Automated Software Engineering, 10, 271–301, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

User Interface Reverse Engineering in Support of Interface Migration to the Web

E. STROULIA stroulia@cs.ualberta.ca
M. EL-RAMLY mramly@cs.ualberta.ca
P. IGLINSKI iglinski@cs.ualberta.ca
P. SORENSON sorenson@cs.ualberta.ca
Computing Science Department, University of Alberta, Edmonton, AB T6G 2E8, Canada

Abstract. Legacy systems constitute valuable assets to the organizations that own them, and today, there is an increased demand to make them accessible through the World Wide Web to support e-commerce activities. As a result, the problem of legacy-interface migration is becoming very important. In the context of the CELLEST project, we have developed a new process for migrating legacy user interfaces to web-accessible platforms. Instead of analyzing the application code to extract a model of its structure, the CELLEST process analyzes traces of the system-user interaction to model the behavior of the application's user interface. The produced state-transition model specifies the unique legacy-interface screens (as states) and the possible commands leading from one screen to another (as transitions between the states). The interface screens are identified as clusters of similar-in-appearance snapshots in the recorded trace. Next, the syntax of each transition command is extracted as the pattern shared by all the transition instances found in the trace. This user-interface model is used as the basis for constructing models of the tasks performed by the legacy-application users; these task models are subsequently used to develop new web-accessible interface front ends for executing these tasks. In this paper, we discuss the CELLEST method for reverse engineering a state-transition model of the legacy interface, we illustrate it with examples, we discuss the results of our experimentation with it, and we discuss how this model can be used to support the development of new interface front ends.

Keywords: legacy-system reengineering, user-interface migration, reverse engineering, interface reengineering, clustering

1. Motivation and background

Legacy systems constitute valuable assets to the organizations that own them. Frequently, a legacy system is the sole repository of valuable corporate knowledge and the sole specification of the organization's business processes, as they have evolved over time. Unfortunately, such systems suffer from two important disadvantages. First, they generally have text-based user interfaces designed with a command-language interaction style. This requires that the users memorize numerous ad-hoc rules for the command-language syntax, making the interface un-intuitive and difficult to learn, especially for today's users, who are accustomed to interacting with form-based and graphical user interfaces (GUIs). Furthermore, due to their limited space for information presentation and simple text-entry-only input capabilities, they often require duplicate data entry or redundant navigation to accomplish a single user task. The second, and more important disadvantage is that, due to the proprietary platforms on which legacy applications run, they are accessible only to users internal to the organization that owns the application. With the advent of electronic commerce, an increasing number of organizations need to make their services available to their partners and customers on the web, and even to integrate their services with those of their partners, and this move is difficult in the case of most legacy systems.

Legacy system re-engineering (Pressman, 1996) projects are labor intensive, due to their high complexity and many constraints. Therefore, there is a pressing need for developing automated support for all re-engineering activities, including user-interface migration. It is not surprising, then, that this problem has been the subject of substantial research activity (Merlo et al., 1995). In general, there are two types of interface migration approaches: a new user interface can be "grafted" on top of the original application, after the code implementing the original interface has been deleted, or the original user interface can be wrapped with a new interface front-end.

If the original legacy application is not highly interactive, calling a set of relatively independent application procedures from a newly developed GUI is fairly simple and does not involve complex reverse engineering. The GUI can be developed independently and the user-initiated events are programmed to invoke the procedures in question (Gannod et al., 2000; Phanouriou and Abrams, 1997). To enable some configuration of the underlying applications and re-formatting of their outputs some reverse-engineering process might be required to identify the internal variables of interest and trace their values (Tucker and Stirewalt, 1999). If, however, the original application is highly interactive, there is a substantial amount of code implementing the original interface that has to be identified, extracted and replaced. Moore et al. (1994) describe a method for migrating graphical user interfaces, from one platform to another. This method uses a knowledge-based model to map the functionalities of the widgets in the user-interface toolkit of the original platform to those of the target platform toolkit. Given a widget in the legacy interface, the tool identifies candidate equivalent widgets in the target platform toolkit. The major disadvantage of this approach is that a new knowledge-based model has to be developed for any new combination of source-target platforms. A similar approach (Antoniol et al., 1995) has been developed to address the problem of migrating text-based interfaces to graphical user interfaces. The interesting difference is that since there is no source widget toolkit, the reverse-engineering process hypothesizes widgets from the code.

An alternative to grafting is to develop web-enabled user-interface wrappers for legacy systems. This solution involves the restructuring of the application in a web client/server architecture (Horowitz, 1998; Tan et al., 1998). The underlying idea is to wrap the legacy application in an application server that communicates with an HTML browser or a thick client through a protocol such as HTTP or Java Remote Method Invocation (RMI) in the case of Java clients, for example. This approach introduces an overhead in accessing the application functionality, which, however, is usually negligible in comparison to the latency introduced by the network between the client browser and the application server.

All the above approaches, both grafting and front-ending, have a common feature: they all adopt code analysis for reverse engineering the original application. However, in most cases, the code of the legacy application is a very poor expression of its design. It includes "dead code" and "glue code" implementing obsolete functionalities and incremental updates. In addition, due to its long-term evolution, the code is highly coupled and even small changes may have unpredictable side-effects. Furthermore, when individual code modules are reused in the context of the new client interface, there is a great danger of missing, or even violating, application logic built in the dialog implemented in the legacy interface.

In our CELLEST project we have adopted a front-ending approach to legacy interface migration, to avoid the risks involved with extracting and reusing isolated code modules. Furthermore, to be able to reuse the business logic implemented in the dialog of the legacy user interface, we have opted to reverse engineer a model of the legacy user-interface behavior instead of the legacy application code. The overall CELLEST method consists of a reverse-engineering phase, aiming at constructing a state-transition model of the legacy user interface, and a forward-engineering phase, aiming at developing user-interface specifications executable by special web-accessible translators. These new user-interface specifications enable the execution of user tasks by using the reverse-engineered model to emulate the user navigation through the legacy user interface. Our method is, to some extent, inspired by "screen scraping", a quite successful industrial practice in legacy-application migration, which has been largely ignored by academic research. They are similar in that they both expose some aspects of the legacy interface to the new interface front-end and they also use the legacy interface to drive the underlying application. However, the CELLEST method brings a substantial innovation to the traditional screen-scraping process. With screen scraping, a developer has to analyze the behavior of the interface and write code that is highly specific to the original application and its user interface to extract the information of interest from this interface. Instead, with CELLEST, a model of the legacy interface behavior and of the specific tasks of interest is semi-automatically constructed. These models, after they have been reviewed and revised, can be executed by multiple platform-specific translators. Thus, the models produced by CELLEST enable the simultaneous migration of the legacy interface to multiple target front-end platforms, where different screen-scraping applications would have to be developed.

The focus of this paper is the reverse-engineering phase of CELLEST, which produces a state-transition model of the legacy-application user interface. To illustrate the use of the model in the overall legacy user-interface migration process, we also briefly discuss the subsequent forward-engineering phase. The input of the reverse-engineering phase is a recorded trace of the user interaction with the legacy interface and its product is a state-transition model specifying the unique legacy interface screens (as states) and the possible commands leading from one screen to another (as transitions). The model states are identified as clusters of similar in appearance snapshots in the recorded trace. Next, the syntax of each transition from one screen to another is extracted as the pattern shared by all the instances of the transition in the trace.

In this paper, we describe the overall CELLEST environment (Section 2) and the corresponding interface-migration process it supports. We then discuss LENDI, which implements the reverse-engineering step of the process: we describe the major steps of the process, feature extraction, screen and action identification (Sections 3.1, 3.2 and 3.3), we illustrate the process with a case study and we present some experiments we have conducted to evaluate its effectiveness (Section 3.4). Then we briefly discuss the forward-engineering step of CELLEST (Section 4). Finally, we conclude with some lessons we have learned and our plans for future research (Section 5).

2. The CELLEST environment and the corresponding user-interface migration process

Let us illustrate the overall user-interface migration process supported by the CELLEST environment with an example scenario, from a real application, on which we have applied our method. Consider an insurance company that computerized its claim department separately from the customer database, and, as a result, owns two separate subsystems, i.e., *Customer* and *Claims*. Using the *Claims* subsystem, the user can retrieve from a sequence of different screens the data necessary for generating a report on the customer's accident. However, to do that, the user has to know the accident number, when usually he/she only knows the claimant's name. As a result, the user has first to search for the claim number in the *Customer* system. Now, the insurance company wants to let its lawyer partners use its systems to generate the required reports. The lawyers, however, are not on the company's intranet, so these services have to be made available through a client-server application, preferably web based.

Traditionally, to accomplish this goal, one would have first, to isolate and extract from the application code all the modules relevant to "claim number search" and "claim data retrieval" and then, make them available through a server application. In addition, a new web-accessible client application, with an interface easily learnable and usable by the lawyers, should be developed to access the server.

The CELLEST environment supports an alternative process, shown diagrammatically in figure 1. Using a specially instrumented emulator, traces of the insurance company employees interacting with the existing interface are captured. The emulator should be used by

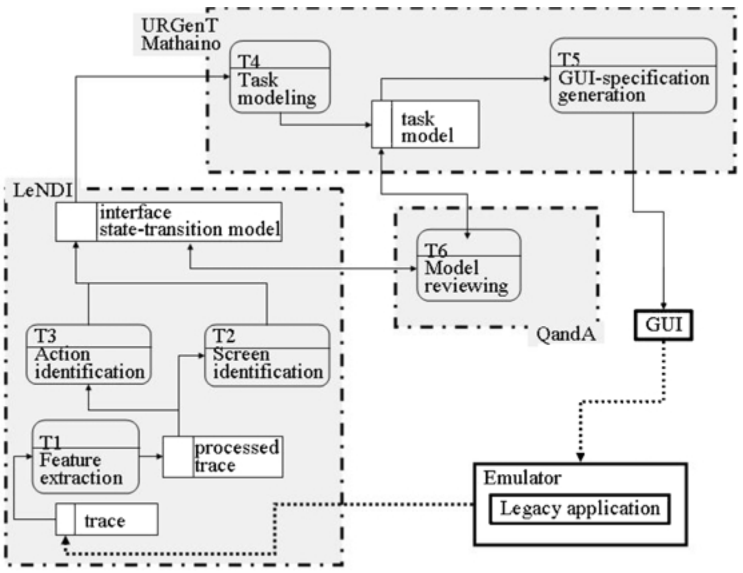


Figure 1. The tools of the CELLEST environment and the overall user-interface migration process.

all employees whose tasks may have to be migrated. The emulator provides the user with a text-based interface that mimics the original hardware terminals used to access the host system, on which the legacy application resides, by implementing the protocol of communication between the host and the emulator user interface. The emulator is instrumented so that it also records the interaction between the legacy application and its users.

A trace recorded by this emulator consists of a sequence of snapshots of the screens forwarded by the legacy application to the user's terminal. Between every two snapshots, the user keystrokes are recorded. The interface reverse-engineering step starts by translating each snapshot in the recorded traces into a vector of visual features (Task T1 in figure 1, described in Section 3.1). It then uses the processed trace to extract a model of the behavior of the legacy user interface (Tasks T2 and T3 in figure 1, described in Sections 3.2 and 3.3). This model is represented as a directed state-transition graph. The graph nodes correspond to the distinct interface screens, which are identified by clustering all the screen snapshots, contained in the recorded trace, according to their visual similarity. Each edge of the graph corresponds to an action that can be taken, i.e., a command that can be executed, when the source-screen node is visible to the user and leads to the destination-screen node. Given all the instances of the transition from one screen to another encountered in the recorded traces, a pattern of the command syntax enabling this transition is extracted. These three tasks are implemented by the LENDI prototype tool.

This model captures the overall behavior of the user interface, to the extent that it has been exercised by the users of the emulators and recorded in the captured traces. To construct models of the specific tasks that need to be migrated (Task T4 in figure 1, described in Section 4.1), in our example the "accident report" task, multiple instances of this task have to be recorded[1] The task model specifies the path on the interface state-transition model through which the user navigates, i.e., the distinct screens that the user visits while performing the task in question. In addition, the model specifies the flow of information between the legacy application and the user, i.e., the input provided by the user (as recorded in the trace), and the output displayed to the user (that the user has to annotate subsequently on the recorded trace). The final step in the CELLEST process (Task T5 in figure 1, described in Section 4.2) is the actual construction of the GUI. The GUI is a web-accessible application, i.e., an applet or a browser accessible web server. Based on the task model, it interacts with the user to receive(display) the expected input(output) information; it also translates this interaction to a sequence of calls to the emulator, similar to the calls that the original legacy user interface used to generate. Currently the emulator is able to emulate IBM 3270 and VT100 data transfer protocols. To date, in the CELLEST project we have focused almost exclusively on IBM 3270, which is a block-mode data transfer protocol. We have also experimented with VT100, which is a character-based protocol but can be emulated in block-mode. These last two steps of the process have been implemented in the URGENT and MATHAINO prototypes.

Since all the above tasks are inductive in nature, i.e., they construct models on the basis of examples, their results are products of "unsafe" inferences. The problem of evaluating whether or not the collected results are sufficient for learning a complete model of the interface is undecidable. In the context of the CELLEST process, we propose to address it in two ways. The first is methodological; we assume that recording emulators will be provided to all users whose jobs involve the functionalities that need to be migrated. In this way, we expect the collected traces to "cover" all interesting aspects of the legacy user interface, albeit not necessarily the whole user interface. In addition, the CELLEST environment includes QANDA (Questions AND Answers), a tool supporting the reviewing, verification and possibly revision of these results by an expert user. The role of the QANDA system in the CELLEST environment is to visualize the intermediate products of the process (screen clusters and task models) so that an expert user, familiar with the legacy interface that is being migrated, can inspect, validate or revise them (Task T6 in figure 1).

3. Interaction reverse engineering with LENDI

The purpose of the LENDI tool, which is the focus of this paper, is to produce a model of the legacy interface behavior, represented as a *directed state-transition graph*. The basic input to LENDI, as shown in figure 1, are the traces recorded during the users' interaction with the legacy application. For legacy systems that use a block-mode transfer protocol between the mainframe application host and the user terminals, such as the 3270 protocol, a trace is a sequence of screen snapshots interleaved with the actions that the user performs, such as character typing and function key presses. The intuition underlying the CELLEST interaction reverse-engineering process is that as the user interacts with the legacy interface, the underlying application goes through a sequence of distinct behavioral states, which correspond to the distinct screens of the legacy interface. The sequence of screen snapshots in the trace parallels the application's behavioral state sequence. The implication is that, identifying the distinct interface screens corresponds to identifying the behavioral states that the application goes through, during its interaction with the users. Consequently, identifying the different actions possible at a distinct screen corresponds to identifying the conditions of the transitions from one behavioral state to another. Therefore, the process of extracting the model of the interface behavior from the recorded traces consists of two steps:

1. the identification of the unique interface screens (i.e., the nodes of the state-transition graph) of the application interface and the corresponding predicates for evaluating whether a new snapshot is an instance of one of them, and
2. the identification of the possible user actions, that are applicable to each of these screens (i.e., the edges enabling the transition from one state to another).

The first sub-problem can be addressed by clustering the recorded screen snapshots: two instances of the same screen should appear more similar than two instances of two different screens. Thus, if all the screen snapshots were organized in clusters according to their visual appearance, each cluster should contain all the snapshots that are instances of some distinct system screen. Clustering is a generic problem with instances in a variety of application domains. In general, clustering algorithms are either batch, assuming that the complete set of input instances is available at the same time, or incremental, allowing for additional instances to be provided after initial clustering. Incremental algorithms, given a new instance, decide the cluster to which it belongs. Batch clustering algorithms are either top-down, starting with a single cluster and continuously decomposing it until a stopping criterion is met, or bottom up, starting with each instance belonging to a cluster by itself and joining clusters until a stopping criterion is met. Irrespective of their control flow, all clustering algorithms require a distance (or similarity) metric, on the basis of which to decide whether to split a cluster (in top-down algorithms) or whether to join two clusters (in bottom-up algorithms) or whether a new instance is sufficiently similar to an existing cluster (in incremental algorithms). Any such metric depends on a set of features that describe the input instances. The result of clustering is a partition of the entire snapshot set, i.e., a set of non-overlapping clusters, such that each recorded snapshot belongs to a cluster.

We have explored two clustering algorithms in the context of the screen-identification process: an incremental algorithm and a top-down algorithm stopping when the number of expected clusters has been reached. As we describe later in detail, these two algorithms have different knowledge requirements and each one is preferable under different usage scenarios. Through analysis of several legacy interfaces, we have identified a set of distinguishing screen features (described in Section 3.1); we use these features to transform the recorded trace snapshots into vectors of feature values, which are then input as instances to the clustering algorithms (described in Section 3.2). We have also developed, based on these features, the corresponding distance metrics required by the two algorithms.

Irrespective of which of the two clustering algorithms is used, the result is the identification of the distinct legacy interface screens and the classification of all recorded trace snapshots as instances of these screens. Using these classified snapshots as examples, a classifier is constructed which, given a new snapshot at run time, recognizes the screen in which each snapshot belongs.

The final step in the interaction reverse-engineering process is action recognition (described in Section 3.3). If there is a transition between two snapshots in the recorded trace, then there exists a command that enables a transition between their corresponding screens. Given all the examples of this transition in the trace, a pattern is extracted that characterizes the syntax of the underlying command. Therefore, at run time, new valid commands can be issued with new parameters.

The run-time ability to recognize the current screen forwarded by the legacy application and to issue commands so that a new desired screen may be obtained is crucial for the new web-based front-end to drive the legacy interface. In Section 4 we briefly discuss how the legacy-interface model constructed by LeNDI is used as the basis for developing the new web-based interface front-end.

3.1. Feature extraction

The snapshots in the interaction trace are recorded as two-dimensional buffers of ASCII characters. Thus, in principle, their similarity could be evaluated by their character-by-character comparison. Such comparison, however, would give rise to many "superficial" differences, due to the dynamic content of many screens. Therefore, after reviewing a variety of legacy interfaces and consulting with experts in the domain of legacy-interface design, we have developed three sets of characteristic features on the basis of which the similarity of two screen snapshots is calculated.

1. *Keywords:* The periphery of the screen often contains special information, such as date, time, system messages, page numbers, titles, or identifiers describing the screen function. LeNDI examines the first two non-empty lines at the top and bottom of the snapshot to detect any such keywords. An encoding of the discovered information and the general area where it appears, i.e., right/middle/left of line 1/2/23/24, is used as a feature.
2. *Screen layout:* The snapshots of dynamic screens that display information retrieved from the application's database back-end may appear quite different. However, they are usually laid out in some canonical structure. This is why we have developed two sets of layout features in LeNDI: projection profiles and layout structure. Projection profiles is a technique widely used in document analysis (Srihari et al., 1992). They are horizontal or vertical projections of all or parts of the snapshot into a histogram. In horizontal profiles, a histogram is constructed for the occurrences of the character(s) of interest per line. In vertical profiles, occurrences are counted per column. Default or user-defined thresholds are used to remove the effect of noise, i.e., insignificant occurrences of character(s). For example, to avoid recognizing a column separator when two "space" characters happen to be aligned in two consecutive rows, the threshold should be setup to 3. Values above the threshold are represented by ones, implying that a column/row separator has been recognized, and values under the threshold are represented by zeros, thus resulting in binary features.

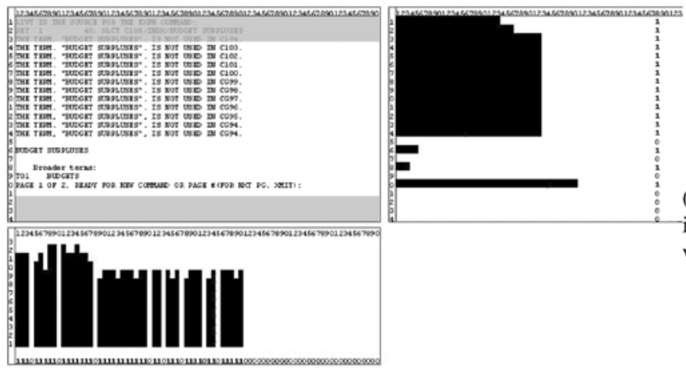
   Five extracted features are based on projection profiles. Two are the horizontal and vertical profiles of all characters. The third is the vertical profile for the numeric content of the snapshot, i.e., the count of digits per column. The fourth encodes the number of words in the two top and bottom lines of the snapshot. This is the only non-binary profile. The fifth is a projection profile of the most frequent character on the snapshot among a default or user defined set of special characters, which are frequently used to impose a geometric pattern on the layout of the screen.

   Layout description is inferred by examining if a snapshot contains a table, a list or a general structure. LeNDI uses the T-Rec algorithm (Kieninger, 1998) to detect tabular structures. If a table or a list is detected, LeNDI stores its specifications as a feature for this snapshot. A list's features are the number of its elements, the increment step, the order (ascending/descending), its first element and its left and right boundaries. A table's features are its upper left corner, its width, its length and the number of its columns. If none was detected, the snapshot is labeled as a general structure with no specification.

   To illustrate these different five types of projection profiles, figure 2 gives an example of all five types extracted for a legacy screen snapshot. Figure 2(a) shows the setup parameters used in this example. The upper and lower vertical cut parameters specify the number of lines from the top and bottom of the snapshot that should be ignored, before building the all-characters vertical profile. The vertical threshold value is used to decide whether to represent a column by one (if the number of characters in this column, ignoring the cut lines, is above or equal to the threshold) or by zero (if the opposite). The horizontal and numbers thresholds are used similarly. Finally the special character set describes which special characters to look for and what type of profile to build for each. For example, #(H) means that a horizontal profile should be built for the

Special Character Set	$-(H), \_(H), \#(V), :(V), *(V), ,(V)$
Upper Vertical Cut	3
Lower Vertical Cut	3
Horizontal Threshold	10
Vertical Threshold	3
Numbers Threshold	3

(a) The projection profiles parameters.



```
LIVT IS THE SOURCE FOR THE XXGN COMMAND:
SET  1          45: SLCT C105/INDX/BUDGET SURPLUSES
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C104.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C103.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C102.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C101.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C100.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C099.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C098.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C097.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C096.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C095.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C094.
THE TERM. "BUDGET SURPLUSES". IS NOT USED IN C094.

BUDGET SURPLUSES

    Broader terms:
T01    BUDGETS
PAGE 1 OF 2. READY FOR NEW COMMAND OR PAGE #(FOR NXT PG. XMIT):
```

(b) An example snapshot and its horizontal (top right) and vertical (bottom) profiles.

Figure 2. An example of projection profiles.

character #. Figure 2(b) shows an example snapshot, with the upper and lower vertical cuts shown in grey. Its horizontal profile is shown to its right and its vertical profile is shown at the bottom. Note that the least significant digit in all profiles except the number of words profile represent the 4 left most columns or the 4 topmost rows, depending on the direction of the profile. So the profile "1110111110111111..." will be represented by the string "..fdf7".

3. *Application-specific features:* Finally, the existence and locations of application-specific keywords, such as "Menu" and "Input" for example, can also be used for screen identification. Furthermore, the initial cursor location, i.e., the location of the cursor on the screen immediately after it is forwarded by the legacy application to the terminal interface, and the label to its left can be significant in distinguishing among screens.

Clearly, the set of features currently extracted from the recorded snapshots is not in any sense complete; they are the result of our domain analysis and it is possible that different types of legacy interfaces may require additional features in order to produce correct clusterings of their snapshots. The LENDI feature set is easily extendible and addition of new features would not disturb the rest of the screen-identification process.

The output of the feature extraction process is a feature vector for every snapshot. It is important to notice that the ability of each individual feature to discriminate between snapshots belonging to different screens differs from one legacy application to another, depending on the approach followed in designing the legacy user interface. Based on our reviewing of multiple legacy user interfaces, we believe that the suite of all the features we have developed is broad enough to deal with a variety of legacy interfaces in the information-systems domain.

3.2. Screen identification

After having transformed the recorded snapshots into vectors of the above features' values, the next step is to produce a partition of the entire snapshot set, i.e., a set of non-overlapping clusters such that each snapshot belongs to one, and only one, cluster. Each of the partition clusters is assumed to contain instances of the same distinct screen of the legacy interface. The end goal of clustering is to infer a classifier that can subsequently be used at run time to recognize new snapshots as instances of the legacy interface screens.

We have implemented two different clustering algorithms and corresponding similarity metrics, which we describe in the paragraphs below. The two algorithms have different input requirements and control flow and are appropriate under different knowledge conditions. The first, single-path incremental clustering, is an iterative process that requires several cycles of feature setup, clustering and result review. This is most suitable when the user is familiar with the clustering process and the available snapshot features but is still exploring the legacy system in hand and does not know it very well. On the other hand, the top-down clustering approach requires only a user estimate of the number of legacy screens to be modeled. This requires enough experience with the legacy system to estimate such a number. Using this estimate, the method provides a good initial partition without any further user setup. The two algorithms are described in the two subsequent subsections and their relative merits are described in Section 3.4.

3.2.1. Single-path incremental clustering. The first clustering algorithm in LeNDI is a single-path incremental clustering algorithm (van Rijsbergen, 1979). It requires as input the trace snapshots, a set of weights defining the relative important of the various features in calculating their similarity, and a similarity threshold, i.e., the minimum similarity value that is necessary to establish that two snapshots are instances of the same screen. The algorithm views clusters as centered at a representative snapshot, the *centroid*, and aims at maximizing the distance, i.e., the dissimilarity, between the centroids of the various clusters. It is incremental because it processes snapshots one by one and is single-path because it examines each snapshot only once. The partition is initialized to contain zero clusters. Each snapshot in the recorded trace is accessed and compared to the *centroid* of each partition cluster. The similarity of two snapshots is the weighted-sum of the similarity of their individual features. If a snapshot is similar enough to a cluster centroid, i.e., if the similarity score surpasses the defined threshold, then the snapshot is assigned to the centroid's cluster. If the new snapshot is not similar enough to any of the cluster centroids, then a new cluster is initialized with the new snapshot as its only instance and centroid.

The way in which individual feature similarity is evaluated depends on the type of the feature. The values of textual features, e.g., titles and codes, are compared using binary matching, i.e., if the feature values are exactly the same then the snapshots are similar in that feature dimension, and different otherwise. For layout features, a similarity factor is calculated based on the number of feature elements that the snapshots have in common. For example, there are four elements defining the table feature: the upper left corner, its width, its length and the number of its columns. When comparing the "table" feature of two snapshots, if they share the number of columns only, then their table-feature similarity is 25%. For projection profiles, similarity is calculated as the ratio of matching bits to the total number of bits in the profile. When comparing two binary projection profiles of the numeric content of two snapshots or of their most frequent characters, the mutual existence of 1s is considered more important than the mutual existence of 0s. This means that the fact that some column on both snapshots contains some numbers is more significant than that some column has no numbers on both snapshots. If a feature is missing on a particular snapshot, the algorithm can be configured to either ignore it in the calculation of the snapshot similarity, or to consider its absence as evidence that there exists a distinct screen that lacks the feature in question.

The algorithm can be configured to use as the cluster centroid either the first snapshot assigned in the cluster, or an "artificial" snapshot derived from the snapshots currently belonging to the cluster. In the latter case, the centroid is recomputed each time a new snapshot is added to the cluster by taking the most frequent value for every feature as the feature value for the centroid. In this manner, the sum of the distances between the centroid and cluster members is minimized.

After all the snapshots in the recorded traces have been clustered, the user can review the results, readjust the feature weights for the computation of the similarity measure and the threshold weight for establishing similarity and repeat clustering. Our experience has been that after a few repetitions (less than ten) clustering accuracy becomes satisfactory, i.e., at 90%. If there are still mis-clustered snapshots, the user can move these snapshots to the clusters where they belong, through the drag-and-drop interface of QANDA, the visualization tool of CELLEST.

After the user has reviewed the partition and no further changes are required, a pattern can be inferred as the "signature" for every cluster in the partition, to be used for classifying new snapshots at run-time. The signature is a generalized pattern of the feature values common in all screen instances. It consists of an artificial feature vector with the values shared by all the cluster members and a "don't care" character, '?', wherever there is no common value. For example, assume a cluster with three instances and with three features for each instance: two multi part features and one string feature. Assume the following feature vectors for the three instances: ("Claims Menu", 1089, 101010), ("Customer List", 3481, 101011), ("Help", 1081, 101011). The signature of this cluster is ("?", ??8?, 10101?). This signature pattern can be used at run time to recognize whether or not a new snapshot belongs to this cluster, i.e., is an instance of the screen corresponding to this cluster. Given a new snapshot, LENDI computes its feature-value vector and evaluates whether it matches any of the cluster signatures. If the feature vector matches a single signature, then LENDI recognizes the snapshot as an instance of the corresponding cluster. If no cluster signature matches, LENDI informs the user that the current snapshot was not recognized. If more than one signature match, LENDI prompts the user to select the best match.

3.2.2. Two-phase top-down clustering. The single-path clustering algorithm, described in the above section, requires substantial parameter configuration from the user. The user has to define the relative weights of the various features and the required similarity threshold. Although LENDI provides default values for these parameters, fine-tuning them is necessary to accomplish precise clustering, and this process can be un-intuitive. For that reason, we

Cluster(p, k)

```
p: the current partition
k: the expected number of the legacy-application screens

while (nClusters < k or maxIncoherence(p) > threshold )
    { Cluster c = findMostIncoherentCluster(p);                        [1]
      Split bestSplit = null;
      for each Feature f                                               [2a]
        for each Value v of f in Snapshots in c                        [2b]
          { Split newSplit = new Split();
            for each Snapshot s in c
              if (s.getFeature(f).getValue() == v) newSplit.accept.add(s);
              else newSplit.reject.add(s);
            if ( newSplit.maxIncoherence() < bestSplit.maxIncoherence() )
              bestSplit = newSplit;
          }
      p.remove(c);                                                     [3]
      p.add(bestSplit.accept);                                         [4a]
      p.add(bestSplit.reject); }                                       [4b]
```

Figure 3. The pseudo-code for the basic top-down unsupervised clustering algorithm.

have explored a second clustering algorithm for LENDI's screen-identification process. This second algorithm is a two-phase top-down clustering algorithm, and is shown in pseudo-code in figure 3. It requires as input the trace snapshots and an estimate of the number of distinct screens in the legacy interface, information that an experienced user of the legacy interface would be able to provide.

Initially, the partition is initialized to contain a single cluster, containing all input snapshots. As long as the current number of clusters is less than the estimated number of clusters, the algorithm identifies the most *incoherent* cluster and splits it into two new clusters, in a way that minimizes the incoherence of the resulting clusters. Incoherence is measured as the average distance of every instance in a cluster from every other instance in the same cluster. The algorithm uses the same features as the ones used in the first algorithm, with every multi-element feature broken into a number of single-element features. All the features are equally weighted and are treated as having discrete, non-ordinal values. If two instances have different values for a feature, that feature contributes its weight to the distance measure. The distance between two instances is then simply the sum of the weights of their differing features.

The figure 3 illustrates the first phase of the top-down clustering algorithm, when the original partition cluster is recursively decomposed in smaller, less incoherent clusters. In each step, the algorithm identifies the most incoherent cluster (line 1) and then calculates the incoherence of the clusters that would be produced if this cluster were split according to each feature-value combination (nested loops in lines 2a, 2b). The feature-value that results in the smallest maximum incoherence is selected; then the cluster is removed (line 3) and it is replaced by two clusters: one that contains the instances with the chosen feature value (line 4a), and one that contains all the rest (line 4b). Note that, when this first phase of the algorithm is complete, the produced partition is associated with a decision tree that defines how this partition was constructed, i.e., which feature and value were used for splitting each intermediate cluster corresponding to the internal tree nodes. This decision tree can be used directly for classifying new snapshots; it functions similarly to the cluster signatures that are being derived after the completion of the single-path incremental clustering algorithm.

As was the case with the single-path clustering, the LENDI user can examine the partition produced by the top-down algorithm using the QANDA interface, drag and drop mis-clustered snapshots to their correct clusters, and then request a revised decision tree to be constructed. The revision of the partition produced by the single-path algorithm involved the recalculation of the cluster centroids and the recomputation of the cluster signatures. The revision of the partition produced by the top-down algorithm involves the reorganization of the cluster tree and the corresponding re-specification of the predicates splitting an intermediate cluster to each children.

We use the MoJoPlus algorithm (Tzerpos et al., 1999; El-Ramly et al., 2001) for comparing two partitions to calculate the minimum number of snapshot(s) moves from a cluster to another and joins of two clusters together, that would have the same effect upon the original partition as the user's feedback. Then, the decision tree is revised to reflect the changes, as shown in figure 4. If two clusters, A and B, need to be joined (line 1), the algorithm simply changes the leaf node A to B, or if A and B share the same parent node, it eliminates the decision node and collapses A, B and their parent. If the instances of a subcluster C of

Revise(tree, moves)

```
tree: the current decision tree
moves: the MoJo moves characterizing the difference between the current
       tree and the authoritative one constructed by the user
A,B,C,C1,C2: clusters

for each move in moves {
    if (move == JOIN(A, B))                                          [1]
{ nodeA = getNode(tree A);
          nodeB = getNode(tree B);
          if (nodeA.parent() == nodeB.parent()) nodeA.parent().set(nodeA);
          else nodeB.set(nodeA); }
    if (move == MOVE(C from A to B)                                  [2]
                    //C is the subset of A that needs to move o B
        { if (A.size() > B.size())                                   [3]
             { Split A on shared features of A not shared by B+C;
               Return if successful;
               Split A on shared features of B+C not shared by A;
               Return if successful; }
          else { Reverse the order of statements in if clause; }
          if (B != null)                                             [4]
             { SplitNodesOnFeatures(A, null, C);
               Return; }
          if (C.size>1)                                              [5]
             { (C1, C2) = bestSplit C;
               SplitNodesOnFeatures(A, B, C1);
               SplitNodesOnFeatures(A, B, C2); }
```

Figure 4. The pseudo-code for the decision tree extension algorithm.

cluster A need to move from A to B (line 2), the algorithm proceeds to discover either the features that could be used to distinguish these instances from the rest of the instances in A, or the features that these instances share in common with all the instances in the target cluster B (line 3). If instances are being moved from a larger cluster to a smaller cluster, the tree-revision algorithm first looks for features distinguishing the instances in the larger set from the instances being moved and those in the destination set. If the instances are moving from a smaller cluster to a larger one, the tree-revision algorithm first looks for features shared by the moved instances and the destination set, but not by the ones in the origin set. If the first of these feature quests fails, the alternative is tried next. If that second quest fails, the algorithm recursively tries again, this time ignoring the features in the destination set (line 4). If this still fails, the moved instances are split into groups according to the best split test for minimizing the maximum cluster incoherence, and then each resulting group is checked recursively for distinguishing features (line 5). If the algorithm recurses down to a single instance, and no distinguishing feature can be found, the algorithm simply reports the failure and proceeds. This situation is, in fact, seldom encountered in all the legacy system traces that we have tested. Once a set of distinguishing features has been found, the algorithm currently selects one at random and uses it to create a new decision node to distinguish the instances from their initial classification. We experimented with various heuristics for selecting among a set of distinguishing features, and we evaluated their effectiveness with ten-fold cross-validation. None proved more reliable than random selection.

3.3. Identification of actions

After snapshot clustering is completed, the clusters produced correspond to the distinct screens of the legacy interface. The next step in LeNDI's process is to identify the edges in this graph, which correspond to the actions enabling the user to navigate from one screen of the legacy application interface to another.

Most legacy interfaces adopt a mix of function-key, menu-driven, command-driven, and form fill-in interaction. In the function-key interaction style, the interface implements a well-structured dialog with the user. At each point of this dialog, the user presses one of a small set of function keys to select one of the corresponding alternative options. A similar kind of interaction can be implemented in a menu-style interface. Such interfaces present the user with a list of items, each of which can be selected by moving the cursor to its location and pressing a function key. In the command-driven interaction style, the user issues textual commands to the system. A command language is specified in terms of the vocabulary of possible command names and the syntax of these commands in terms of the arguments they require and the options they allow. The command-driven interaction style enables a more dynamic system-user interaction, since the transitions of the system from one state to another are caused by possibly complex, multi-parametric commands instead of simple function-key presses. Finally, in the form fill-in interaction style, the interface presents the user with forms that require the entry of specific types of information at particular locations on the screen. The completion of the form is signaled to the system with the press of a function key or the typing of a command at a particular command line.

```
Transition      := start-screen action end-screen
action          := action-item*
action-item     := location data-item || location data-item function-key
location        := x-y-coord || range || null
x-y-coord       := [1,80],[1,24]
range           := x-y-coord, x-y-coord
data-item       := keyword* argument* option*
keyword         := string ∈ { keywords }
argument        := string
option          := string ∈ { options }
function-key    := {PF1, PF2, PF3, ..., Enter}
start-screen    := screen-state-id
end-screen      := screen-state-id
screen-state-id:= integer
```

Figure 5. A grammar for describing transitions in legacy systems.

To date, our research in LENDI has focused on systems adopting a combination of function-key and command-driven interaction style, since after reviewing several legacy systems we have found that this combination occurs very often. LENDI possesses a general "transition model", shown in terms of a BNF grammar in figure 5. According to this model, each *transition* from a *start state* to an *end state* is caused by an *action* which may consist of one or more *action items*. An action item involves the entry of a *data item* at a particular *location* of the screen, which may be static or may vary dynamically within an area range. An action item may conclude with the press of a *function key*.

To perform action modeling, LENDI groups the snapshots of each cluster according to the destination of the user action performed on them. LENDI assumes that there is a single action leading from a start screen-state to an end screen-state, therefore all the transitions in one group must be instances of the same action. Next, LENDI attempts to infer the command form(s) and/or the function key that defines this action, assuming that it conforms to the general model described by the transition-model grammar.

LENDI starts analyzing each group of action instances, one word at time, starting with the first word in all instances. It uses a set of heuristic rules for command-language design to discover any relations between the four most frequent terms that appear as the first word in various instances of the same action and whether any of these terms is an optional or mandatory command keyword or argument. According to these rules, LENDI assumes that if there are different versions of the same command name they will most likely be prefixes of a canonical command name or substrings of this name, possibly with the vowels removed. LENDI examines only the 4 most frequent terms, since our experience showed that it is unlikely that more than four forms of the same command will be available. In order for a particular string to be identified as the canonical command keyword, its different variants have to appear frequently enough, i.e., at least 33% of the times that the action occurred. If no keyword appears sufficiently often, LENDI assumes that the command name is implicit, and that the user has to only enter its arguments. It assumes that an argument is optional if it does not appear in some of the action instances, otherwise it assumes it is mandatory. The same analysis is applied to the second word, and so on. LENDI assumes that the command

First word	Second Word	Third Word	Function Key
R	Farm	Loans	enter
R	xxx		enter
R	term=tax	Deductions	enter
R	tax	Deductions	enter
R	B6		enter
R	S1		enter
R	Elections		enter

(a) The action instances

R * [*] (enter)

(b) The inferred model for the action

Figure 6. An example of modeling user actions in a command-driven legacy system.

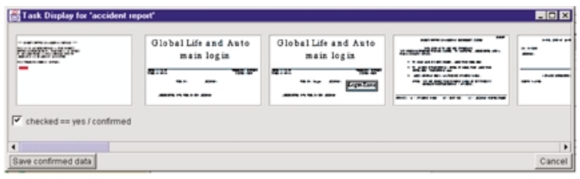
keyword, if any, can be at any position, and is not necessarily the first word. LENDI collapses the collected hypotheses in a compact form.

An example of modeling the retrieve command of a command-driven library system is shown in figure 6. LENDI examines the first word, which is *R* for all the examples and concludes that *R* is a compulsory keyword for this action. Then it examines the second word. By applying the rules mentioned above, no relation can be discovered between the words in the second position and none of them appears more than 33% of the time. The conclusion is that the second word is an argument for the command that is mandatory because all the instances have a second word. Doing the same analysis for the third word concludes that it is an optional argument. The inferred model is shown in figure 6(b), where * means a mandatory argument and [*] means an optional argument. The command R (or RETRIEVE) takes, as an argument, a phrase of one word at least, searches the library catalog indexes for it, and creates a results set for the matching items.

3.4. LENDI evaluation

To illustrate the overall user-interface reverse-engineering process, implemented in LENDI, we will revisit the example case study we introduced in Section 2, of developing a new web-based interface, accessible by the insurance company lawyers for generating accident reports. Next we will provide some quantitative evidence for the effectiveness of the process, using some publicly available library applications.

3.4.1. LENDI on the "legacy insurance application". The first step in the process is the collection of traces of the legacy insurance application. In this phase, the legacy-system users perform their tasks as usual; the emulator middleware is transparent and completely unobtrusive. Figure 7(a) shows a sequence of recorded screen snapshots, as seen through the LENDI user interface. In this case study, the LENDI user chose the incremental screen clustering algorithm for the screen-identification step. Ideally, this step would be performed by a user of the insurance application who would have had a tutorial of the LENDI process. It was actually performed by a developer in the CELLEST project (not the primary LENDI developer). Figure 7(b) shows a dialog box, where the LENDI user configures the process by specifying the combination and the relative weights of the particular features that will be used for clustering the trace snapshots.



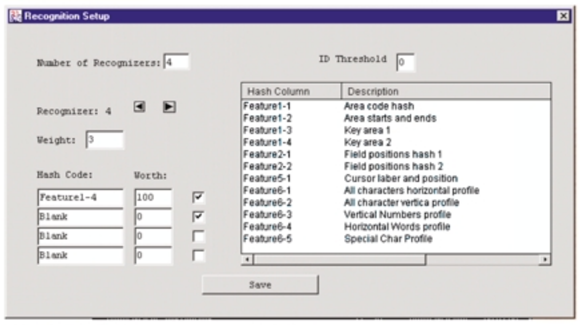
Task Display for 'accident report'

Global Life and Auto main login

Global Life and Auto main login

☑ checked == yes / confirmed

(a) Reviewing the recorded trace.



Recognition Setup

Number of Recognizers: 4

ID Threshold 0

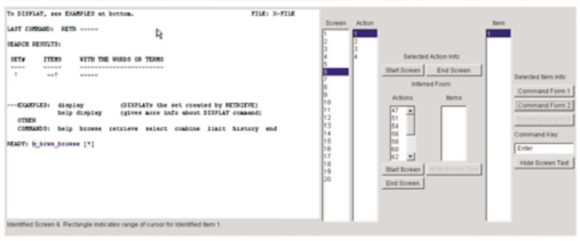
Recognizer: 4

Weight: 3

Hash Code:	Worth:	
Feature1-4	100	✓
Blank	0	✓
Blank	0	
Blank	0	

Hash Column	Description
Feature1-1	Area code hash
Feature1-2	Area starts and ends
Feature1-3	Key area 1
Feature1-4	Key area 2
Feature2-1	Field positions hash 1
Feature2-2	Field positions hash 2
Feature5-1	Cursor laber and position
Feature6-1	All characters horizontal profile
Feature6-2	All character vertica profile
Feature6-3	Vertical Numbers profile
Feature6-4	Horizontal Words profile
Feature6-5	Special Char Profile

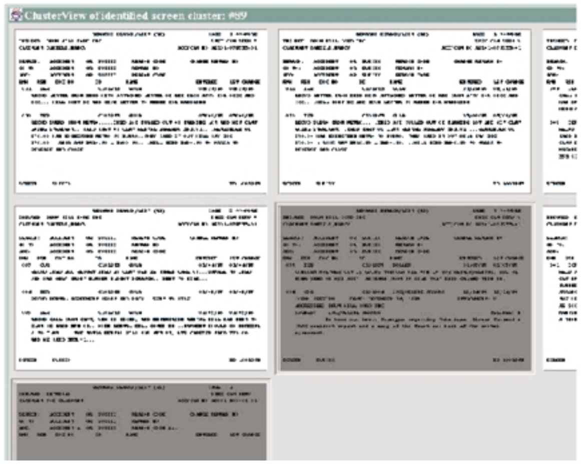
(b) Configuring the features for the single-path clustering algorithm.



```
To DISPLAY, see EXAMPLES at bottom.                    FILE: X-FILE

LAST COMMAND:  RETR -----

SEARCH RESULTS:

 SET#     ITEMS     WITH THE WORDS OR TERMS
 ----     -----     ----------------------
  !        --!      -----


---EXAMPLES:  display          (DISPLAYs the set created by RETRIEVE)
              help display     (gives more info about DISPLAY command)
   OTHER
   COMMANDS:  help  browse  retrieve  select  combine  limit  history  end

READY: b_brws_browse [*]
```

Identified Screen 6. Rectangle indicates range of cursor for identified item 1.

(c) Reviewing the actions possible at a screen.



ClusterView of identified screen cluster: #89

(d) Reviewing the snapshots of one of the clusters.

Figure 7. Snapshots form the CELLEST environment interface, during the reverse-engineering phase using LENDI.

Once LENDI completed the snapshot clustering, the produced clusters were reviewed. In our example, the first screen of the insurance application is fairly static—it contains a title in the first line with the keyword "SPLASH". The title feature is very successful in clustering all the instances of this screen. However, the review process revealed classification errors in other snapshot clusters; the process of configuration, identification and review was actually repeated nine times in this case study, until the analyst was satisfied with the result of the classification. Figure 7(d) shows the interface enabling the user to review all the snapshots of a cluster. The intent of this view is to reveal non-uniform clusters, i.e., clusters with snapshots that are not instances of the screen represented by the cluster, such as the dark gray snapshots in this picture.

When the LENDI user was satisfied that no snapshots were misclustered, he proceeded to the action-identification step. We will discuss in detail one particular command. When a legacy user has retrieved the claim number from the *Customer* subsystem, he/she navigates to a screen behaving as a menu where he/she can enter the claim number and then type a keyword that determines the kind of report he/she wants to see, i.e., "xs" or "c1" for "expense summary" and "medical claims" respectively. The second keyword determines the destination screen of the user's action. Nine instances of the transition to the "Expense Summary" screen were recorded, as shown in Table 1.

LENDI's algorithm for action recognition inspected these instances and concluded that all of them terminate with the same function key, "Enter", but there is no instance with only the function key and no text content. Then it proceeded to consider the first word of each instance as a candidate command name. The most frequent words are "120921", "123233" and "045536" which appear in two instances each. They are not similar to each other according to the criteria defined in the algorithm. Each of these words appears in only 20% of the instances. So, none of them qualified to be a command name; the inference is that the command must always start with a variable string. Then the algorithm considered the second word in every instance. There is only one word, "xs", that appears all the time in this position. So, the conclusion is that this is a mandatory keyword. As a result, the syntax for this command is specified as "* xs (Enter)".

Figure 7(c) shows the QANDA window on which the LENDI user reviews the results of the action recognition. The representative of a cluster is shown at the left of the interface. The interface shows that four actions were modeled for the screen shown, each causes a transition to a different target screen. The first one has two forms as the active buttons to the

Table 1. Recorded instances of the transition to the expense summary screen (node named *sum_display* in figure 11).

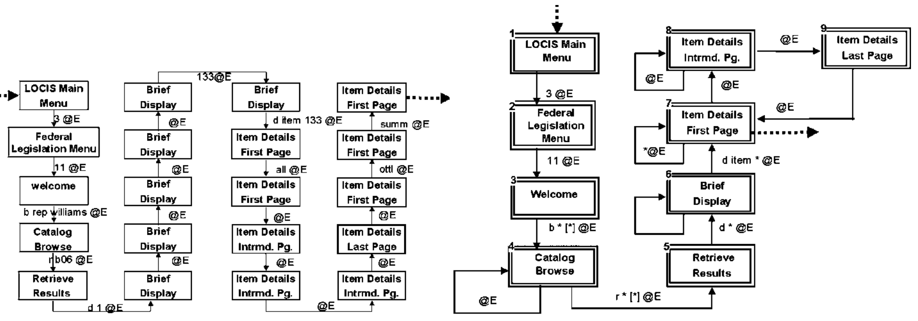
	First word	Second word	Control key
1	120921	xs	Enter
2	123233	xs	Enter
3	022323	xs	Enter
4	265765	xs	Enter
5	123233	xs	Enter
6	045536	xs	Enter
7	120921	xs	Enter
8	045536	xs	Enter
9	986443	xs	Enter

right of the window suggests. One of these forms is shown on the lowest line of the cluster representative, which is *b_brws_browse*[ ](*Enter*), where _ is used to separate the variant forms of the same keyword. The instances used to model this action are listed in the middle list with a scroll bar, where one can select any of them to view along with the corresponding destination snapshot.

3.4.2. Some quantitative experimental results with LENDI. Let us now discuss the efficiency and accuracy of LENDI's reverse-engineering of the legacy user interface. We report here on the results of our experimentation with an IBM 3270 trace of the public on-line Library of Congress Information System (LOCIS, locis.loc.gov). It was recorded while a user was browsing the library catalogue, retrieving sets of catalogue entries, displaying them, and running into some system errors. This trace is 406 snapshots long. Manually, a user built an authoritative partition for this trace, which had 17 distinct clusters. Some screens had only 1 or 2 snapshots in the trace, while others had up to 157. Figure 8 depicts a segment of the LOCIS trace and a part of the derived model.

Single-path incremental clustering experiment. The single-path incremental clustering algorithm is quite interactive, and the efficiency of the reverse-engineering process based on this algorithm depends on the experience of the user configuring its parameters. In this section, we report on an experiment performed by a user who had no particular familiarity with the LOCIS system but was familiar with the overall CELLEST process and was given a tutorial on how to use LENDI. Out of the feature suite available in LENDI. This user used the following features to perform the single-path incremental clustering:

1. a3 is the exact text at middle of the topmost non-empty line on the screen instance, which LENDI identified as a potential code or title for the LOCIS trace.
2. c1 encodes the cursor's label.
3. d2 encodes the vertical all-character binary profile.
4. d5 encodes a special character binary profile. This chosen character is the most frequent on the snapshot among a default or user defined set of characters.



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

(b) The corresponding part of the state-transition model. * indicates a required argument and *!* indicates a required number.

Figure 8. Diagrammatic representation of a partial LOCIS trace and its model.

Table 2. The setup for the single-path incremental clustering algorithm for the LOCIS experiment.

	Feature employed	Weight	Ignore if empty
1	a3	30	N
2	c1	10	N
3	d2	20	N
4	d5	10	Y
5	e1	5	Y
6	e2	25	N

5. e1 encodes the layout description of a snapshot and e2 encodes the specification of this description if it is a list or a table.

A threshold of 40% was used along with the choice that the cluster centroid will be its representative. It took eight recognition/review/reconfiguration rounds to reach the setup shown in Table 2, which the user thought was satisfactory. The column "Ignore if empty" indicates whether to ignore a feature if missing on some snapshot, or not.

The partition produced by the final configuration consisted of 23 different clusters. It included 17 mis-clustered snapshots (4.2%) and six redundant clusters. A mis-clustering is a false positive error that assigns snapshots with potentially different behaviors to the same screen cluster, causing false connections between the state-transition graph nodes. Redundant clusters are considered false negative errors which are duplications in the state-transition graph, resulting from the snapshots of the same screen being split into two or more clusters. This partition was reviewed by the user; 12 corrective operations, i.e., cluster joins and moves of instance groups, are necessary to fix the errors identified by the user. After, moving the mis-clustered instances to their clusters, a signature was calculated for every cluster. Then the signature generated was used to identify the instances in the training set, i.e., the same trace that was used to build the model. One snapshot was misclustered (0.25%). A measure of the algorithm's performance on unseen test data was obtained with repeated 10-fold cross validation. 10-fold cross validation on LOCIS yielded an error rate of (8%) (see Table 3) on the data in the test sets for the single-path incremental clustering algorithm.

Two-phase top-down clustering experiment. For the same LOCIS trace, 39 single-element features^2 were extracted for every snapshot of the trace. Then, the top-down clustering algorithm was applied to the data with a threshold of 17 clusters. The result was that 14 (3.4%) snapshots were clustered into 3 redundant clusters. On the other hand, 44 (10.8%) snapshots were misclustered. Ignoring the 3 unnecessary splits, we can say that 89.2% of the instances were "correctly" clustered. The partition was again reviewed and revised by a user. Using QANDA, the user corrected the preliminary clustering of the LOCIS trace and MoJoPlus inferred the operations necessary to obtain the desired authoritative partition. The tree-revision algorithm was applied, and a new tree containing 46 nodes and having a maximum depth of 12 was produced. When this decision tree was tested on the 406

Table 3. A comparison of the two clustering algorithms implemented in LENDI against one another and against C4.5.

Clustering method	MoJo plus	Training error (%)	Test error (%)
(a) Using a LOCIS trace, consisting of 406 snapshots belonging in 17 distinct screens			
Two-phase top-down clustering	20	0.00	3.4
Incremental clustering with signatures	12	0.25	8
C4.5 (supervised learning)	–	1.20	2.4
(b) Using a HOLLIS trace, consisting of 542 snapshots belonging in 29 distinct screens			
Two-phase top-down clustering	50	0.0	5.4
Incremental clustering with signatures	92	0.6	1.7
C4.5 (supervised learning)	–	1.0	4.3

snapshots, all were correctly classified. 10-fold cross validation on LOCIS yields an error rate of 3.4% on the data in the test sets (see Table 3).

Table 3 reports on the results of two experiments: one with the LOCIS library (a) and another with the Harvard library (HOLLIS, hollis.harvard.edu) (b). For each of the two clustering algorithms, the table reports on the distance of the partition produced by the algorithm from the authoritative partition produced after the user's review and revisions (column 2), the error when the revised partition was used to classify the training examples (column 3), and the average test error with 10-fold cross-validation experiments (column 4). As can be seen from the table, in both experiments the test error of the single-path algorithm was smaller than the test error of the top-down algorithm but its training error was larger in both cases. This implies that the signature generation process tends to overfit the training data and is not as effective in recognizing new snapshots.

In addition, we compared the behavior of these two algorithms against C4.5 (Quinlan, 1993), a standard decision-tree learning algorithm. Note that C4.5 requires all examples to be labeled, so it cannot be used as an alternative to our clustering algorithms that start by not knowing the screen in which each snapshot belongs. However, it could be used to build a decision tree after the authoritative partition has been established by the user, after reviewing the output either of the incremental single-path algorithm or of the top-down algorithm. It could, in effect, substitute the signature generation step in the incremental single-path algorithm and the tree-revision step in the top-down algorithm. As can be seen from the data, the top-down clustering algorithm and its tree-revision method is comparable to C4.5, although slightly less effective. This implies that substituting the tree-revision step with the application of C4.5 on the classified snapshots, after the user has revised the produced partition, would produce the most effective classifier for new snapshots.

Considering the variety of practices used in designing legacy user interfaces, complete automation of the reverse-engineering process is not possible. There will always be a need for some user feedback to complete the user-interface model. Developing better feature sets and smarter clustering methods can reduce the user input. However, the quality of the screen-identification result ultimately depends on the sufficiency of the input traces; if the recorded traces do not provide a sufficient number of screen snapshots, clustering may miss screens and consequently, if instances of the omitted screens are encountered later at run time, they will be misclassified. “Sufficiency of example traces” is the fundamental assumption of the overall process, and, at this point, we have only a methodological answer to it; namely, we assume that the recording emulators stay in place for a sufficiently long time, so that an appropriate number of traces is collected. Errors due to lack of sufficient traces can be corrected using the QANDA tool.

Even when clustering has correctly identified all user-interface screens however, the induced classifier that will be used to recognize new snapshots at run time may contain errors, irrespective of whether it is a signature pattern or a decision tree. Even the best of several C4.5 versions that we implemented could not fully generalize and still had 2.4% error in the LOCIS example. Again this problem is due to the insufficiency of the recorded snapshots. If the features do not correlate well with the commonalities of the instances in a cluster or if some clusters have only 1 or 2 snapshots in the trace, then the model produced will not be free of generalization (test) error. To eliminate classification error, the induced classifier may be “manually” corrected with additional features.

In general, however, it is unlikely that the complete legacy interface will have to be migrated. Usually only specific user tasks have to be made available through the web, and 100% snapshot classification correctness is essentially required only for the screens included in the navigation of these tasks’ executions. Thus, practically, the collection of sufficient examples for these specific tasks does not present a major challenge.

4. Interface migration to the web

Although this paper focuses on the CELLEST process for reverse engineering legacy user interfaces as implemented in LENDI, to illustrate how this process supports the overall objective of legacy migration to the web, we discuss the process following LENDI’s user-interface reverse engineering, namely, the development of the new, web-accessible user interface front-end.

As discussed in Section 3, LENDI’s process produces a state-transition model of the legacy interface behavior. The states of the model are the unique interface screens, and using the classifier induced by LENDI—either cluster signatures or a decision tree—each new snapshot can be recognized as an instance of these states. The transitions of the model correspond to the commands of the interface command language, and using the extracted action patterns each new user key-stroke on a snapshot can be recognized as an instance of a particular command that should lead to an instance of the command’s destination screen. This model constitutes a “map” of how users navigate through the legacy-interface screens to accomplish their tasks. Based on this intuition, the next step in the CELLEST process is to model specific user tasks in terms of the screens that the user has to navigate through and the information he/she has to exchange with the legacy interface. It further uses these task models as the basis for generating new, web-accessible front-ends for the legacy interface, thus completing the overall interface-migration process. In this section, we briefly describe Processes T2 and T3 in figure 1 to illustrate how the interface model produced by the reverse-engineering process implemented in LENDI provides the foundation for the subsequent phase of interface migration. Two systems have been developed to implement processes T2 and T3, URGENT (User interface ReGENeration Tool) (Kong et al., 1999) and its successor, MATHAINO (Kapoor and Stroulia, 2001; Stroulia and Kapoor, 2002). Both these systems share the same task-analysis step, but differ slightly on the type of the new user interfaces they produce.

4.1. Task analysis

The objective of the task-analysis step (Kapoor and Stroulia, 2001; Stroulia and Kapoor, 2002) is to model the information exchange that takes place between the user and the legacy application while the user interacts with the interface to accomplish a specific task. This step requires as input a set of task-specific traces recorded by the emulator, much in the same way as required by LENDI, but with two differences. First, the snapshots that appear on the user's terminal interface during the task execution are recognized as instances of the legacy-application screens. Second, the user has to highlight the areas of interest on these screens, in order to indicate the pieces of information displayed by the interface relevant for accomplishing the task in question.

The basic intuition underlying the task-analysis step is that during the task execution, the user navigates through a path of screens and executes a sequence of actions, in order to provide(obtain) some pieces(s) of information to(from) the system. The derived task model specifies

- the navigation path that the task execution follows, i.e., the sequence of legacy screens that the user visits to accomplish the task (note that this navigation path is a subgraph of the overall interface state-transition model), and
- the problem variables exchanged between the user and the legacy interface and their scope, and how exactly the information exchange is implemented in the original legacy user interface.

To identify the task navigation path, the screen sequences of the different example task traces are compared. A simple algorithm, similar to the Unix diff, is used to identify the alternative paths that can be used to accomplish the task in question.

The first objective of the information-exchange analysis step is to identify the variables exchanged between the user and the legacy interface during the task execution. The assumption is that there are as many input variables as there are distinct input values provided by the user to the application. Similarly, it is assumed that there are as many output variables as the screen areas highlighted by the user. Given that assumption, the objective becomes to more precisely characterize how each variable is used during the execution of the task. The different variables are classified as follows:

- Constants: a variable, whose value is the same in all the corresponding actions of all example task traces.
- Enumerated: a variable that takes values from a set for all example traces.
- Derived: a variable, whose value is obtained through an information-acquisition action and is subsequently provided to the legacy interface through an information-input interaction.

- Redundant: a variable, whose value is provided as input to the system through multiple information-input interactions.
- Unpredictable: an independent variable provided as input by the user.

At this point, an expert user can use the QANDA system to review the identified problem variables, correct their scope and annotate them with semantic information such as meaningful names, for example.

The second objective of the information-exchange analysis is to specify how exactly the information is exchanged, i.e., where on the screen it is displayed (in the case of output) and where it must be entered (in the case of input). User input usually occurs in static screen locations in the legacy interface. Information display, on the other hand, may appear in static or dynamic locations. If the screen where the information appears is static, then the information in question always appears in the same location that can be specified in terms of static x, y coordinates. If the screen is dynamic, the location of the information may vary but often it is still possible to specify it in terms of a pattern defined relative to static "landmarks" of the screen. When the location of the displayed information can be specified, either in terms of static coordinates or relative to screen "landmarks", the information can be automatically extracted from the original interface. For all the pieces of information that appear in dynamic locations in the screen, the whole legacy screen can be shown to the user to select them. The resulting task model is, in effect, executable: given all the unpredictable variables, it can be used as a driver of the original legacy interface to accomplish the analyzed task.

4.2. GUI specification and generation

After the analysis of a task, an abstract specification of a GUI is generated. Based on the variables identified in the previous step, a set of heuristics are used to identify widgets that could be used to implement the information-exchange interaction. At this point, some optimization of the current system-user interaction can be accomplished: the new interface will automatically retrieve from the legacy interface all the variables displayed at static positions on the screen and will feed them as input to the appropriate front-end widgets. It will also receive the variables input by the user and feed them to the appropriate locations of the screens in the underlying legacy interface. Finally, for all the variables that appear in dynamic locations on the screen, the new interface will expose the underlying screen for the user to select them. In the new interface, the user will be required to input the problem variables to the system only once. The new interface will buffer all the recurrent variables and will deliver them to all screens that use them. At the GUI specification phase, for all problem variables exchanged during the task, depending on their type, a class of graphical interaction objects, appropriate for the data-entry action at hand, are identified (Lewis and Rieman, 1993). For example, for an input action manipulating a date, appropriate graphical objects might be a calendar, a combination of three scrolling lists for year, month and day selection, a simple text-entry box, etc. The selection of the actual widget is based on a set of heuristics. The selected widgets are placed in forms, each of which traverses a segment of the task navigation path. The number of forms, their corresponding path segments and their relative order are decided based on a set of heuristics that ensure that no form can be submitted before all the input(output) information necessary for the execution of its corresponding navigation path has been entered by (displayed to) the user (Kapoor and Stroulia, 2001).

On the basis of the GUI specification, the final step in the process is the actual generation of a GUI. URGENT uses a collection of java beans to implement a set of simple graphical objects, including text and password entry boxes, radio and combo boxes, and display fields. The result is an applet through which the user can perform the task at hand (Kong et al., 1999). MATHAINO extends this work by providing a set of components that act as web-accessible servers, interpreting the abstract GUI specification so that it can be accessed from a variety of platform-specific thin clients, such as HTML browsers and WAP-enabled devices (Kapoor and Stroulia, 2001; Stroulia and Kapoor, 2002).

Because of the more user-intensive nature of the interface-migration process, the nature of our experimentation and evaluation of URGENT and MATHAINO is different. We have a number of case studies. We were able to collect task-specific traces for six different tasks on four different systems with ASCII-based interfaces. In addition to the accident-report task discussed below, we have used URGENT and MATHAINO to construct wrappers for several Unix tasks, such as reading and sending email messages using pine and examining the contents of directories and files, for a job-search application running on a mainframe, and for several variations of book searches in the Harvard library (Kong et al., 1999; Kapoor and Stroulia, 2001; Stroulia and Kapoor, 2002). Our experience with all these case studies has been that, with a small number of examples—usually around five—and with an effort of only a few hours, a legacy task could be wrapped with a simpler, form-based front-end.

4.3. Migration of the "accident report" task of the "legacy insurance application"

Let us now illustrate the interface-migration process using our "Legacy Insurance Application" case study. A small part of the interface state-transition model constructed by LENDI for the insurance system is diagrammatically depicted in figure 9. The notation used in

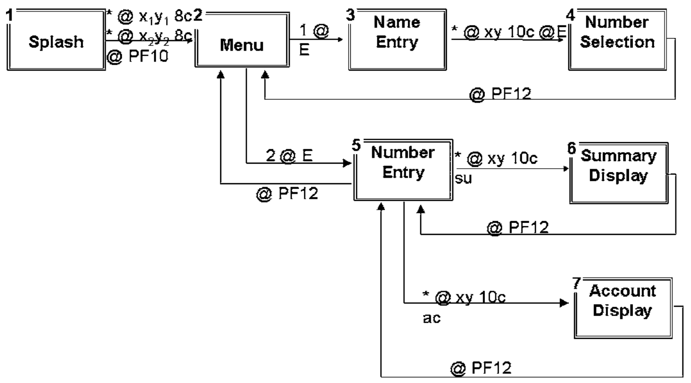
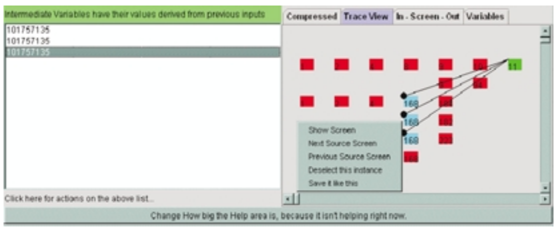


Figure 9. The interface state-transition diagram for the insurance system.

figure 9 is the same as the notation used in figure 6. Additionally, "* @ x1y1 8c" means that some undefined string whose maximum length is 8 characters must be entered at location x1, y1 on the screen. The system, in its initial behavioral state presents the "SPLASH" screen-state. The only possible transition from this state is to the "menu" screen-state, achieved through typing two strings at two different locations on the screen and pressing the "F10" function key.

Task analysis. The task-analysis step is initiated by providing a set of task-execution examples and does not require any further interaction until it is completed. Then, the QANDA system provides four different views on the task model and the corresponding navigation path: (a) a "compressed" view, where only the different screens are shown, (b) a "detailed" view that shows each snapshot in the trace, (c) an "information flow" view that is basically the "detailed" view annotated with arrows illustrating the flow of the problem variables from one snapshot to another, and (d) an "annotation" view which allows the editing of the problem variables. Figure 10 shows the detailed QANDA view of the recorded trace on the right pane. In addition, it shows the flow of some problem variable, whose values are shown on the left pane. This variable is displayed to the user on screen *11* (top right) and is provided as input to the application in screen *168* (bottom middle), which is visited three times. The QANDA user can verify or correct the scope and the flow of the selected variable.

A diagrammatic representation of the task model produced for the accident-report example of the insurance legacy system is shown in figure 11. The task model of figure 11 is


Intermediate Variables have their values derived from previous inputs

101757135
101757135
101757135

Click here for actions on the above list...

Change How big the Help area is, because it isn't helping right now.

Figure 10. Reviewing the task model of the accident-report task.

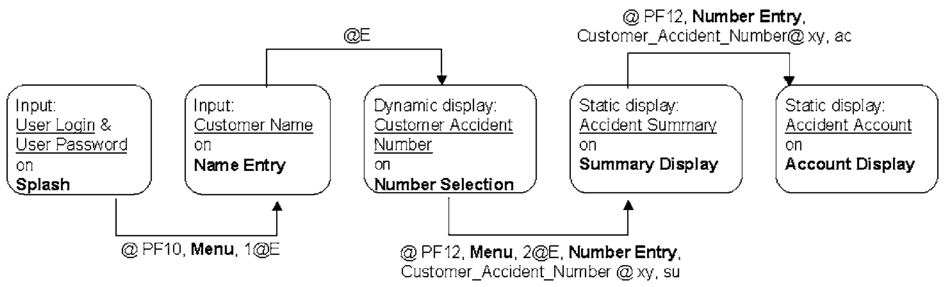
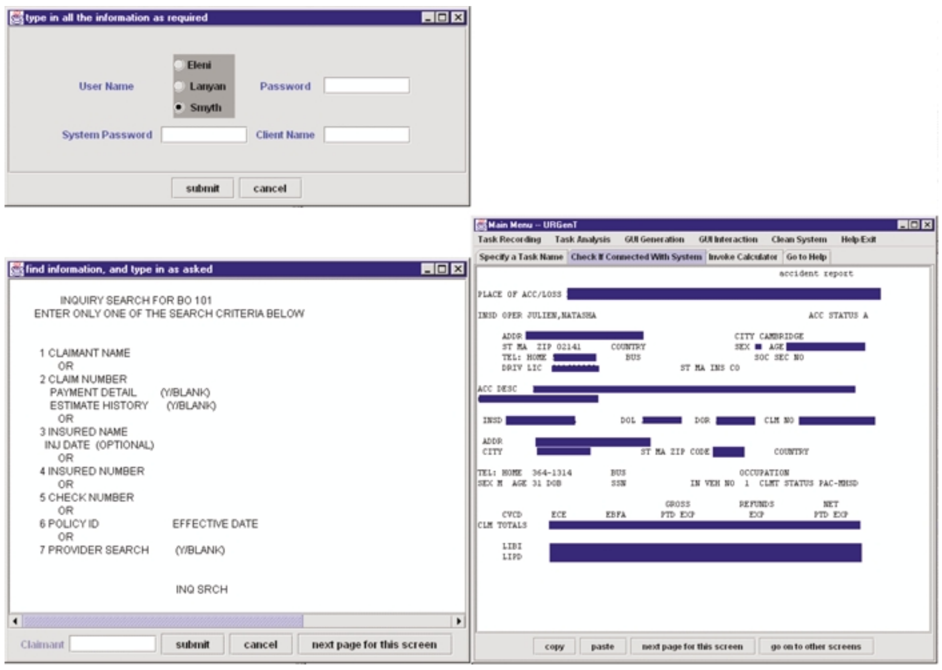

Input: User Login & User Password on **Splash**
Input: Customer Name on **Name Entry**
Dynamic display: Customer Accident Number on **Number Selection**
Static display: Accident Summary on **Summary Display**
Static display: Accident Account on **Account Display**
@E
@ PF12, **Number Entry**, Customer_Accident_Number@ xy, ac
@ PF10, **Menu**, 1@E
@ PF12, **Menu**, 2@E, **Number Entry**, Customer_Accident_Number @ xy, su

Figure 11. The task model of the accident report task in the insurance system.

based on the state-transition model of figure 9. This model depicts the sequence of necessary system-user interactions, involving the exchange of problem and user variables. Interactions related to system constants are hidden because they can be automated in the new interface front end, thus relieving the user from having to remember and repeat tedious steps. So for this particular task, the user has to input his/her "user-id" and "password", which will be transferred at particular locations of the screen "SPLASH" of the legacy interface. Then through a sequence of two actions, i.e., pressing "F10" at the "SPLASH" screen and typing 1 and "Enter" at the resulting screen "Menu", the user will reach the screen "Name Entry", where he/she will input the "customer name" variable. Then, by pressing "Enter", the user will move to the "Number Selection" screen where he/she will select from a dynamic screen the customer's accident number. Next, after pressing "F12" the user will return to the "Menu" screen, where he/she will select option "2" and press "Enter" to go to the "Number Entry" screen, where he/she will input the selected accident number, and type "xs". This action will cause the application to move to the "Summary Display" screen, where the "accident summary" is displayed at a standard location. Finally, after pressing "F12" the user will return again to the "Number Entry" screen, where he/she will input the selected accident number and then "ac", which will lead the application to the "Account Display" screen", which displays at a static location the accident accounting.

GUI generation. Finally, when the user completes the task-analysis review, an abstract user interface is generated, consisting of widgets appropriate for the data types of the problem variables involved. The user can review and, to some extent, modify the choices of widgets. Figure 12 shows the interface constructed for the accident-report task. This interface consists of three forms. The first one enables the user to input all the variables that need to be "told" to the system. The second exposes the legacy screen where claimant names are associated with claim numbers, so that the user can select the number on which the report should be generated. Finally the third window contains all the information of interest to the user collected from the two last screens (i.e., sum-display and acct-display).[3] This interface greatly simplifies the system-user interaction. In the legacy interface the user has to navigate through 7 screens and 9 transitions, where in the new interface the user has to go through only 3 forms and 2 transitions. Note that the sequence of the forms preserves the logic of the legacy interface dialog: the output information to be displayed to the user defines boundary between two subsequent forms. The underlying assumption is that, if some user input was entered after some system output in the original legacy interface, then there may be a dependency between them and the same order is preserved in the new front end. This is a conservative heuristic, and it may miss the opportunity to optimize the front-end design but it will not violate the legacy application logic (Kapoor and Stroulia, 2001). According to this heuristic, the first form of the accident-report task (shown in figure 12) contains widgets to receive all input information provided in the first two screens of the legacy interface (shown in figure 11). After this input some information is displayed in the third legacy screen of the task; the corresponding display widget and all subsequent input before the next display are included in the second form.



User Name: ( ) Eleni ( ) Lanyan [✓ Smyth]   Password
System Password   Client Name

```
        INQUIRY SEARCH FOR BO 101
    ENTER ONLY ONE OF THE SEARCH CRITERIA BELOW


     1 CLAIMANT NAME
         OR
     2 CLAIM NUMBER
       PAYMENT DETAIL       (Y/BLANK)
       ESTIMATE HISTORY      (Y/BLANK)
         OR
     3 INSURED NAME
      INJ DATE  (OPTIONAL)
         OR
     4 INSURED NUMBER
         OR
     5 CHECK NUMBER
         OR
     6 POLICY ID                  EFFECTIVE DATE
         OR
     7 PROVIDER SEARCH            (Y/BLANK)


                                  INQ SRCH
```

Claimant

```
                                                   accident report

PLACE OF ACC/LOSS [redacted]

INSD OPER JULIEN,NATASHA                                ACC STATUS A

    ADDR [redacted]                            CITY CAMBRIDGE
    ST MA  ZIP 02141     COUNTRY               SEX [redacted] AGE [redacted]
    TEL: HOME [redacted]   BUS                    SOC SEC NO
    DRIV LIC [redacted]               ST MA INS CO

ACC DESC [redacted]

 INSD [redacted]        DOL [redacted]  DOR [redacted]  CLM NO [redacted]

 ADDR       [redacted]
 CITY       [redacted]    ST MA ZIP CODE [redacted]     COUNTRY

TEL: HOME  364-1314     BUS                  OCCUPATION
SEX M  AGE 31 DOB       SSN          IN VEH NO  1  CLMT STATUS PAC-MHSD

                                  GROSS        REFUNDS       NET
    CVCD    ECE      EBFA        PTD EXP        EXP        PTD EXP
CLM TOTALS [redacted]

    LIBI   [redacted]
    LIPD
```

Figure 12. The three forms of the new graphical user-interface front end, for the accident-report task.

5. Summary and conclusions

In this paper, we described the CELLEST process for reverse engineering the system-user interaction of legacy systems with text-based interfaces and for wrapping task-specific segments of this interaction with new web-accessible front-ends. This method consists of the following steps: First system-user interaction traces are unintrusively collected by a middleware. Next the dynamic behavior of the system interface is reverse engineered, in terms of the screens it presents to the user and the navigation it allows through them. Finally, task-specific navigation paths are analyzed in order to extract a model of the task, in terms of the interface navigation and the information exchange it implies and an appropriate web-based interface is constructed for wrapping this navigation and enabling its execution from a standard web browser.

The CELLEST process uses artificial-intelligence algorithms to leverage and advance similar manual industrial practices, such as screen scraping. These industrial practices are not automated: they assume that an analyst will model the user interface behavior and will develop the new application to interpret and drive this behavior. Therefore, they are labor-intensive and error prone. The novelty of the CELLEST approach lies in the following three important contributions:

- It is highly automated. As a result it is light-weight in terms of the skills it assumes since it requires more an expert-user's familiarity with the application interface as opposed to a software-developer's understanding of the interface-behavior implementation. In fact, it has been our experience that junior members of our research group could use the environment effectively within a few of days of practice. With legacy systems, developed over a long period of time by different people, the latter type of knowledge is often unavailable where the former is not.

- It constructs a high-level, intermediate abstraction of the legacy system behavior, to support interface migration. Understanding the system-user interaction at the level of information exchange enables the simultaneous migration to multiple target platforms. Furthermore, the new user interfaces can be more tailored to their native look and feel and don't have to mimic closely the original legacy user interface.
- It is code-independent, and therefore its applicability is not constrained by programming-language details. This advantage comes at the cost of being essentially limited to as-is user-interface migration, as opposed to code-based analysis techniques that are applicable to more general reengineering and maintenance problems. However, for purposes such as web-enabling and lightweight system integration, "code understanding" is an expensive and possibly brittle approach to interface reverse engineering and migration. Our experiments with the CELLEST environment indicate that "trace understanding" is an effective alternative.

The applicability of this method depends primarily on the generality of the recording component, and secondarily, on the feature set we have developed for recognizing screens. We have successfully tested our recording and feature extraction components with block-mode data transfer protocols or those which can be emulated in block-mode. Specifically we applied our method to IBM 3270 and VT100. The challenge in scroll mode data transfer protocol, e.g. IBM 5250 and VT100 is to define a concept corresponding to the "snapshot" concept and, possibly, to define an extended set of features on it. A potential solution, which we have already explored in reverse-engineering VT100 interfaces, is to define as a received snapshot the contents of the terminal buffer just before each user action. We believe that similar approaches could be employed with other protocols, such as AS400 for example, although more experimentation is clearly necessary.

The method's usefulness lies in its ability to construct a high-level model of the interaction behavior between the legacy system and its users. Instead of analyzing the widgets implementing the legacy interface so that they can be replaced by functionally similar widgets in a target platform, an approach that has been the state of the art until today in interface migration, our method constructs a model of the interface behavior in terms of behavioral states and possible commands that enable the transitions between them. Thus, instead of replicating the same interaction with different widgets in new platforms, we can encapsulate interesting behavioral segments with new user-interface front-ends on different platforms. More importantly, this high-level interaction model can be simultaneously migrated to multiple platforms at once.

On the other hand, as is common with wrapping approaches, the user-interface front ends developed with CELLEST may suffer a performance cost due to the introduction of a new layer on top of the existing legacy user-interface; the new front-end layer access the application through the legacy protocol emulator. This cost however is marginal when compared to the latency introduced by the network separating the new web-based interface and the host; this latter cost is unavoidable when web-based access to the application host is required.

The work reported in this paper is still in progress, but the results of our initial experimentation with the interface reverse engineering, task-modeling and interface migration processes are quite promising, and we continue to develop and evaluate this process.

Finally, because exactly this work has been motivated by a partnership with an industrial sponsor and its methodology is inspired by industrial practices in the area, we believe that the CELLEST process can potentially have an impact to legacy migration practices.

Acknowledgments

This work was supported by a generous contribution from Celcorp (www.celcorp.com), a Collaborative Research and Development grant by NSERC 215451-98 and ASERC, the Alberta Software Engineering Research Consortium.

Notes

1. In our more recent work (El-Ramly et al., 2002), we have developed a data-mining algorithm to discover similar interaction segments in the recorded trace. Such similar segments can play the role of task examples.
2. These features are the full suite of LENDI's features, discretized in order for their similarity to be evaluated as a binary question.
3. Some areas in the last two windows are blacked-out because the data they contain are private.

References

Antoniol, G., Fiutem, R., Merlo, E., and Tonella, P. 1995. Application and user interface migration from basic to visual C++. In *Proceedings of the 1995 International Conference on Software Maintenance*, Nice, pp. 76–85.

El-Ramly, M., Iglinski, P., Stroulia, E., Sorenson, P., and Matichuk, B. 2001. Modeling the system-user dialog using interaction traces. In *Proceedings of the 8th Working Conference on Reverse Engineering*, Stuttgart, Germany: IEEE Computer Society Press, pp. 208–217.

El-Ramly, M., Stroulia, E., and Sorenson, P. 2002. From run-time behavior to usage scenarios: An interaction-pattern mining approach. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Canada: Edmonton, AB, pp. 315–324.

Gannod, G., Mudiam, S., and Lindquist, T. 2000. An architectural-based approach for synthesizing and integrating adapters for legacy software. In *Proceedings of the 7th Working Conference in Reverse Envineering*, Brisbane, Australia, pp. 128–137.

Horowitz, E. 1998. Migrating software to the World Wide Web. *IEEE Software*, 15(3):18–21.

Kapoor, R.V. and Stroulia, E. 2001. Mathaino: Simultaneous legacy interface migration to multiple platforms. In *Proceedings of the 9th International Conference on Human-Computer Interaction*, New Orleans, LA, USA, vol. 1, pp. 51–55.

Kieninger, T. 1998. Table structure recognition based on robust block segmentation. In *Proceedings of Document Recognition V, SPIE*, San Jose, CA, vol. 3305, pp. 22–32.

Kong, L., Stroulia E., and Matichuk B. 1999. Legacy interface migration: A task-centered approach. In *Proceedings of the 8th International Conference on Human-Computer Interaction*, Munich, Germany, pp. 1167–1171.

Lewis, C. and Rieman, J. 1993. Task-centered user interface design. http://www.acm.org/perlman/uidesign.html.

Merlo, E., Gagn, P.Y., Girard, J.F., Kontogiannis, K., Hendren, L.J., Panangaden, P., and De Mori, R. 1995. Reverse engineering and reengineering of user interfaces. *IEEE Software*, 12(1):64–73.

Moore, M., Rugaber, S., and Seaver, P. 1994. Knowledge-based user interface migration. In *Proceedings of the 1994 International Conference on Software Maintenance*, Victoria, BC, pp. 72–79.

Phanouriou, C. and Abrams, M. 1997. Transforming command-line driven systems to web applications. *Computer Networks and ISDN Systems*, 29(8–13):1497–1505.

Pressman R. 1996. *Software Engineering: A Practitioner's Approach*, 4th edn. McGraw-Hill Companies.

Quinlan, J. 1993. *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

van Rijsbergen, C.J. 1979. *Information Retrieval*, 2nd edn. London: Butterworths.

Srihari, S., Lam, W., Govindaraju, V., Srihari, R., and Hull, J. 1992. Document image understanding. Tech. Report CEDAR-TR-92-1, Center of Excellence for Document Analysis, State University of New York at Buffalo.

Stroulia, E. and Kapoor, R.V. 2002. Reverse engineering interaction plans for legacy interface migration. In *Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces*, Valenciennes, France, pp. 295–310.

Tan, Y.S., Lindquist, D.B., Rowe, T.O., and Hind, J.R. 1998. IBM eNetwork host on-demand: The beginning of a new era for accessing host information in a web environment. *IBM Systems Journal*, 37(1):133–151.

Tucker, K. and Stirewalt, K. 1999. Model based user-interface reengineering. In *Proceedings of the 6th Working Conference on Reverse Engineering*, Atlanta, Georgia, pp. 56–63.

Tzerpos, V. and Holt, R. 1999. MoJo: A distance metric for software clustering. In *Proceedings of the 6th Working Conference on Reverse Engineering*, Atlanta, Georgia, pp. 187–193.