Mining System-User Interaction Traces for Use Case Models

Mohammad El-Ramly, Eleni Stroulia and Paul Sorenson

Computing Science Department
University of Alberta
Edmonton AB, T6G 2H1, Canada
{mramly, stroulia, sorenson} @cs.ualberta.ca

Abstract

While code understanding is the primary program comprehension activity, it is quite challenging to recognize the application requirements from code, since they have usually been occluded by a set of layers of later implementation decisions. An alternative source of evidence, especially valuable for understanding the purposes for which the application was built, can be the dynamic behavior of the system, and more specifically the system-user interaction. We have developed a method for modeling the application behavior from the user's perspective in the form of use case models, using recorded traces of system-user interaction. We use data mining and pattern matching methods to mine these traces for frequently occurring user tasks. When interesting patterns are discovered, they are augmented with semantic information and they are used to build use case models. We demonstrate a successful application of this method to recover use case models from interaction traces with legacy 3270 systems to serve user interface reengineering activities.

1. Introduction

Software development is rarely a "green-fields" process: more often than not, the new system under development has to be integrated with other existing systems, either newly-acquired off-the-shelf components (COTS) or legacy systems, whose functionality cannot easily be replaced. In the case of COTS, they usually come with documentation on how to integrate. But, to ensure that the desired features of a legacy system are properly integrated, it is imperative to understand how this system is actually currently used, that is, it is necessary to extract a model of its use cases.

The problem of understanding an application's use cases is, to some extent, akin to understanding the intent behind its development, and is, therefore, extremely challenging. Research in this field is still sparse, and has focused mainly on extracting use cases of object-oriented applications [3] by examining their code. But, the majority of legacy systems have been developed before the advent of the object-oriented design paradigm, in languages that do not provide strong encapsulation support. Even worse, the legacy application code is often unavailable, and even when it is available, it is usually a very poor expression of the application design. It is scarcely structured and usually includes "dead" or obsolete code and "glue" code of incremental updates that violate its original architecture; or "ignorant surgeries" as Parnas calls them [8]. It may even include obsolete and confusing comments that contradict the code.

More insight as to the purpose of the application from the user's perspective could be gained by inspecting how the application is actually used. The actual run-time behavior of the application could be an evidence of its use cases, as they are actually exercised by its current users. A major component of the run-time behavior of an application, is the system-user interaction. Recorded traces of this interaction can be an alternative or a complementary source of information for legacy system behavior modeling and use case extraction. Such use case models are important for legacy software understanding, maintenance, migration and reengineering activities.

System-user interaction is a rich source of knowledge and a faithful representation of how the system is currently being used. In [4],[5] we presented the CelLEST method for legacy user interface (UI) reengineering and integration. This method uses recorded traces of the interaction between the legacy system and its users as input, and does not require examination of the legacy code. The CelLEST project demonstrated that this easily available type of input can be sufficient for lightweight reverse engineering and user interface wrapping activities. Using the CelLEST method and tools, it is possible to build, semi-automatically, a model of the legacy text-based interface (TBI) as a state-transition graph. Then, given multiple examples of task-specific traces, an optimized task-centered graphical user interface (GUI) is constructed, that wraps the legacy UI and enables the performance of the same task from a browser.

Proceedings of the 10 th International Workshop on Program Comprehension (IWPC'02)
1092-8138/02 $17.00 © 2002 **IEEE**

In this paper we discuss how the CelLEST method can be further automated by discovering the use case models, needed for the GUI construction, from the interaction traces. We use knowledge discovery and pattern searching algorithms to search for frequently occurring patterns in the traces. These patterns are expected to correspond to the tasks of most interest to the legacy system user. An expert user can review them and accept them, reject them or request a modification to the discovery task parameters to look for more or different patterns. Then, the discovered patterns are annotated with the semantic information needed for the CelLEST GUI construction step, regarding the information exchange that occurs between the application and the user on each legacy system screen. Each legacy screen is a manifestation of an internal behavioral state of the legacy system that allows the user a specific set of inputs, offers him a set of outputs, and allows him to perform transitions to any of a limited set of other screens. An instance of a screen is called a "screen snapshot" or simply "snapshot".

The primary goal of this work is to further automate the CelLEST UI reengineering method and minimize the human effort needed in the process, i.e. eliminate the collection of task-specific traces. However, our long-term objective is to develop a general process for use case discovery from system-user interaction traces in support of tasks such as program comprehension, building UIs for new applications that are consistent with the user conceptual models, documenting interactive systems, and building help or user support systems [9].

After this introduction, section 2 briefly describes the CelLEST UI reengineering method. Section 3 discusses an example system-user interaction trace and Section 4 discusses an example use case model. Section 5 presents the use case pattern discovery process and algorithm. Section 6 reports the results of an experiment using this method. Finally, Section 7 provides a summary of our work and highlights future work directions.

2. The CelLEST Process for User Interface Reengineering

The CelLEST UI reengineering method is applicable when the reengineering objective is to provide a more usable GUI to current legacy services, or to make them available through the Web. Consider for example the problem of building an easy-to-use task-centered Web-interface for the reservation system of a hotel that runs on a mainframe, to allow reservations from the web, or integrating the front-ends of the claim systems of different insurance companies after their merger. The method assumes that it is not necessary to reengineer the underlying legacy application or modify its capabilities, or change the underlying platform on which it runs; hence, full-scale reverse engineering and code comprehension is not needed.

While the current industrial practices can accomplish the task above, they are mostly manual, time-consuming and potentially error prone. The CelLEST project aims to apply artificial intelligence methods to increase the automation of this process.

The CelLEST method consists of two phases. In its reverse engineering phase, the *LEgacy Navigation Domain Identifier* (LeNDI) prototype [4] is used to produce a behavioral model of the legacy system using traces of its current use. For legacy systems that use a block-mode data transfer protocol between the system and its user terminals, such as IBM 3270, a trace is a sequence of screen snapshots interleaved with the user actions performed in response to receiving the snapshots on his terminal. The behavioral model produced is represented as a state-transition graph. Each node of the graph corresponds to a distinct screen of the legacy interface. The graph nodes are identified by clustering similar screen snapshots together. To that end, we have developed a supervised clustering algorithm that clusters snapshots based on the similarity of the keywords they contain and their layout. Then a decision tree can be induced to classify new snapshots, that is, to recognize the interface screen of which they are an instance. Each edge of the graph corresponds to a possible user action, that is, a sequence of cursor movements and keystrokes on a particular screen that causes a transition to another screen.

After an expert user has reviewed and validated the legacy UI model, the process of analyzing task-specific navigation plans and constructing new interfaces to wrap these tasks can start. Each set of task-specific traces represents multiple executions of a single user task. The corresponding user task is modeled with the aid of some user input to identify the information exchange that takes place between the user and the legacy application during the task. The CelLEST test-bed for the forward engineering phase is a prototype tool called Mathaino [5]. Mathaino constructs an abstract specification of a GUI for the modeled task, which is subsequently "translated" to a platform-specific task-centered GUI implementation.

Figure 1 diagrammatically depicts the CelLEST approach to legacy UI migration. A task-centered GUI can combine the functionality of several legacy screens, related to one task in one GUI unit, e.g. a tabbed pane or a form. The new GUI drives the execution of user tasks through the legacy system using the state-transition graph and an API to the data transfer protocol used. This approach can be extended to integrate several legacy systems' front-ends in one GUI. Throughout the process, an expert user of the legacy system can review and inspect the constructed models through a visualization interface.

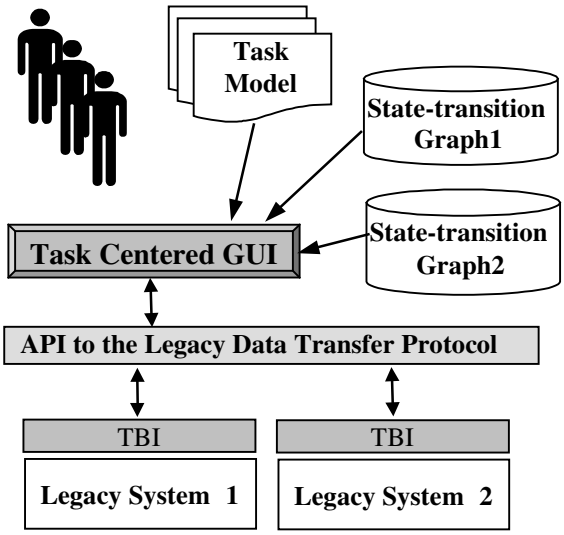


Figure 1: The CelLEST approach to legacy user interface migration.

The CelLEST UI reengineering method is powerful in that it constructs a high-level model of the interaction behavior between the legacy system and its users. Traditional UI reengineering approaches analyze the legacy code to detect GUI widgets or text-based interface code to replace them by functionally similar widgets or code in new platforms. Instead of replicating the same interaction with different widgets in new platforms, the CelLEST method encapsulates interesting behavioral segments of the legacy interface into new UI front-ends on different platforms with fundamentally different widget toolkits, such as WAP devices for example [5].

In this paper, we describe how patterns of frequent user tasks can be discovered automatically by applying data mining techniques to interaction traces, and hence eliminating the need to collect task-specific traces. A draft use case model is inferred for each pattern and then enriched with the semantic information required for the CelLEST forward engineering phase.

3. An Example System-User Interaction Trace

A trace of a user's interaction with the public online Library of Congress Information System (LOCIS) (IP: 140.147.254.3 or locis.loc.gov) through an IBM 3270 connection was recorded while a user was retrieving detailed information about some pieces of federal legislation. The user started by making the necessary menu selections to open the relevant library catalog. Then he repeated two information retrieval tasks for several times. Figure 2.a shows 20 consecutive screen snapshots of this trace with the keystrokes that occurred on each of them. The snapshots in solid-line represent a complete instance of one of the two information retrieval tasks. In this scenario, the user issued a *browse* command with some keyword(s) to browse the relevant part of the library catalog file. Then he issued a *retrieve* command to retrieve a subset of the catalog items. Finally the user displayed brief information about the items in this set using the *display* command and selected some items to display their full or partial information, e.g. the full legislation, its abstract, its list of sponsors, its official title, etc., by typing a *display item* command and then a display option. The actual LOCIS trace that was given as input to LeNDI was 454 snapshots long. LeNDI built a corresponding state-transition graph. Figure 2.b shows the part of this graph corresponding to the trace segment in Figure 2.a. The left top corner of every screen contains its ID, as given by LeNDI. Labels on the edges are the user action (command) models inferred.

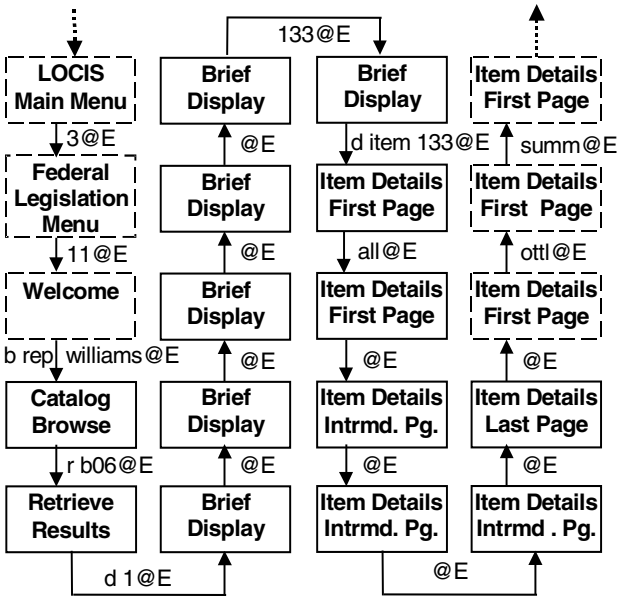
4. An Example Use Case Model

A use case describes a sequence of interactions (activities) between a system and an external "actor" that results in the actor accomplishing a task that provides benefit to someone. The actor can be the application user, another software application, a piece of hardware, or some other entity that interacts with the system to achieve some goal [10].

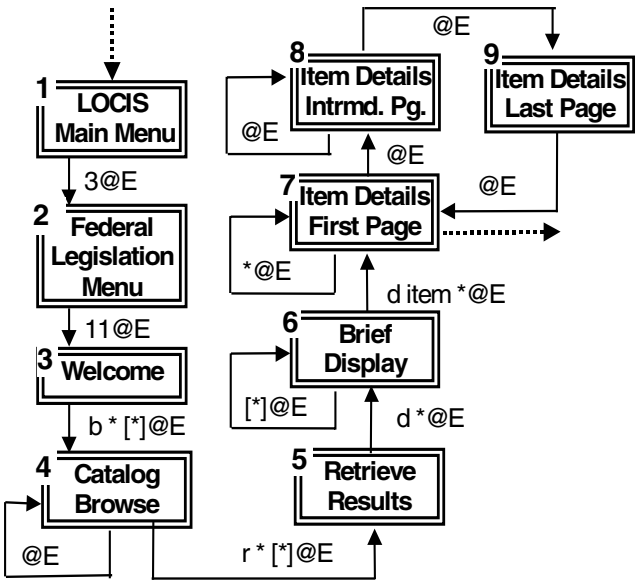
Provided that enough instances of the task shown in Figure 2.a appear in the traces recorded from the legacy application and that they meet some user-defined criterion for what constitutes "sufficiently interesting" patterns, our method can discover that these instances represent a candidate use case model. The pattern corresponding to the task instance shown in Figure 2.a and all similar ones is 4^{+}-5-6^{+}-7^{+}-8^{+}-9, where + is one or more. In section 6, we explain in details how this pattern was discovered from the input traces. After review by an expert, it was concluded that this pattern is a real use case.

The use case pattern is then enriched with the relevant action models and finally augmented with the semantic information necessary for the subsequent phase of the process. The Mathaino prototype in the forward engineering phase of CelLEST requires that the information exchange between the user and the application on each screen is specified, i.e. the user inputs to the system and the system outputs displayed to the user. A suitable standard notation can be used to represent the resulting use case model.

Figure 3 shows the enriched use case pattern for the task of Figure 2.a, a textual description of the corresponding use case model and a graphical representation of this model using activity diagrams, which are part of the UML toolkit [7].



(a) A part of the LOCIS trace used in the example.



(b) The corresponding portion of the state-transition graph. * means a mandatory argument, [*] is an optional argument and @E means Enter key.

Figure 2: An example trace of user interaction with the Library of Congress Information System.

5. Use Case Discovery

Sequential pattern mining is an important problem that has received a lot of attention by the knowledge discovery in databases (KDD) community. Many algorithms and tools have been developed, and it is now possible to mine massive amounts of sequential data to discover new, interesting and non trivial knowledge. For example, one can mine the Web access log files collected by Web servers for user access patterns, a process known as Web usage mining [6].

Mining legacy system-user interaction traces for recurrent patterns of user activity, corresponding to use cases, is similar to mining episodes in sequences. To tackle this problem, five issues need to be addressed:

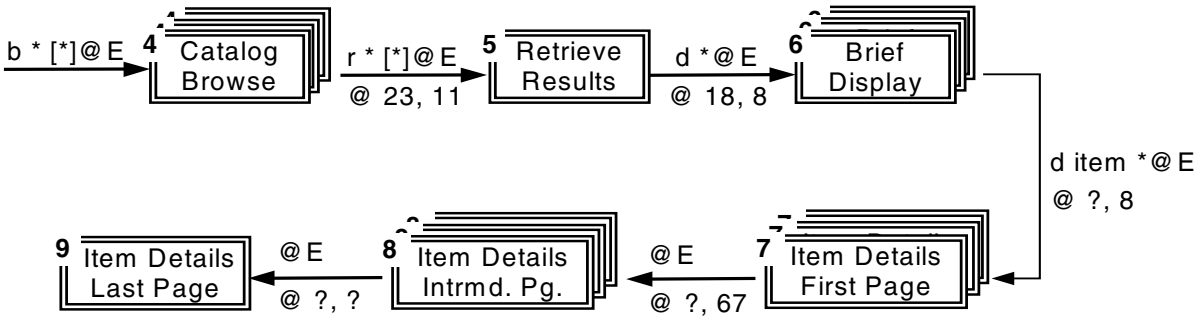
1. Constructing an appropriate representation syntax of the input traces: the syntax has to be compact, so as to enable the representation of long traces;
2. Defining a criterion for what constitutes sufficient evidence for a pattern in the traces: the criterion has to discriminate among spurious frequent occurrences and "real" tasks;
3. Extracting the patterns that fulfill the criterion;
4. Verifying or modifying the results of the extraction process through user feedback; and
5. Building use case models for the extracted patterns.

In the next few sections, we present how each of these five issues is addressed by our method.

5.1 Preprocessing

An interaction trace is initially represented as a sequence of snapshots, represented by integers. Each integer is the ID of the screen, of which the snapshot is an instance, according to the classifier constructed by LeNDI. Let's denote this representation as *R0*. *R0* often contains repetitions, resulting from accessing many instances of the same screen consecutively, such as for example, browsing many pages of a library catalog. These repetitions may result in missing some important patterns, if pattern discovery is performed on traces in *R0* format. For example the trace segment 4-5-6-6-6-6-6-6-7-7 in Figure 4 below will not be counted as an instance of the pattern 4-5-6-7 unless we allow an error of at least five insertions (irrelevant noise screens) in the instances of the patterns discovered.

To address this problem, traces are encoded using the run-length encoding algorithm, which replaces immediate repetitions with a count followed by the item being repeated. We denote this representation of the trace as *R1*. Figure 4 shows *R0* and *R1* representations of the trace segment of Figure 2.a.



(a) A use case pattern for an information retrieval task for a federal legislation from LOCIS, augmented with action locations. @ ?, 67 means that the user action occurs on the screen snapshot at unspecified row and column 67.

Use case name: Retrieving Information on a Federal Legislation

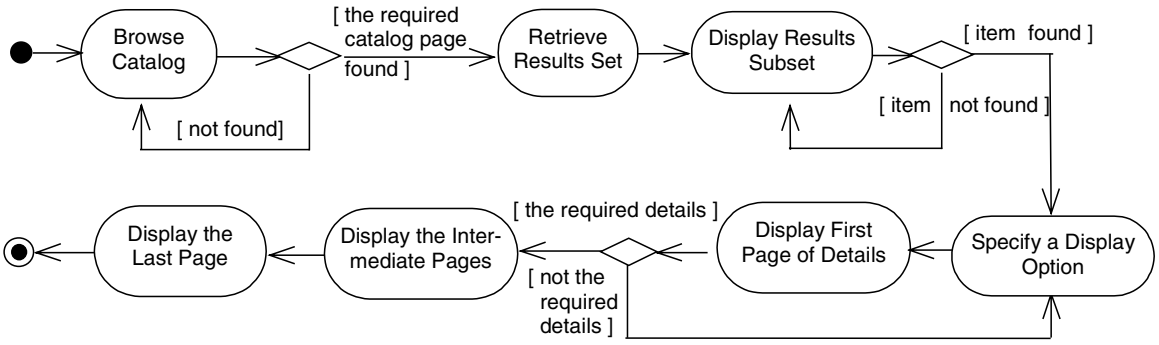
Participating actor: LOCIS User

Entry condition: The user issues a *browse* command to LOCIS

Flow of events:
1- Turn the catalog pages until the relevant page.
2- Issue a *retrieve* command to construct a results set for the chosen catalog entry.
3- Display the results set and turn its pages until the required item is found.
4- Issue a *display item* command.
5- Specify a display option.
6- Display the item details.
7- Repeat steps 5 and 6 till retrieving the right details

Exit condition: The user retrieves the required information about the federal legislation of interest.

(b) A textual description of the use case.



(c) The corresponding activity diagram.

Figure 3: An example use case pattern and the corresponding textual and UML models.

***R0* : 1-2-3-4-5-6-6-6-6-6-6-7-7-8-8-8-9-7-7-7**

***R1* : 1-2-3-4-5-**(6)**6-**(2)**7-**(3)**8-9-**(3)**7**

Figure 4: Preprocessing interaction traces.

5.2 Pattern Qualification Criterion

A common problem in KDD research is the discovery of a large number of patterns, many of which are simply spurious frequent occurrences and thus not interesting. This makes it difficult to comprehend them and distinguish from them the actual patterns of interest. So,

before pattern extraction, one needs to define the criterion of interesting patterns to focus the discovery process and reduce the retrieval of insignificant patterns.

We define this criterion in terms of four elements: the minimum pattern length (*minL*), the maximum pattern length (*maxL*), the minimum number of occurrences (also called frequency and support) of the pattern (*minFreq*) and the pattern *Score*. The default values for *minL*, *minFreq* and *Score* are 2, 2 and 1 respectively. If *maxL* is unspecified, then the algorithm discovers the longest possible patterns that meet the other conditions of the criterion. In our work, we adopted a simple scoring function for pattern evaluation aiming at balancing the weight of the pattern length with the weight of the pattern frequency. Our experiments showed that this function is adequate for our purposes. The function has two forms: one for exact patterns and another for approximate patterns with insertion errors. An exact pattern is one whose instances are all identical. An approximate pattern with insertion errors is one whose instances share an ordered sequence of IDs while each instance may contain up to a user defined number of extra screens.

1. For an exact pattern of length *pattL* and frequency *pattF*, the scoring function is:

   $Score = \log2 (pattL) * \log2 (pattF)$

2. For an approximate pattern, let the minimum length of any instance of this pattern be *pattL* and the pattern frequency be *pattF* and assume that the average number of insertion errors per instance is *avIns*. Then, the scoring function is:

   $Score = \log2 (pattL) * \log2 (pattF) * Density$

   $Density = pattL /(pattL + avIns)$

According to this function a pattern of length 4 and frequency 10 will have a *Score* equal to a pattern of length 10 and frequency 4, if they have the same density.

5.3 Pattern Extraction

We used a simplified variant of the Seq-R&G algorithm [2] to extract maximal exact use case patterns from interaction traces. A maximal pattern is a pattern that is not a sub-pattern of any other pattern with the same frequency.

To avoid an exhaustive search of all possible patterns of all lengths, the algorithm uses a popular idea in KDD literature. The idea is that a long pattern cannot meet the minimum frequency constraint unless its sub-patterns meet this constraint. To apply this, we can start by discovering patterns of length *minL* first using exhaustive search, then combining the ones that meet the *minFreq* constraint to construct patterns of length *minL*+1, thus avoiding an exhaustive search for the later. However, we still have to count the instances of each constructed *minL*+1 long pattern in the input traces to see which ones meet the *minFreq* constraint. Then, the qualified patterns of length *minL*+1 are used to construct patterns of length *minL*+2. This process is repeated until the qualified patterns of length *maxL* are discovered or until no more qualified patterns can be discovered.

Figure 5 shows the algorithm used. It starts by loading the desired traces from the database and asking the user for the parameters *minL*, *maxL*, *minFreq* and *Score* of the criterion for defining interesting patterns. Steps 4 and 5 create a candidate and a potential pattern lists. The first is used to store the patterns already discovered and the second is used to store the patterns currently under examination that do not meet the qualification criterion yet. Step 6 sets a window for scanning the traces, whose initial length is set to the defined minimum length, i.e. *winLen* = *minL*. Step 7 divides all the traces into segments of length *minL* by moving the window along each trace, sliding one item at time. For example if *winLen* = 3, the trace {3-4-5-6-3-6} will be divided into the subsequences {3-4-5}, {4-5-6}, {5-6-3} and {6-3-6}. Step 8 stores the segments that meet the criterion of interesting patterns in the candidate pattern list. Step 9 is repeated as long as there are more potential patterns whose lengths are less than *maxL*. In steps 9.a and 9.b, a new potential pattern list is constructed from the existing candidate patterns of length *winLen*. This is done by examining patterns in a pair-wise fashion. For every pair of patterns *i* and *j* of length *winLen*, if the suffix of pattern *i* (its last *winLen*–1 IDs) matches the prefix of pattern *j* (its first *winLen*–1 IDs), then a new potential pattern of length *winLen*+1 is formed. This is done by concatenating the first ID in pattern *i* to the beginning of pattern *j*. Step 9.c increments *winLen* by 1 for the next iteration. Step 9.d counts the number of occurrences of every new potential pattern in all traces under analysis. Step 9.e evaluates every potential pattern according to the user's criterion and removes unqualified patterns. Step 9.f breaks the loop if no new patterns were discovered in the current iteration. Step 9.g copies the qualified patterns to the candidate pattern list. When there is no more potential patterns, step 10 removes non-maximal patterns from the candidate pattern list. Finally, step 11 presents to the user all the extracted patterns ordered by length, frequency or score.

5.4 User Feedback and Post-processing

Once all the maximal patterns have been identified, one can change the pattern selection criterion to narrow or widen the results set if he feels that too little or too many patterns were retrieved. An implementation of the Shift-OR algorithm [1] is used to retrieve the instances of a discovered pattern to verify that they correspond fully or partially to a real use case.

1. Load N traces of length trL_i where $1 \le i \le N$ from the trace database
2. Read the four parameters *minL*, *maxL*, *minFreq* and *Score* from the user
3. Evaluation criterion *EvCriterion* = new Criterion (*minL*, *maxL*, *minFreq*, *Score*)
4. *candidatePatternList* = new PatternList ()
5. *potentialPatternList* = new PatternList ()
6. Initial window Length *winLen*= *minL*
7. For (i = 1; $i \le N$; i++)
   Break $trace_i$ into $trL_i - winLen + 1$ segments using a sliding window of length *winLen*
8. Store segments that meet *EvCriterion* in *candidatePatternList.*
9. while (*winLen* < *maxL*) {
   a. Empty *potentialPatternList*
   b. For every $pattern_i$ and $pattern_j \in$ *candidatePatternList* which are of length *winLen* {
      If $\text{suffix}_{winLen-1}$ ($pattern_i$) == $\text{prefix}_{winLen-1}$ ($pattern_j$) {
         Construct *newPattern* = first ID of $pattern_i$ + $pattern_j$
         Add *newPattern* to *potentialPatternList* }
      }
   c. *winLen* ++
   d. For (i = 1; $i \le N$; i++) {
      Break $trace_i$ into $trL_i - winLen + 1$ segments using a sliding window of length *winLen*
      For (j = 1; $j \le trL_i - winLen + 1$; j++)
         If $segment_j$ is in *potentialPatternList* increment its counter by 1
   e. Remove from *potentialPatternList* all segments that do not meet *EvCriterion*
   f. If *potentialPatternList* is empty, break the loop
   g. Copy all segments in *potentialPatternList* to *candidatePatternList*
   }
10. Remove non-maximal patterns from *candidatePatternList*
11. Present *candidatePatternList* to the user in the desired order.

Figure 5: The use case pattern extraction algorithm

Also, it is possible to search for instances of a discovered pattern with insertion errors, using a variant of the Shift-OR algorithm [11] for approximate pattern matching. These instances may include variations in the specifics of the task or in the path used to accomplish the task, e.g. opening a list of choices to choose from instead of typing a choice. These cases are alternative scenarios for the use case that also result in successful task completion.

Additionally, one can choose a set of patterns whose scores and/or frequencies are within specific range(s) and compact it by removing any pattern that is a subset of another pattern, even if it is maximal. This makes results easier to comprehend

5.5 Use case Modeling

Finally, when results are satisfactory, the interesting patterns discovered are annotated with any necessary semantic information to build use case models as described in section 4 to feed the forward engineering phase of CelLEST or for other usage.

6. An Experiment

In this experiment, we used a trace, 454 snapshots long, recorded while a user was retrieving information about the federal legislation in the USA from LOCIS. Part of this trace is shown in Figure 2.a. Let's denote this raw trace as *T-R0*. In this trace, the user performed two different information retrieval tasks several times. He accessed 18 LOCIS system screens with the frequencies shown in Table 1.

The trace was preprocessed and represented in *R1* format. Let the preprocessed trace be *T-R1*. Then a few different criteria were tried until, finally, the following criterion for pattern discovery was established on *T-R1*:

minL = 4
maxL = 9
minFreq = default, which is 1
Score = 5

Next, the pattern-extraction algorithm was applied and the 14 maximal patterns shown in Table 2 were discovered. The longest of these patterns is 7 items long.

The maximal pattern set was then compacted by removing patterns that are subsets of others. The patterns of the compact set are marked with a checkmark in Table 2. Note that a + sign is added to any pattern ID whose repetition count in *T-R1* is greater than 1 in some of the pattern instances.

Finally, sample instances of each pattern were reviewed to evaluate whether any of the resulting patterns corresponds fully or partially to an actual use case. This inspection revealed that two sub-patterns; 4^{+}-5-6^{+}-7^{+}-8^{+}-9 and 4^{+}-14-15^{+}-6^{+}-7^{+}, are interesting and represent two different use cases. The first is a sub-pattern of pattern 3 in Table 2, and the second is a sub-pattern of pattern 4.

The first use case is described earlier in section 4 and its model is shown in Figure 3. In the second use case, the user browsed the desired part of the currently open library catalog. Then he issued a *select* command to retrieve some records from the catalog. The *select* command constructs separate subsets of results for the specified search term, each for a different search field, e.g. one for the records that have the search term in the title, one for the records that have it in the abstract, etc. Then, the user issued a *combine* command to merge some of these subsets together into one set using some logical operators. Finally he displayed brief information about the items in this set and selected some items to display their full or partial information.

We used the approximate pattern matching algorithm to look for instances of both of the discovered patterns with up to 3 insertion errors. None existed for the first pattern and these three were discovered for the second:

- 4^{+}-14-15^{+}-*13*-6^{+}-7^{+}
- 4^{+}-14-*11*-15^{+}-6^{+}-7^{+}
- 4^{+}-*11*-14-15^{+}-*11*-6^{+}-7^{+}

The extra screens; 11 and 13, are in italic. By checking them in Table 1, one can see that 11 is an "Error" screen, that results from mistakes in the command issued and 13 is "Display List" screen that lists the numbers of the federal bills in a results set if a *display* command was issued with the parameter */list*. Hence, the conclusion is that the first instance is a slightly different scenario for the use case with an extra step of displaying the content of the combined results set before displaying brief information about the items in the set. While the last two instances include some noise resulting from user mistakes.

In the context of the CelLEST method, the patterns discovered are augmented with the information exchange between the user and the system on each screen and then are used for automatic construction of a new task-centered GUI. However, these patterns can be used for other general or specific program comprehension tasks, e.g. system re-documentation, requirements recovery, building help systems, etc.

Table 1: The LOCIS screens used in the experiment.

Screen ID	Screen Description	Frequency
1	Main LOCIS Menu	5
2	Fed Leg Menu	3
3	Welcome	3
4	Browse Results	69
5	Retrieve Results	31
6	Brief Display	83
7	Display item page 1/1 or 1^{st}/n	114
8	Display item page (2 or more/n)	29
9	Display item last page (n/n)	17
10	Help	2
11	Error	36
12	Search History	20
13	Display List	4
14	Select Command Results Page	13
15	Combine Command Results Page	14
16	Release Command Results Page	9
17	Comments & Logoff	1
18	Goodbye	1

Table 2: The maximal patterns retrieved from the LOCIS trace. The patterns of the compact results set are marked with checkmarks.

	Pattern	Frequency	Score	
1	4^{+}-5-6^{+}-7^{+}	10	6.64	
2	6^{+}-7^{+}-8^{+}-9-7^{+}	7	6.52	
3	4^{+}-5-6^{+}-7^{+}-8^{+}-9-7^{+}	5	6.52	√
4	4^{+}-14-15^{+}-6^{+}-7^{+}-4^{+}-14	5	6.52	√
5	7^{+}-8^{+}-9-7^{+}	9	6.34	
6	7^{+}-8^{+}-9-7^{+}-8^{+}	6	6.00	√
7	7^{+}-8^{+}-9-7^{+}-4^{+}	6	6.00	√
8	4^{+}-14-15^{+}-6^{+}-7^{+}	6	6.00	
9	4^{+}-14-15^{+}-6^{+}	8	6.00	
10	9-7^{+}-4^{+}-5	7	5.61	√
11	7^{+}-4^{+}-5-6^{+}-7^{+}	5	5.39	√
12	6^{+}-7^{+}-4^{+}-14-15^{+}	5	5.39	√
13	6^{+}-7^{+}-4^{+}-14	6	5.17	
14	7^{+}-4^{+}-14-15^{+}	6	5.17	

7. Summary and Future Work

In this paper, we discussed a behavior-analysis method for understanding the use cases of software applications. The assumption underlying this work is that inspecting "how the legacy application is actually used" can lead to understanding of "what uses it was intended to support" in the first place.

Our method is part of the CelLEST environment, developed to support legacy UI migration to Web-accessible platforms. It inspects traces of legacy screen

snapshots, sent to the user by the legacy application, during their interaction. These trace snapshots are identified as instances of unique behavioral screens, based on the screen classification produced by the CelLEST classifier, LeNDI. By mining recurring screen patterns in the trace, our method extracts the actual consistent uses of the interface, which potentially correspond to the use cases of the legacy application.

Our method is most suitable for mature stable systems with rich presentation. For batch systems or systems with very limited UI, there is not enough information in the system-user interaction traces to model the use cases.

In the context of CelLEST, our use case mining method is used for automatically extracting examples of user tasks, that are subsequently used for developing new Web-accessible interfaces for these tasks, using a "demonstrational programming" tool, called Mathaino. More generally however, it can be used for all activities where requirements reverse engineering is desired, such as platform migration and re-implementation, or user documentation, etc. In our future work, we will explore the applicability of our method to these activities.

Although the application presented in this paper used traces of interaction with 3270 legacy systems, our use case discovery method is applicable to other kinds of interaction traces, e.g. sequences of events in a windows-based application. This is another future work direction that we like to explore.

References

[1] Baeza-Yates, R. and Gonnet, G., *A New Approach to Text Searching*, Communications of the ACM, vol. 35, no. 10, 74-82, Oct. 1992.

[2] Baixeries, J., Casas, G. and Balcázar, J., *Frequent sets, sequences, and taxonomies: new, efficient algorithmic proposals*, Tech. Rep. LSI-00-78-R, *Universitat Politècnica de Catalunya* (UPC), Spain, 2000.

[3] Di Lucca, G., Fasolino, A., and De Carlini, U., *Recovering Use Case Models from Object-Oriented Code: a Thread-based Approach*, in Proc. 7th Working Conf. on Reverse Engineering (WCRE 2000), pp. 108-117, IEEE Computer Society Press, 2000.

[4] El-Ramly, M., Iglinski, P., Stroulia, E., Sorenson, P. and Matichuk, B., *Modeling the System-User Dialog Using Interaction Traces*, in Proc. 8th Working Conf. on Reverse Engineering (WCRE 2001), pp. 208-217, IEEE Computer Society Press, Oct. 2001.

[5] Kapoor, R. and Stroulia, E., *Simultaneous Legacy Interface Migration to Multiple Platforms,* in Proc. 9th Int. Conf. on Human-Computer Interaction, vol. 1, pp. 51-55, Lawrence Erlbaum Associates, Aug. 2001.

[6] Mortazavi-Asl, B., *Discovering and Mining User Web-page Traversal Patterns,* M.Sc. Thesis, The School Of Computing Science, Simon Fraser University, Canada, Apr. 2001.

[7] OMG, *The OMG Unified Modeling Language Specification,* version 1.3, OMG, 1999.

[8] Parnas, D., *Software Aging*, in Proc. 16th Int. Conf. on Software Engineering (ICSE'94), pp. 279-287, 1994.

[9] Paternò, F., *Task Models in Interactive Software Systems*, in Handbook of Software Engineering and Knowledge, vol. I, World Scientific Publishing Co., USA, 2002 (to appear).

[10] Wiegers, K, *Hearing the Voice of the Customers*, chapter 8 in Software Requirements, Microsoft Press, 1999.

[11] Wu, S. and Manber, U., *Fast Text Searching Allowing Errors*, Communications of the ACM, vol. 35, no. 10, 83-91, Oct. 1992.