams, which is a variant of the statechart notation. The approach is complementary to our technique which does not rely on program inputs. Pattern-based design recovery [23] finds collaboration diagrams and statecharts from source code, which is also similar to our work, but it relies on pattern matching rather than legacy code comments.

7 Conclusions and future work

We have proposed a framework for reverse engineering legacy code in order to discover the stakeholder goals it was intended to fulfill. Our proposal has been illustrated with two case studies involving public-domain legacy email systems (SquirrelMail and Columba). The case studies suggest that the process of recovering stakeholder goals can be systematized. Moreover, reverse engineered goal models are traceable in the code, making it feasible to forward engineering goal models into new architectures.

The proposed framework definitely has limitations and needs further research. Specifically, the proposed methods won't work well in the absence of well-thought out comments. Moreover, further experimentation is needed to evaluate the effectiveness of heuristics used throughout the proposed reverse engineering process. In future work, we propose to study methods for the recovery of softgoals using hints from architecture and design documents and also to compare the reverse engineered goal models with those derived through requirements elicitation.

References

[1] R. Castello. SquirrelMail, http://www.squirrelmail.org.

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

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

[4] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[5] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zhang. Bandera: Extracting finite-state models from Java source code. In *ICSE00*, pages 439–448.

[6] L. M. Cysneiros and J. C. S. P. Leite. Non-functional requirements: from elicitation to conceptual models. *IEEE Trans. on Softw. Eng.*, 30(5):328–350, May 2004.

[7] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, Apr. 1993.

[8] E. H. D'Hollander, F. Zhang, and Q. Wang. The Fortran parallel transformer and its programming environment. *Journal of Information Sciences*, (106):293–317, 1998.

[9] F. Dietz and T. Stich. The Columba project, http://columba.sourceforge.net.

[10] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program Languages and Systems*, 9(3):319–349, 1987.

[11] S. Fickas, L. Ehlhardt, M. Sohlberg, and B. Todis. Towards personal RE: A challenging case study, 45-02. Technical report, Computer Science Department, University of Oregon.

[12] M. Fowler. *Refactoring: Improve the design of existing code*. Addison-Wesley, Reading MA, 1997.

[13] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. In *RE'03*, pages 105–114, 2003.

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.

[15] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, Oct. 1996.

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. On Programming Languages and Systems*, 12(1):26–60, 1990.

[17] B. Hui, S. Liaskos, and J. Mylopoulos. Goal skills and preference framework. In *RE'03*, pages 117–126.

[18] R. Kazman, S. G. Woods, and S. J. Carriere. Requirements for integrating software architecture and reengineering models: CORUM II. In *WCRE'98*, pages 154–163, 1998.

[19] J. Leite. Working results on software re-engineering. *ACM SIGSOFT Software Engineering Notes*, 21(2):39–44, 1996.

[20] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Trans. Software Engineering*, 30(2):126–139, 2004.

[21] H. Muller, J. Jahnke, D. Smith, M. Storey, S. Tilley, and K. Wong. Reverse engineering: A roadmap. In *Future of Software Engineering, ICSE'00*, pages 49–60, 2000.

[22] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, Jun 1992.

[23] J. Niere, J. P. Wadsack, and A. Zundorf. Recovering UML diagrams from Java code using patterns. In *SCASE'01*, 2001.

[24] W. Opdyke. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, 1992.

[25] K. Otteinstein and L. Ottenstain. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177–184, May 1984.

[26] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. Software Eng.*, 21(2):107–125, Feb. 1995.

[27] M. Samek. *Practical statecharts in C/C++*. Quantum programming for embedded systems. CMP books, 2002.

[28] N. Shankar. Steps towards mechanizing program transformations using PVS. *Science of Computer Programming*, 26(1–3):33–57, May 1996.

[29] T. Systa, K. Koskimies, and E. Makinen. Automated compression of state machines using UML statechart diagram notation. *Information & Software Technology*, 44(10):565–578, 2002.

[30] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[31] A. van Lamsweerde. Goal-oriented requirements engineering: From system objectives to UML models to precise software specifications. In *ICSE 2003*, pages 744–745, 2003.

[32] A. van Lamsweerde and L. Willemet. Inferring declarative requirements from operational scenarios. *IEEE Trans. Software Engineering*, 24(12):1089–1114, Nov. 1998.

[33] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *ASE*, 10(2):203–232, 2002.

[34] M. Weiser. Program slicing. *IEEE Trans. Software Engineering*, 10(4):352–357, July 1984.

[35] www.eclipse.org. Eclipse IDE, Refactoring in JDT, EMF, UML2.

[36] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite. Refactoring source code into goal models, ftp.cs.toronto.edu/csrg-technical-reports/510. Technical report, University of Toronto, 2005.

[37] F. Zhang and E. H. D'Hollander. Using hammock graphs to structure programs. *IEEE Trans. Software Engineering*, 30(4):231–245, Apr. 2004.