Recovering Software Requirements from System-user Interaction Traces

Mohammad El-Ramly   Eleni Stroulia   Paul Sorenson

Department of Computing Science, University of Alberta
221 Athabasca Hall, Edmonton, Alberta Canada T6G 2E8
{mramly, stroulia, sorenson}@cs.ualberta.ca

Abstract. *As software systems age, the requirements that motivated their original development get lost. Requirements documentation is usually unavailable, obsolete, poor or incomplete. Recapturing these requirements is critical for software reengineering activities. In our CelLEST process we adopt a data-mining approach to this problem and attempt to discover patterns of frequent similar episodes in the sequential run-time traces of the legacy user-interface behavior. These patterns constitute operational models of the application's functional requirements, from the end-user perspective. We have developed an algorithm, IPM, for interaction-pattern discovery. This algorithm discovers patterns that meet a user-specified criterion and may have insertion errors, caused by user mistakes while using the application or by the availability of alternative scenarios for the same user task. The algorithm initially constructs a set of short patterns by exhaustively inspecting the traces and then iteratively extends them to construct larger ones, using a matrix data structure to reduce the number of pattern extensions explored, during each iteration.*

1. Introduction and Motivation

Any software application actively used in a real-world domain, requires to be continually evolved, through bug fixing, adaptations to its behavior and enhancements and upgrade of its functionality (Lehman's Laws of Software Evolution [12]). In this process, the original functional requirements, if they had once been properly documented, become blurred, outdated and eventually lost. This problem of "requirements' loss" becomes especially critical when the goal of the maintenance activity is to migrate the system functionality to a new platform, which is the objective of a substantial number of current IT projects that aim at Web-enabling antiquated legacy systems.

Recovering the necessary requirements to support the migration can be a hard challenge, considering that these requirements were captured at the very early stages of software development. Legacy system requirements are usually scattered between various documents, the source code, database triggers and the stakeholders of the system. Documentation is often poor, outdated, incomplete or unavailable. Legacy code is hard to understand as it is scarcely structured and includes "dead" or obsolete code and "glue" code of incremental updates that violate the original architecture. Requirements gathering from stakeholders via interviews or other techniques can be labor intensive, time consuming and inaccurate. In this work, we present a novel method for requirements recovery that is based on discovering and modeling the frequent user tasks accomplished using the legacy system under analysis.

In our work in the context of the CelLEST project [7][11] [18], we have been investigating the problem of supporting legacy interface migration to the Web. The fundamental methodological assumption underlying the CelLEST process is that more insight regarding the purpose of the application from the users' perspective could be gained by inspecting how the application is actually used. The actual run-time behavior of the application constitutes evidence of its functional requirements, as they are actually exercised by its current users. An accessible expression of the run-time behavior of an application is traces of the interaction between the application's user-interface and its users. It is a rich source of knowledge and a faithful representation of how the system is currently being used. Thus, unlike traditional reengineering approaches, CelLEST is not based on legacy-code understanding, but adopts an interaction-based approach to reverse engineering and program comprehension. Instead of understanding the structure of the application from its code, it models the tasks accomplished by the legacy-application users based on traces of their interaction with the application.

In this effort, we formulated the "requirements recovery" problem as an instance of the sequential-pattern mining problem. The patterns sought in this problem are frequent subsequences of user interaction with the legacy system user-interface, and hence are called interaction-patterns. The underlying intuition is that traces of the dynamic behavior of the legacy interface during its interaction with the user represent purposeful "walks" through the underlying dialogue model implemented by the legacy interface. Frequently occurring patterns in the traces can then be interpreted as "walks" in service of the same user task, and can thus be used as recaptured documentation of the tasks the user can accomplish using the legacy application, that is, the functionalities provided by this application. These patterns can be described using the use case notation or other similar notations.

The rest of the paper is organized as follows: Section 2 presents the related work in the area of requirements recovery. Section 3 discusses the overall CelLEST project and gives the context and the background of the "requirements-recovery as interaction-pattern mining" problem. Section 4 presents the necessary terminology and formally defines the problem. Section 5 presents our algorithm for interaction-pattern mining. Section 6 is a case study and evaluation of the algorithm. Finally, Section 7 summarizes our work to date, and concludes with the lessons we have learned and some pointers to future work.

2. Related Requirements Recovery Work

Requirements recovery research is still fairly scarce. Previous work in this area had explored a variety of methods that assume different input information and recover various different types of requirements.

In the REVERE project [16] natural language processing (NLP) methods were employed to recover software requirements from the available *documentation*, such as requirements specifications, operating manuals, user interview transcripts and data models. The method suffers from the well-known shortcomings of NLP and needs to be adapted (trained) to the various documentation styles, structures and notations, but provides rapid analysis for voluminous documentation.

Cohen [4] used Inductive Logic Programming (ILP) to discover specifications from C *code*. These specifications are in Datalog (Prolog with no function symbols). The software discovered two thirds of the specifications with about 60% accuracy, in a program containing over one million lines of source code. This is provided that training data is sufficient, otherwise results will contain numerous inconsistent specifications.

The AMBOLS project [13] aimed to recover requirements by employing semiotic methods and intensive interviews with the *stakeholders* to analyze and model the system behavior from various viewpoints.

In [17], *data* reverse engineering was proposed as a means for business rules recovery from legacy information systems. Particularly, an approach for extracting constraint-type business rules from database applications was outlined, but without an implementation or experimental evaluation.

Di Lucca *et al* [6] presented a method for recovering a use case model from *threads of execution* of object-oriented (OO) code. A thread is a sequence of method executions linked by messages exchanged between objects. Threads are triggered by input events and terminated by output events. However, most legacy systems were developed before the emergence of OO concepts and languages, and thus cannot be analyzed this way.

The work presented above represent different directions in exploring and tackling the requirements recovery problem. Researchers explored different available inputs, e.g.

existing documentation, people, code, data, and threads of OO program runs. To our knowledge, none of the methods presented was tested on large scale to claim that it is generally applicable or well established. It is possible to use different available inputs and methods to gather bits and pieces of legacy system requirements, cross verify them and then integrate them together to form the recovered system requirements as accurate and complete as possible.

In the CelLEST project we employ another yet unexplored, easy-to-collect input to recover the necessary requirements needed to support the CelLEST method for legacy interface migration to the Web.

3. Interface Migration in CelLEST

The CelLEST method consists of two phases. In its reverse engineering phase, the *LEgacy Navigation Domain Identifier* (LeNDI) prototype [7][18] is used to produce a state-transition model of the legacy interface behavior using traces of its current use. A trace is a sequence of screen snapshots interleaved with the user actions performed in response to receiving the snapshots on the user's terminal. Each state of the model corresponds to a distinct screen. Screen snapshots are classified into the distinct interface screens by a classifier induced after clustering the trace snapshots according to their visual similarity. Each transition in the model corresponds to a possible user action, that is, a sequence of cursor movements and keystrokes on a particular screen that causes the transition of the interface from the current screen to a new one.

Figure 1.a shows 20 consecutive screen snapshots from an example trace of interaction with the Library Of Congress Information System (LOCIS) through its IBM 3270 public connection (IP: 140.147.254.3). The trace was recorded while a user repeatedly retrieved detailed information about pieces of federal legislation. The keystrokes that occurred on each snapshot are shown as labels on the arrows. The solid-line snapshots constitute a complete instance of the information retrieval task. The user started by making the necessary menu selections to open the relevant library catalog. Then, he issued a *browse* (*b*) command with some keyword(s) to browse the relevant part of the library catalog file. Then, he issued a *retrieve* (*r*) command to retrieve a subset of the catalog items. Then, he displayed brief information about the items in this set using *display* (*d*) command. Finally, he selected an item using the *display item* (*d item*) command to see its full or partial information, e.g. the full legislation, its abstract, its sponsors list, etc.

LeNDI built the state-transition model corresponding to the input trace. Figure 1.b shows the part of the model relevant to the trace segment in Figure 1.a. The top left corner of each screen shows its ID, as assigned by LeNDI. The labels on the edges are models of the user actions that enable the transition of the interface from a screen to another.

After an expert user has reviewed and validated the built state-transition model, the next step is the discovery of

interaction-patterns, which is the core subject of this paper. The discovered pattern corresponding to the task instance of Figure 1.a is $\{4^+,5,6^+,7^+,8^+,9\}$, where + is one or more.

To provide a complete view of the interface-migration process of CelLEST, in service of which interaction-pattern discovery is employed, we also discuss briefly the subsequent steps of the process. Each set of task-specific traces, i.e., instances of a discovered interaction-pattern, represents multiple executions of a single user task. The corresponding user task is modeled, with the aid of some information from expert users of the legacy application, in terms of the information exchange between the user and the legacy application during the task. Figure 2 shows the

interaction-pattern corresponding to Figure 1.a, annotated with such interaction information, and a use case formulation of the pattern. Next, in the forward engineering phase of CelLEST, a prototype tool called Mathaino [11] is used to construct a declarative user-interface specification for the modeled task. This specification is also executable by a suite of special-purpose platform-specific components. Thus the new user-interface becomes a front-end for the original legacy user-interface, available in multiple new platforms, e.g. XHTML-enabled browsers or WAP devices [11]. The new interface executes the underlying application using its state-transition model, the task model, and an API to the data-transfer protocol used by the legacy system.

(a) A segment of an example trace of interaction with the legacy user-interface of LOCIS. @E means Enter key.

(b) The corresponding part of the state-transition graph. * is a mandatory argument and [*] is an optional argument.

Figure 1: An example trace of user interaction with the Library of Congress Information System.

(a) The interaction-pattern discovered for the information retrieval task of Figure 1.a, augmented with action locations. @ ?, 67 means that the user action occurs on the screen snapshot at an 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- Flip 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 using *display* command 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 until retrieving the needed details
Exit condition: The user retrieves the required information about the federal legislation of interest.
_____

(b) A textual description of the corresponding use case.

Figure 2: An example interaction-pattern and the corresponding functional requirement described as a use case.

4. Problem Statement

In this section we provide the terminology and formulation of the problem of discovering interaction-patterns in the recorded traces of interaction with a legacy user-interface.

1. Let *A* be the **alphabet** of legacy screen IDs, i.e. the set of IDs given by LeNDI to the screens of the legacy system under analysis.

2. Let $S = \{s_1, s_2, ..., s_n\}$ be a set of sequences. Each **sequence** s_i is an ordered set of screen IDs from *A* that represents a recorded trace of interaction between the user-interface of the legacy system and one of its users, similar to the partial trace shown in Figure 1.a.

3. An **episode** *e*, is an ordered set of screen IDs occurring together in a given sequence.

4. A **pattern** *p* is an ordered set of screen IDs that exist in every episode $e \in E$, where *E* is a set of episodes of interest according to some user-defined criterion *c*. We say that *e* and *E* "support" *p*.

   Depending on the nature of the application, and the type of patterns sought, interestingness criteria differ. We introduce ours in the sequel. We refer to the individual IDs in an episode *e* or a pattern *p* using square brackets, e.g. *e*[1] is the first ID of *e*. Also, |*e*| and |*p*| are the number of items in *e* and *p* respectively.

5. If a set of episodes *E* supports a pattern *p*, then the following must hold:

   - $p[1] = e[1]$ $\forall e \in E$
   - $p[|p|] = e[|e|]$ $\forall e \in E$
   - $\forall$ pair of positive integers (i, j), where $i \leq |p|$, $j \leq |p|$ and $i < j$, $\exists e[k] = p[i]$ and $e[l] = p[j]$ such that $k < l$.

   This indicates that the first and last IDs in *p* must be the first and last IDs of any episode $e \in E$, respectively, and that all IDs in *p* should exist in the same order in *e*, but *e* may contain extra IDs. Hence, $|p| \leq |e| \, \forall e \in E$. The above predicate defines the class of patterns we are interested in: these are patterns with at most a preset number of insertions. For example, the episodes {2,4,3,4}, {2,4,3,2,4} and {2,3,4} support the pattern {2,3,4} with at most 2 insertions.

6. The **location list** of a pattern *p*, written as *loclist* (*p*), is a list of triplets (*seqnum*, *startLoc*, *endLoc*), each is the location of an episode $e \in E$, where s_{seqnum} is the sequence containing *e*. *startLoc* and *endLoc* are the locations of *e*[1] and *e*[|*e*|] in s_{seqnum}, respectively.

7. The **prefix** of a pattern *p*, written as *prefix* (*p*), is the sub-pattern that results from removing *p*[|*p*|].

8. The **suffix** of *p*, written as *suffix* (*p*), is the sub-pattern that results from removing the first ID of *p*, i.e. *p*[1].

9. The **support** of a pattern *p*, written as *support* (*p*), is the number of episodes in *S* that support *p*. Note that *support* (*p*) = *loclist* (*p*)**.**length

10. The **density** of a pattern *p* supported by a set of episodes *E*, written as *density* (*p*), is the ratio of |*p*| to the average episode length of episodes $\in E$:

$$density\ (p) = \frac{|p| * support\ (p)}{\sum_{e \in E} |e|}$$

11. A **qualification criterion** *c*, or simply **criterion**, is a user defined quadruplet (*minLen*, *minSupp*, *maxError*, *minScore*), where, given a pattern *p*:

    - The **minimum Length** *minLen* is a threshold for |*p*|.
    - The **minimum support** *minSupp* is a threshold for *support* (*p*).
    - The **maximum error** *maxError* is the maximum number of insertion errors allowed in any episode *e* $\in E$. Thus, $|e| \leq |p| + maxError\ \forall\ e \in E$.
    - The **minimum score** *minScore* is a threshold for the scoring function we use to evaluate/rank the discovered patterns. This function is:

    $$score\ (p) = \log_2 (|p|) * \log_2 (support(p)) * density(p)$$

    Our experiments showed that this function is suitable and sufficient for our application as it considers and balances between the pattern length, its support and its density. The default values for *minLen*, *minSupp*, *maxError* and *minScore* are 2, 2, 0 and 0 respectively.

12. A **maximal pattern** is a pattern *p1* of length *l* that cannot be combined with any other pattern *p2* of length *l* to form a third pattern *p3*, where *p3* = *p2* + *p1*[*l*] if *suffix* (*p2*) = *prefix* (*p1*) or *p3* = *p1* + *p2*[*l*] if *suffix* (*p1*) = *prefix* (*p2*) and *p1*, *p2* and *p3* meet the *maxError* constraint, without loss of support.

13. A **qualified pattern** is a maximal pattern that meets the user-defined criterion, *c*.

14. A **candidate pattern** is a pattern under analysis that meets the *minSupp* and *maxError* conditions, but is not qualified yet.

Given the above definitions, the problem of interaction-pattern discovery can be formulated as follows:
Given
   1. an alphabet *A*,
   2. a set of sequences *S*, and
   3. a user criterion *c*
Find all the qualified patterns in *S*.

5. Interaction-Pattern Mining
5.1 Related Pattern Mining Problems and Algorithms

Recently, the problem of mining patterns in sequential data has received a lot of attention due to the emergence of many interesting applications for it in several areas.

From a data-mining viewpoint, the discovery of patterns in genetic and protein sequences is similar to our interaction-pattern mining problem. The bio-informatics community is interested either in deterministic patterns with noise, e.g. flexible gaps, wild-cards (don't care characters) and/or

ambiguous characters (which can be replaced by any character of a subset of *A*), or alternatively, in probabilistic patterns. Since bio-sequential data is usually very large, one popular search strategy is to discover short or less ambiguous patterns using exhaustive search, possibly with pruning. Then the patterns that have enough support are extended to form longer or more ambiguous patterns. This process continues until no more patterns can be discovered. Two elegant algorithms of this category are PRATT [10] and TEIRESIAS [9]. None of them handles patterns with insertions as is the case in our problem.

Some instances of the sequential-pattern mining problem were inspired by applications in the retail industry [1]. Variants of the Apriori algorithm were invented to solve this problem. Our problem formulation is different than [1], but similar to the one presented in [14] as "discovery of frequent episodes in event sequences". In this problem, discovered frequent episodes or patterns can have different types of ordering: full (serial episodes), none (parallel episodes) or partial and have to appear within a user-defined time window. The support of a pattern is measured as the percentage of windows containing it. Some Apriori-based algorithms were developed to tackle this problem, e.g. WINEPI and MINEPI [14] and Seq-Ready&Go [2].

The CelLEST interaction-pattern discovery problem aims at identifying fully ordered patterns only, possibly with a number of insertion errors less than a predefined upper bound. It differs from the formulation of [14] in that it does not restrict the pattern length with a window length. In [8], we addressed a simpler version of this problem, in which we discovered exact interaction-patterns with no insertion errors allowed, using an Apirori-based algorithm. But, since this severely limits the number and type of patterns retrieved, we needed to accommodate insertion errors. Hence, we developed an algorithm, called IPM (Interaction-Pattern Miner), to solve this case.

IPM uses the same strategy of developing longer candidate patterns from shorter ones. Similar to Web usage-pattern mining [15], IPM has three steps: preprocessing of input sequences, pattern discovery and pattern analysis. Unlike Apriori-based algorithms, IPM avoids multiple passes over the input by maintaining location lists of candidate patterns, which are used to generate the location lists of the candidate patterns of the next iteration. Another difference is that IPM uses a matrix data structure to greatly limit the number of combinations explored in each iteration of candidate pattern generation. IPM reports only the qualified patterns from all the candidates generated.

5.2 Preprocessing
An interaction trace is initially represented as a sequence *s* of screen IDs. We call this representation *R0*. *R0* often contains repetitions, resulting from accessing many instances of the same screen consecutively, e.g. browsing the pages of a library catalog. Repetitions can hinder the discovery of important patterns. For example the episode

{4,5,6,6,6,6,6,6,7} in Figure 3 does not support the pattern {4,5,6,7} if *maxError*<5. To avoid this problem, *s* is encoded using the run-length encoding algorithm that replaces immediate repetitions with a count followed by the repeated ID. Repetition counts are stored separate from the sequence. We call this representation *R1*. Figure 3 shows *R0* and *R1* for the trace segment of Figure 1.a.

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 3: Preprocessing interaction traces.

5.3 Pattern Discovery with IPM
The input to the IPM algorithm is a set of sequences *S* and a criterion *c*. The algorithm outputs all qualified patterns in *S*. The algorithm consists of two distinct phases.

First, it exhaustively searches the input sequences to identify all the candidate patterns of length 2 that meet the "minimum support" and "maximum error" conditions during an initialization phase (Procedure 1). For every such pattern, a location list is constructed. The candidate patterns are stored in a matrix $|A|$ x $|A|$ of pattern lists, *ptList*, whose rows and columns are labeled after the IDs $\in$ *A*. Each cell *ptList*[*i,j*] of the matrix contains every pattern *p*, such that *p*[2]= *i* and *p*[/*p*/]= *j*. For example, the pattern {1,3,4,2} will be stored in *ptList*[3,2].

In the second phase (Procedure 2), the algorithm recursively extends the candidate pattern set. For every pair of patterns *p1* and *p2* of length *l*, if *prefix* (*p1*) = *suffix* (*p2*), a new pattern *p3* of length *l*+1is generated such that *p3 = p2 + p1*[*l*], and is then stored in *ptList* [*p1*[1], *p1*[*l*]]. *p1* can only extend patterns in *ptList* [*p1*[1], *p1*[*l*-1]]. For example, if *p1* = {1,3,4,2}, then it will be used to extend the patterns of length 4 in *ptList* [1, 4] which have the format {?,1,?,4}, where ? refers to any ID $\in$ *A*. Clearly, the extension will succeed only with patterns of the format {?,1,3,4}, if any.

The location list of the extended pattern *p3* is constructed from the location lists of *p1* and *p2* (Sub-procedure 2.1) Locations of the episodes that support *p3* but have more than *maxError* insertion errors are excluded. If *support* (*p3*) (which equals *loclist* (*p3*)**.**length) is ≥ *minSupp*, then *p3 and loclist* (*p3*) are stored in *ptList*, otherwise *p3* is ignored. If *support* (*p3*) = *support* (*p1*) and/or *support* (*p3*) = *support* (*p2*), then *p1* and/or *p2* is marked as non-maximal. When no more candidates can be generated, the algorithm reports only the qualified patterns in *ptList*.

5.3.1 Procedure 1: Producing the Initial Candidate Pattern Set
Figure 4 depicts the first procedure of the IPM algorithm. Step 1 creates the pattern list matrix, *ptList*. Steps 3-14 are repeated for every input sequence $s_k \in$ *S*. Step 3 iterates over the IDs of s_k, from s_k [1] to s_k [|s_k| - *maxError*-1]. In steps 4-5, each ID is used to build a new pattern with each of its consecutive IDs up to *maxError*+1. For example if

$s_k=\{1,3,2,3,4,3\}$ and *maxError*=2, then $p[1]$ will be glued to each of $p[2]$, $p[3]$ and $p[4]$ separately, resulting in the generation of the patterns {1,3}, {1,2} and {1,3}. Steps 6-7 add the new pattern in *ptList*, if it is not already there. The location of the episode supporting the pattern is added to its location list in step 8. Steps 9-14 perform the same function as steps 2-8, but they handle the last *maxError* IDs of s_k. Note that the only cells of *ptList*, used by Procedure 1, are the diagonal cells. This is because for a pattern of length 2, $p[2]=p[|p|]$. Steps 15-18 remove from *ptList* any pattern whose support is less than *minSupp*.

Input: An alphabet *A*, a criterion *c* and a set of sequences *S*.
Output: All candidate patterns of length 2.
Steps:
1. **Create** a matrix of pattern lists *ptList* |A| x |A|

2. **For every** trace $s_k \in S$, $1 \le k \le |S|$
3.   **For** $i = 1$ **to** $|s_k| - maxError - 1$
4.   **For** $j = i + 1$ **to** $i + maxError + 1$
5.   **Construct** new pattern $p = s_k[i] + s_k[j]$
6.   **If** p NOT in *ptList* $[s_k[j], s_k[j]]$
7.   **then Add** p to *ptList* $[s_k[j], s_k[j]]$
8.   **Add** (k,i,j) to *ptList* $[s_k[j], s_k[j]]$.getLocationList (p)

9.   **For** $i = |s_k| - maxError$ **to** $|s_k| - 1$
10.   **For** $j = i + 1$ **to** $|s_k|$
11.   **Construct** new pattern $p = s_k[i] + s_k[j]$
12.   **If** p NOT in *ptList* $[s_k[j], s_k[j]]$ **then**
13.   **then Add** p to *ptList* $[s_k[j], s_k[j]]$
14.   **Add** (k,i,j) to *ptList* $[s_k[j], s_k[j]]$.getLocationList (p)

15. **For every** $id \in A$
16.   **For every** pattern p in *ptList* $[id, id]$
17.   **If** *ptList* $[id, id]$.getLocationList (p).length< *minSupp*
18.   **then Remove** p from *ptList* $[id, id]$

Figure 4: Procedure 1 pseudo-code.

5.3.2 Procedure 2: Generating Longer Candidate Patterns from Shorter Ones.

Figure 5 depicts the second procedure of the IPM algorithm. This procedure iterates as long as more candidate patterns can be generated. In each iteration, it executes steps 3-18. Steps 4-6 iterate over every pattern $p1$ of length l in *ptList*. Steps 7-8 check if $p1$ can be used to extend any pattern $p2$ from its end. Only the patterns in *ptList* $[p1[1], p1[l-1]]$ are inspected because these are the ones whose second ID $p2[2] = p1[1]$ and whose last ID $p2[l] = p1[l-1]$. If extension is possible, steps 9-10 generate the new pattern $p3$ and its location list. Step 11 checks if $p3$ satisfies the minimum support condition. If yes, step 12 adds $p3$ to *ptList*. Steps 13-16 mark $p1$ and/or $p2$ as non-maximal if they have the same support as $p3$. Step 17 sets the flag *morePatterns* to true to execute a new iteration. Step 18 increments the pattern length counter l for the next iteration. When no more candidates can be generated, steps 20-24 report only the qualified patterns.

Figure 6 depicts the pseudo-code for creating the location list of a new candidate pattern. It combines the location lists of two patterns $p1$ and $p2$ of length l to provide the location list of $p3$, where $p3 = p2 + p1[l]$. Step 2 iterates over the locations of the episodes supporting $p2$. Steps 3-5 retrieve *startLoc* and *endLoc* of such an episode $e2$. Step 6 retrieves the locations of the episodes that support $p1$ and satisfy some conditions. Assume such an episode $e1$, then:

- $e1$ and $e2$ should be in the same sequence
- $e1$ should not be a sub-episode of $e2$ and vise versa.
- The overlap of $e1$ and $e2$ should be at least l-1 long.
- The distance from *startLoc* of $e2$ to *endLoc* of $e1$, inclusive, should be no more than $l + 1 + maxError$.

Steps 7-9 construct the location list of $p3$ and remove duplicates. Finally, step 10 reports the results back.

Input: A matrix of pattern lists initialized with all candidate patterns of length 2 and their location lists and a criterion *c*.
Output: All the qualified patterns according to *c*.
Steps:
1. $l = 2$
2. **Repeat**
3.   *morePatterns* = false
4.   **For** every $a \in A$
5.   **For** every $b \in A$
6.   **For every** pattern $p1$ in *ptList* $[a, b]$ **with** $|p1| == l$
7.   **For every** pattern $p2$ in *ptList* $[p1[1], p1[l-1]]$
                       with $|p2| == l$
8.   **If** *suffix* $(p2)$ == *prefix* $(p1)$ **then**
9.   **Construct** new pattern $p3 = p2 + p1[l]$
10.   **Construct** *loclist* $(p3)$ (Sub-procedure 2.1)
11.   **If** *support* $(p3) \ge minSupp$
12.   **Then Add** $p3$ to *ptList* $[p1[1], p1[l]]$
13.   **If** *support* $(p3)$ == *support* $(p1)$
14.   **Then Mark** $p1$ as non-maximal
15.   **If** *support* $(p3)$ == *support* $(p2)$
16.   **Then Mark** $p2$ as non-maximal
17.   *morePatterns* = true
18. $l++$
19. **While** *morePatterns* == true

20. **For every** $a \in A$
21.   **For** every $b \in A$
22.   **For every** pattern p in *ptList* $[a, b]$
23.   **If** $|p| \ge minLen$ AND *score* $(p) \ge minScore$
                      AND p is maximal
24.   **Report** p

Figure 5: Procedure 2 pseudo-code.

Input: The location lists of patterns $p1$ and $p2$ of length l and *maxError*. The lists are sorted by *seqnum* and *startLoc*.
Output: The location list of $p3$, where $p3 = p2 + p1[l]$.
Steps:
1. **Create** a empty location list *Loc3*
2. **For** $i = 1$ **to** *loclist* $(p2)$.length
3. $loc2 = loclist(p2)$.getLocation(i)
4. $st = loc2.startLoc$
5. $end = loc2.endLoc$
6.   **Find a set** $Loc1 = ($**any** $loc1 \in loclist(p1)$ **such that**
                $loc1.seqnum = loc2.seqnum$ AND
                $st < loc1.startLoc \le end - l + 1$ AND
                $end < loc1.endLoc \le st + maxError + l))$
7.   **For every** $loc1 \in Loc1$
8.   **Add** a triplet $(loc1.seqnum, st, loc1.endLoc)$ **to** *Loc3*
9. **Remove** any duplicates from *Loc3*
10. **Return** *Loc3*

Figure 6: Sub-procedure 2.1 pseudo-code.

5.4 An Illustrative Example

Let us now illustrate the operation of the IPM algorithm with a simple example. Let $A = \{1,2,3,4\}$, $S = \{s_1, s_2\}$, where $s_1 = \{1,3,2,3,4,3\}$ and $s_2 = \{2,3,2,4,1,3\}$ and $c = $ (*minLen*, *minSupp*, *maxError*, *minScore*) = (2,2,1,). *minScore* takes the default value of 0. The objective is to discover all qualified patterns in *S* that meet *c*.

Tables 1 to 3 show the steps of applying IPM. Patterns are enclosed between curved brackets, e.g. {2,1}, and their locations in the input sequences are between parentheses, e.g. (2,3,5). Candidate patterns are shown in bold. Patterns with insufficient support are shown in gray for clarification, although they are not stored in *ptList*. Candidate patterns of the previous iteration that turned out to be non-maximal in the current iteration are shown in normal font and followed by ¬max.

Table 1 shows the pattern list matrix, *ptList*, containing all the initial candidate patterns of length 2 generated by Procedure 1. Table 2 shows *ptList* after the first iteration of Procedure 2, during which all candidate patterns of length 3 were generated and non-maximal patterns of length 2 were marked. Table 3 shows *ptList* after the second iteration of Procedure 2. Only one pattern of length 4 was discovered. Table 4 shows the discovered qualified patterns, their support, density and score.

5.5 Understanding the Extracted Patterns

After reviewing the discovered patterns, one can change the criterion *c* to narrow or widen the results set, if too few or too many patterns were retrieved. One can compact the results set by removing any pattern that is a sub-pattern of another one, even if it is maximal. Using the location list of each pattern, one can retrieve its instances to examine whether or not they correspond to a "real" user task.

6. A Case Study and Evaluation

To evaluate the IPM algorithm, we performed a case study with a real legacy application. We collected traces of interaction between the Library of Congress application (LOCIS) and a user who accessed federal legislation information by performing three different retrieval tasks with a variety of parameters. The user conducted three sessions, using the IBM 3270 public connection of LOCIS Each was recorded as a data sequence. Thus, $S = \{s_1, s_2, s_3\}$ where $|s_1|$, $|s_2|$ and $|s_3|$ are 454, 185 and 369 respectively. Part of s_1 is shown in Figure 1.a. LeNDI, the reverse engineering tool of CelLEST, was used to build the state-transition model corresponding to the three recorded traces, which is partially shown in Figure 1.b. The model has 26 nodes, each corresponds to a LOCIS system screen. So, the alphabet *A* is $\{1,2,3,\ldots,26\}$. The descriptions of these screens are in Table 5. The frequency (Fr) of a screen is number of times it was recorded in *S*.

After preprocessing *S*, the IPM algorithm was applied to it multiple times with different criteria, in order to limit the size of the results set. Table 6 shows the 19 patterns discovered using the last criterion we used, which is $c = $ (*minLen*, *minSupp*, *maxError*, *minScore*) = (5,8,2,6), ordered by their score. Compacting this results set gave the patterns shown with check-marks.

Table 1. Output of procedure 1.

Last ID / 2nd ID	1	2	3	4
1	{2,1} (2,3,5) {4,1} (2,4,5)			
2		{1,2} (1,1,3) **{3,2}** **(1,2,3)(2,2,3)** {2,2} (2,1,3)		
3			**{1,3} (1,1,2) (2,5,6)** **{3,3} (1,2,4) (1,4,6)** **{2,3} (1,3,4) (2,1,2)** **{4,3} (1,5,6) (2,4,6)**	
4				**{2,4} (1,3,5)(2,3,4)** **{3,4} (1,4,5)(2,2,4)**

Table 2. Output of procedure 2, iteration 1.

Last ID / 2nd ID	1	2	3	4
1				
2		{3,2} ¬max	{3,2,3} (1,2,4)	**{3,2,4}** **(1,2,5)(2,2,4)**
3		{1,3,2}(1,1,3) {3,3,2} {2,3,2}(2,1,3) {4,3,2}	**{1,3} (1,1,2)(2,5,6)** **{3,3} (1,2,4)(1,4,6)** {2,3} ¬max {4,3} ¬max {1,3,3} (1,1,4) {2,3,3} (1,3,6) {3,3,3} {4,3,3}	{1,3,4} {3,3,4}(1,2,5) **{2,3,4}** **(1,3,5)(2,1,4)** {4,3,4}
4			{3,4,3} (1,4,6) **{2,4,3} (1,3,6)(2,3,6)**	{2,4} ¬max {3,4} ¬max

Table 3. Output of procedure 2, iteration 2.

Last ID / 2nd ID	1	2	3	4
1				
2			**{3,2,4,3}(1,2,6)(2,2,6)**	{3,2,4}¬max
3			**{1,3} (1,1,2)(2,5,6)** **{3,3} (1,2,4)(1,4,6)**	**{2,3,4} (1,3,5)(2,1,4)**
4			{2,4,3}¬max	

Table 4: The final result: All the qualified patterns in *S*.

Pattern *p*	|p|	support (p)	Density(p)	Score (p)
{3,2,4,3}	4	2	0.80	1.60
{2,3,4}	3	2	0.86	1.36
{1,3}	2	2	1.00	1.00
{3,3}	2	2	0.67	0.67

Next, we reviewed sample instances of each interaction-pattern, to see if it corresponds fully or partially to a real user task. This inspection revealed that the three patterns shown in bold in Table 6 closely correspond to repetitive user tasks. The actual/complete interaction-patterns of these tasks are:

1) $\{4^+, 5, 6^+, 7^+, 8^+, 9\}$
2) $\{4^+, 14, 15^+, 6^+, 7^+\}$
3) $\{21^+, 22, 23, 22, 6^+, 7^+\}$

Table 5: LOCIS screen IDs, descriptions and frequencies.

ID	Screen Description	Fr	ID	Screen Description	Fr
1	Main LOCIS Menu	11	14	Select Result	20
2	Federal Leg. Menu	8	15	Combine Result	24
3	Welcome	8	16	Release Result	9
4	Browse Result	104	17	Comments & Logoff	1
5	Retrieve Result	43	18	Goodbye	4
6	Brief Display	147	19	Ready	1
7	Display item (1/1 or 1st) pg.	233	20	System Message	22
8	Display item (2/n or more/n) pg.	79	21	Livt Results (1/1) pg.*	22
9	Display item (last) pg.	65	22	Expand Results (1/n)pg.♦	20
10	Help	36	23	Expand/Livt Results (n/n, i.e. last) pg.	24
11	Error	51	24	Expand/Livt Results (2/n or more/n) pg.	5
12	Search History	44	25	Livt Results (1/n)	8
13	Display List	5	26	Expand Results (1/1)	14

*Livt command is used to view Legislative Indexing Vocabulary Thesaurus online.

♦*Expand* command combines *Livt* and *Select* commands.

Table 6: The qualified patterns retrieved from the LOCIS traces using $c = (5, 8, 2, 6)$.

	Pattern	*Support*	*Score*	*Density*	
1	6-7-8-9-7	19	9.01	0.91	
2	6-7-8-9-7-4	11	7.77	0.87	√
3	7-8-9-7-4	14	7.64	0.86	
4	4-14-15-6-7	12	7.57	0.91	
5	7-4-5-6-7	14	7.28	0.82	
6	6-7-4-14-15-6	10	7.26	0.85	
7	**6-7-4-14-15-6-7**	**8**	**7.04**	**0.84**	√
8	6-7-4-14-15	11	6.90	0.86	
9	22-23-22-6-7	13	6.81	0.79	
10	**7-4-5-6-7-8**	**9**	**6.81**	**0.83**	√
11	7-4-14-15-6	11	6.80	0.5	
12	7-4-14-15-6-7	8	6.65	0.86	
13	4-5-6-7-8	8	6.48	0.93	
14	6-7-9-7-4	10	6.32	0.82	√
15	6-7-8-9-4	11	6.31	0.79	√
16	7-9-7-4-5	9	6.25	0.85	√
17	**21-22-23-22-6-7**	**8**	**6.20**	**0.87**	√
18	14-15-6-7-4	8	6.06	0.82	√
19	4-5-6-7-9	9	6.02	0.82	√

Note that although *S* was processed in *R1* format. By checking the instances of each pattern in the original sequences in *R0* format, we knew which screens are consecutively repeated and added [+] to them.

The task model representing the functional requirement corresponding to the first discovered interaction-pattern was presented earlier in Figure 2. In the second task, 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. In the third task, the user's starting point was issuing a *livt* command to view the index terms related to the one he used as a parameter for *livt*. Then, he expanded some of the displayed terms using *expand* command, and finally displayed the needed information as in the two previous tasks. Indeed these three interaction-patterns correspond quite faithfully to the three information retrieval tasks, performed by the user.

7. Conclusions and Future Work

In this paper, we formulated the problem of requirements recovery as mining of sequential patterns in the interaction traces of the legacy interface behavior and we described the algorithm we developed to address it. In the context of the CelLEST project, our end goal is to develop automatic support for migrating the user-interfaces of legacy applications to the Web, in order to make the services that these applications provide accessible to a wider Web-based public. To that end, an intermediate objective is to recover the legacy-application's functional requirements, that is, to model the user tasks it supports. We view this problem as an instance of the sequential pattern-mining problem: user tasks are patterns of frequently occurring episodes in the legacy-interface run-time behavior traces. In this variant of the problem, the episodes supporting the discovered patterns match only approximately. Because the users may face exceptional conditions while executing their tasks, spurious intermediate states may exist in a variety of locations in some of the episodes. The IPM algorithm addresses this requirement. We presented an evaluation of our algorithm against a realistic case study with a real legacy application.

Our approach is a synthesis of a variety of related practices, in both the industrial sector and in academia. "Screen scraping", i.e., developing parser-and-driver components for extracting (and providing) information from (to) block-mode transfer protocol screens, has been a popular technique for wrapping legacy applications. However, reengineering research has largely ignored it and to our knowledge, no automatic support has ever been developed for it. Demonstrational programming [5] uses examples of similar interaction sequences on a graphical user interface in order to produce scripts that implement complex user tasks, thus customizing the original user interface. Conceptually similar types of activities are now conducted under the "adaptive user-interfaces" research agenda. In parallel, a variety of model-based methods have been proposed for user-interface design, many of them using user-task models, but to our knowledge, none has exploited usage scenarios. Our approach is currently being

adopted partially by Celcorp [3], the industrial sponsor of the CelLEST project. Our next objective in this project is to further evaluate our algorithm with real and artificial data on traces from different applications, and different interaction styles.

8. References

[1] Agrawal, R. and Srikant, R.. Mining Sequential Patterns. In Proc. 11th Int. Conf. on Data Engineering, IEEE Computer Soc. Press, 1995, 3-14.

[2] Baixeries, J., Casas, G. and Balcazar, J.L. Frequent Sets, Sequences, and Taxonomies: New, Efficient Algorithmic Proposals. Report Number: LSI-00-78-R, El departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Spain, Dec. 2000.

[3] Celcorp Web Site. http://www.celcorp.com

[4] Cohen, W. Recovering Software Specifications with Inductive Logic Programming. In Proc. 12th National Conf. on Artificial Intelligence, AAAI Press, 1994, vol. 1, 142-148.

[5] Cypher, A. (ed.). Watch What I Do: Programming By Demonstration, MIT Press, Cambridge, MA, 1993.

[6] 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 Rev. Eng., 2000, IEEE Computer Soc. Press, 108-117.

[7] 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 Rev. Eng., IEEE Computer Soc. Press, Oct. 2001, 208-217.

[8] El-Ramly, M, Stroulia, E., and Sorenson, P. Mining System-User Interaction Traces for Use Case Models. In Proc. 10th Int. Workshop on Program Comprehension, Oct. 2002, IEEE Computer Soc. Press. (To appear)

[9] Floratos, A. Pattern Discovery in Biology: Theory and Applications. Ph.D. Thesis, Department of Computer Sci., New York Univ., Jan. 1999.

[10] Jonassen, I. Methods for Finding Motifs in Sets of Related Biosequences. Dr. Scient Thesis, Dept. of Informatics, Univ. of Bergen, 1996,

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

[12] Lehman, M., Ramil, J., Wernick, P. and Perry, D. Metrics and Laws of Software Evolution – The Nineties View. In Proc. 4th Int. Software Metrics Symposium, 1997, 20-32.

[13] Liu K., Alderson, A. and Qureshi, Z. Requirements Recovery from Legacy Systems by Analysing and Modelling Behaviour. In Proc. Int. Conf. on Software Maintenance, Sep. 1999, IEEE Computer Soc. Press, 3-12.

[14] Mannila, H., Toivonen, H. and Verkamo, A. Discovery of Frequent Episodes in Event Sequences. Data Mining and Knowledge Discovery, Nov. 1997, vol.1, no. 3, 259 - 289.

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

[16] Rayson, P., Emmet, L., Garside, R., and Sawyer, P. The REVERE Project: Experiments with the Application of Probabilistic NLP to Systems Engineering. In Proc. 5th Int. Conf. on Applications of Natural Language to Information Systems, June 2000, 288-300.

[17] Shao, J and Pound, J. C. Extracting Business Rules form Information Systems. BT Tech. J., Oct. 1999, vol. 17, no. 4.

[18] Stroulia, E., El-Ramly, M., Kong, L., Sorenson, P. and Matichuk, B. Reverse Engineering Legacy Interfaces: An Interaction-Driven Approach. In Proc. 6th Working Conf. on Rev. Eng, 1999, IEEE Computer Soc. Press, 292-302.